IMS
Version 15.1.0

Database Administration

(January 27, 2018 edition)

IBM

IMS

Version 15.1.0

Database Administration

(January 27, 2018 edition)

IBM

Contents

About this information

These topics describe IMS™ database types and concepts, and also describe how to design, implement, maintain, modify, back up, and recover IMS databases.

This information is available in IBM® Knowledge Center.

Prerequisite knowledge

Before using this book, you should understand basic z/OS® and IMS concepts and your installation's IMS system. IMS can run in the following environments: DB Batch, DCCTL, TM Batch, DB/DC, DBCTL. You should understand the environments that apply to your installation. The IMS concepts that are explained in this information pertain only to administering an IMS database. You should know how to use DL/I calls and languages such as assembler, COBOL, PL/I, and C.

You can learn more about z/OS by visiting the "z/OS basic skills" topics in IBM Knowledge Center.

You can gain an understanding of basic IMS concepts by reading *An Introduction to IMS*, an IBM Press publication.

IBM offers a wide variety of classroom and self-study courses to help you learn IMS. For a complete list of courses available, go to the IBM Skills Gateway and search for IMS.

How new and changed information is identified

New and changed information in most IMS library PDF publications is denoted by a character (revision marker) in the left margin. The first edition (-00) of *Release Planning*, as well as the *Program Directory* and *Licensed Program Specifications*, do not include revision markers.

Revision markers follow these general conventions:
- Only technical changes are marked; style and grammatical changes are not marked.
- If part of an element, such as a paragraph, syntax diagram, list item, task step, or figure is changed, the entire element is marked with revision markers, even though only part of the element might have changed.
- If a topic is changed by more than 50%, the entire topic is marked with revision markers (so it might seem to be a new topic, even though it is not).

Revision markers do not necessarily indicate all the changes made to the information because deleted text and graphics cannot be marked with revision markers.

How to read syntax diagrams

The following rules apply to the syntax diagrams that are used in this information:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following conventions are used:
  - The >>--- symbol indicates the beginning of a syntax diagram.
  - The ---> symbol indicates that the syntax diagram is continued on the next line.
  - The >--- symbol indicates that a syntax diagram is continued from the previous line.
  - The --->< symbol indicates the end of a syntax diagram.
- Required items appear on the horizontal line (the main path).

```
►►──required_item──────────────────────────────────────────────►◄
```
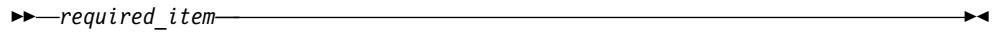
- Optional items appear below the main path.

```
►►──required_item──┬──────────────┬────────────────────────────►◄
                   └─optional_item─┘
```

If an optional item appears above the main path, that item has no effect on the execution of the syntax element and is used only for readability.

```
                   ┌─optional_item─┐
►►──required_item──┴───────────────┴────────────────────────────►◄
```
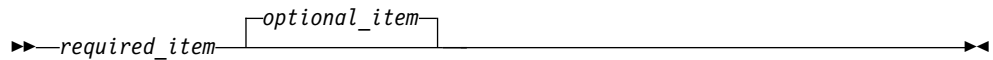
- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──required_item──┬─required_choice1─┬──────────────────────────►◄
                   └─required_choice2─┘
```
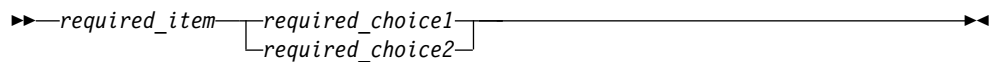
If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──required_item──┬──────────────────┬──────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```
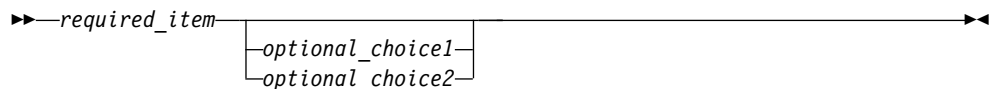
If one of the items is the default, it appears above the main path, and the remaining choices are shown below.

```
                   ┌─default_choice──┐
►►──required_item──┼─────────────────┼──────────────────────────►◄
                   ├─optional_choice─┤
                   └─optional_choice─┘
```
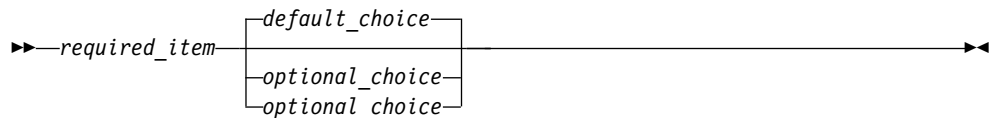
- An arrow returning to the left, above the main line, indicates an item that can be repeated.

```
                   ┌───────────────┐
►►──required_item──▼─repeatable_item─┴──────────────────────────►◄
```

If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
>>--required_item--+--repeatable_item--+------------------><
                   '--,--<
```
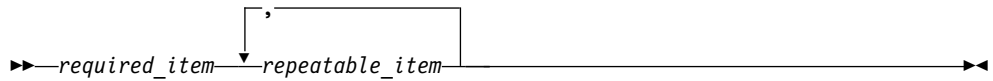
A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.

```
>>--required_item--| fragment-name |------------------------><
```

fragment-name:

```
|--required_item--+------------------+--|
                  '--optional_item--'
```
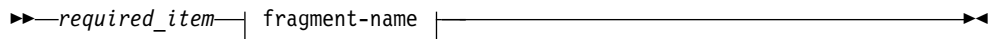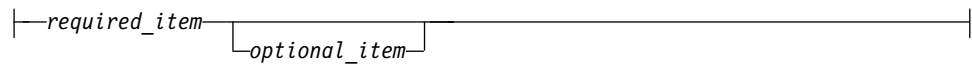
- In IMS, a b symbol indicates one blank position.
- Keywords, and their minimum abbreviations if applicable, appear in uppercase. They must be spelled exactly as shown. Variables appear in all lowercase italic letters (for example, *column-name*). They represent user-supplied names or values.
- Separate keywords and parameters by at least one space if no intervening punctuation is shown in the diagram.
- Enter punctuation marks, parentheses, arithmetic operators, and other symbols, exactly as shown in the diagram.
- Footnotes are shown by a number in parentheses, for example (1).

Accessibility features for IMS 15

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including IMS 15. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size.

Keyboard navigation

You can access IMS 15 ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the IMS 15 ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide Volume 1*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide

includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

Online documentation for IMS 15 is available in IBM Knowledge Center.

IBM and accessibility

See the *IBM Human Ability and Accessibility Center* at www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments

Your feedback is important in helping us provide the most accurate and highest quality information. If you have any comments about this or any other IMS information, you can take one of the following actions:

- Click the **Contact Us** tab at the bottom of any IBM Knowledge Center topic.
- Send an email to imspubs@us.ibm.com. Be sure to include the book title and the publication number.

To help us respond quickly and accurately, please include as much information as you can about the content you are commenting on, where we can find it, and what your suggestions for improvement might be.

Part 1. General database concepts, standards, and procedures

The following topics provide an introduction to the administration of databases using IMS Database Manager, including a general discussion of IMS databases, basic standards and procedures used when working with IMS databases, the basic design and review process, and database security.

Chapter 1. Introduction to IMS databases

The introduction to IMS databases describes the tasks of database administration and discusses the key concepts and terms used when administering IMS Database Manager.

Database administration overview

The task of database administration is to design, implement, and maintain databases.

This information describes the tasks involved in administering the Information Management System Database Manager (IMS DB). IMS is composed of two parts: IMS Database Manager and IMS Transaction Manager. IMS Database Manager manages the physical storage of records in the database. IMS Transaction Manager manages the terminal network, the input and output of messages, and online system resources. The administration of IMS Transaction Manager is covered in *IMS Version 15 System Administration*. IMS networking is covered in *IMS Version 15 Communications and Connections*.

This book presents the database administration tasks in the order in which you normally perform the tasks. You perform some tasks in a specific sequence in the database development process while other tasks are ongoing. It is important for you to grasp not only what the tasks are, but also how they interrelate.

This first part of the book provides important concepts and procedures for the entire database administration process. The second part contains the chapters corresponding to particular tasks of database administration.

Related concepts:

"Database administration tasks" on page 4

DL/I

Data Language/I (DL/I) is the IMS data manipulation language, which is a common high-level interface between a user application and IMS.

DL/I calls are invoked from application programs written in languages such as PL/I, COBOL, VS Pascal, C, and Ada. It also can be invoked from assembler language application programs by subroutine calls. IMS lets the user define data structures, relate structures to the application, load structures, and reorganize structures.

Related concepts:

Application programming for IMS DB (Application Programming)

Related reference:

Database management (Application Programming APIs)

CICS

Customer Information Control System (CICS®) accesses IMS databases through the database resource adapter (DRA).

CICS or other transaction management subsystems (excluding IMS Transaction Manager) can access IMS full-function databases and data entry databases (DEDBs) in a DB/DC or DBCTL environment through the DRA.

Whenever tasks differ for CICS users, a brief description about the differences is included.

DBCTL and DCCTL

Database Control (DBCTL) supports non-message-driven batch message processing (BMP) programs. Data Communications Control (DCCTL) is a transaction management subsystem that does not support full-function DEDBs or MSDBs (main storage databases), but does support GSAM databases in BMP regions.

DBCTL has its own log and participates in database recovery. Locking is provided by IMS program isolation (PI) or the internal resource lock manager (IRLM).

To access databases in a DCCTL environment, DCCTL must connect to an external subsystem that provides database support.

Open Database Access (ODBA)

Any program that runs in a z/OS address space can access IMS DB through the Open Database Access (ODBA) callable interface.

Any z/OS application program running in a z/OS address space that is managed by z/OS Resource Recovery Services (RRS) can access IMS full-function databases and data entry databases (DEDBs). z/OS application programs that use the ODBA interface are called ODBA applications.

From the perspective of IMS, the z/OS address space involved appears to be another region called the z/OS application region.

Types of programs that can call the ODBA interface include:
- Db2® for z/OS stored procedures, including COBOL, PL/I, and Java™ procedures
- Enterprise Java Beans running in WebSphere® Application Server for z/OS
- Other z/OS applications

Related tasks:

➡ Accessing IMS databases through the ODBA interface (Communications and Connections)

Database administration tasks

The database administration tasks relevant to IMS databases are listed in this topic.

Participating in design reviews
> Design reviews are a series of formal meetings you attend in which the design and implementation of the database are examined. Design reviews are an ongoing task during the design and implementation of a database system. They are also held when new applications are added to an existing system.

Analyzing data requirements
> After the users at your installation identify their data processing requirements, you will construct data structures. These structures show

what data will be in your database and how it will be organized. This task precedes the actual design of the database.

Designing your database
After data structures are identified, the next step is to design your database. Database design involves:

- Choosing how to physically organize your data
- Deciding which IMS processing options you need to use
- Making a series of decisions about design that determine how well your database performs and uses available space

Developing a test database
Before the applications that will use your database are cut over to production status, they should be tested. Depending on the form of your existing data, you can use one or more of the IMS Database Design Aids to design, create, load, and test your test database.

Implementing your database design
After your database is designed, implement the design by describing to IMS the database's characteristics and how application programs will use the database.

If your IMS system is enabled to manage application control blocks by using the catalog, you can use the IMS Enterprise Suite Explorer for Development to issue SQL statements that describe the databases and how applications use it.

However, if your IMS system uses an ACB library, this task consists of coding database descriptions (DBDs) and program specification blocks (PSBs), both of which are a series of macro statements. Another part of implementing the database design is determining whether to prebuild application control blocks (ACBs) of the database or to build the ACBs dynamically.

Loading your database
After database characteristics are defined, write an initial load program to put your data into the database. After you load the database, application programs can be run against it.

Monitoring your database
When the database is running, routinely monitor its performance. A variety of tools for monitoring the IMS system are available.

Tuning your database
Tune your database when performance degrades or utilization of external storage is not optimum. Routine monitoring helps you determine when the system needs to be tuned and what type of tuning needs to be done. Like monitoring, the task of tuning the database is ongoing.

Modifying your database
As new applications are developed or the needs of your users change, you might need to make changes to your database. For example, you can change database organization, database hierarchies (or the segments and fields within them), and you can add or delete one or more partitions. Like monitoring and tuning, the task of modifying the database is ongoing.

Recovering your database
Database recovery involves restoring a database to its original condition after it is rendered invalid by some failure. The task of developing recovery procedures and performing recovery is an important one.

However, because it is difficult to separate data recovery from system recovery, the task of recovery is treated separately in *IMS Version 15 Operations and Automation*.

You can use Database Recovery Control (DBRC) to support the recovery of your databases. If your databases are registered in the RECON data set, DBRC gains control during execution of these IMS utilities:

- Database Image Copy
- Online Database Image Copy
- Database Image Copy 2
- Change Accumulation
- Database Recovery
- Log Recovery
- Log Archive
- DEDB area data set create
- HD and HISAM Reorganization Unload and Reload
- HALDB Index/ILDS Rebuild

You must ensure that all database recoveries use the current IMS utilities, rather than those of earlier releases.

Establishing security
You can keep unauthorized persons from accessing the data in your database by using program communication blocks (PCBs). With PCBs, you can control how much of the database a given user can see, and what can be done with that data. In addition, you can take steps to keep non-IMS programs from accessing your database.

Setting up standards and procedures
It is important to set standards and procedures for application and database development. This is especially true in an environment with multiple applications. If you have guidelines and standards, you will save time in application development and avoid problems later on such as inconsistent naming conventions or programming standards.

Related concepts:

"Database administration overview" on page 3

Related reference:

Log Archive utility (DFSUARC0) (System Utilities)

Log Recovery utility (DFSULTR0) (System Utilities)

Database concepts and terminology

This topic discusses the terms and concepts you need to understand to perform IMS database administration tasks.

To understand this topic, you must know what a DL/I call is and how to code it. You must understand function codes and Segment Search Arguments (SSAs) in DL/I calls and know what is meant when a call is referred to as qualified or unqualified (explained in *IMS Version 15 Application Programming*).

How data is stored in a database

The data in a database is grouped into a series of *database records*. Each database record is composed of smaller groups of data called *segments*. A segment is the smallest piece of data IMS can store. Segments, in turn, are made up of one or more *fields*.

The following figure shows a record in a school database. Each of the boxes is a segment or separate group of data in the database record. The segments in the database record contain the following information:

COURSE
> The name of the course

INSTR
> The name of the teacher of the course

REPORT
> A report the teacher needs at the end of the course

STUDENT
> The names of students in the course

GRADE
> The grade a student received in the course

PLACE
> The room in which the course is taught
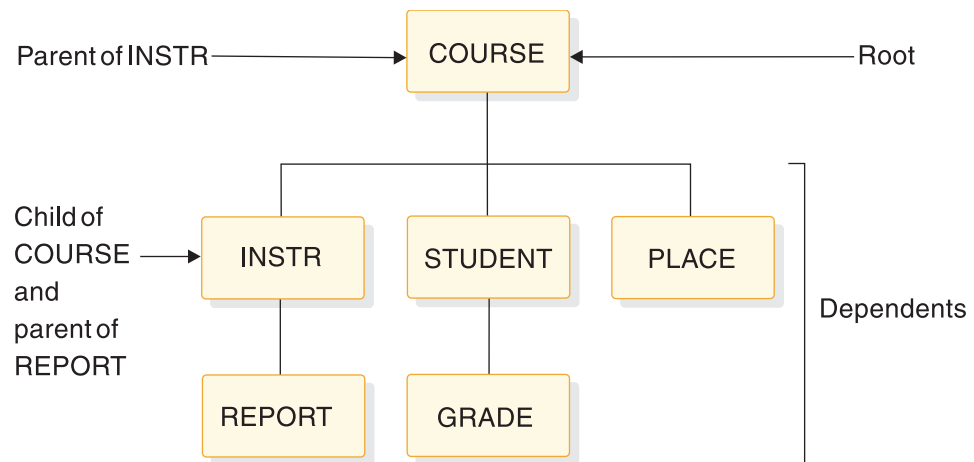
Figure 1. Segment types in the school database record

The segments within a database record exist in a *hierarchy*. A hierarchy is the order in which segments are arranged. The order implies something. The school database is storing data about courses that are taught. The COURSE segment is at the top of the hierarchy. The other types of data in segments in the database record would be meaningless if there was no COURSE.

Root segment

Only one *root segment* exists within a database record. All other segments in the database record are called *dependent segments*.

In the example shown in "How data is stored in a database," the COURSE segment is the root segment. The segments INSTR, REPORT, STUDENT, GRADE,

and PLACE are the dependent segments. The existence of dependent segments hinges on the existence of a root segment. For example, without the root segment COURSE, there would be no reason for having a PLACE segment stating in which room the course was held.

The third level of dependent segments, REPORT and GRADE, is subject to the existence of second level segments INSTR and STUDENT. For example, without the second level segment STUDENT, there would be no reason for having a GRADE segment indicating the grade the student received in the course.

Parent and child segment

Another set of words used to refer to how segments relate to each other in a hierarchy is *parent segment* and *child segment*. A parent segment is any segment that has a dependent segment beneath it in the hierarchy.

In the figure shown in "How data is stored in a database" on page 7, COURSE is the parent of INSTR, and INSTR is the parent of REPORT. A child segment is any segment that is a dependent of another segment above it in the hierarchy. REPORT is the child of INSTR, and INSTR is the child of COURSE. Note that INSTR is both a parent segment in its relationship to REPORT and a child segment in its relationship to COURSE.

Segment type and occurrence

The terms *segment type* and *segment occurrence* distinguish between a type of segment in the database and a specific segment instance.

This is in contrast to the terms root, dependent, parent, and child, which describe the *relationship* between segments.

The database shown in "How data is stored in a database" on page 7 is actually the design of the database. It shows the segment types for the database. "Relationship between segments" shows the actual database record with the segment occurrences.

A segment occurrence is a single specific segment. Math is a single occurrence of the COURSE segment type. Baker and Coe are multiple occurrences of the STUDENT segment type.

Relationship between segments

One final term for describing segments is *twin segment*. Twin (like root, dependent, parent, and child) describes a relationship between segments. Twin segments are multiple occurrences of the same segment type under a single parent.

In the following figure, the segments Baker and Coe are twins. They have the same parent (Math), and are of the same segment type (STUDENT). Pass and Inc are not twins. Although Pass and Inc are the same segment type (GRADE), they do not have the same parent. Pass is the child segment of Baker, and Inc is the child segment of Coe.
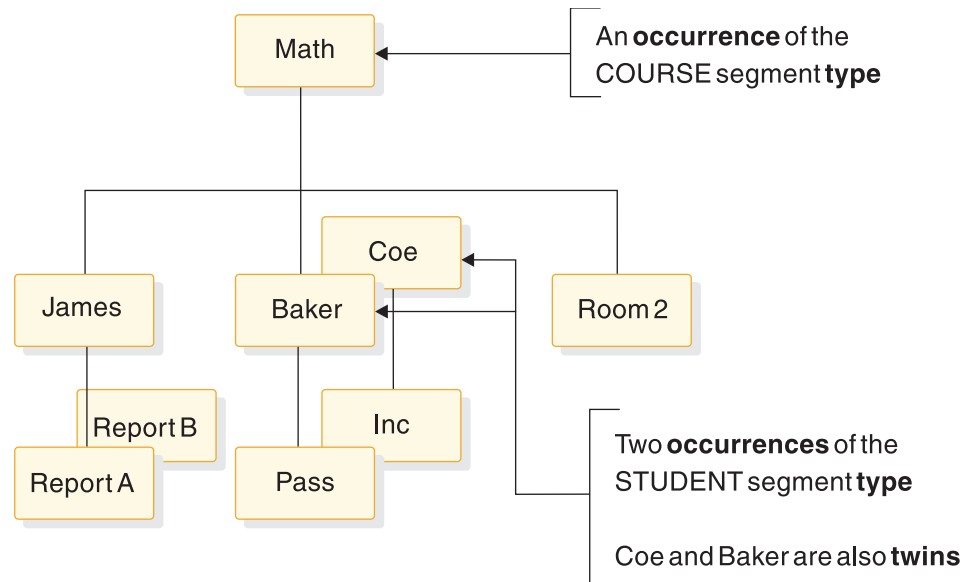
Figure 2. Segment occurrences in a school database record

The following topic discusses the hierarchy in more detail. Subsequent topics describe the objects in a database, what they consist of and the rules governing their existence and use. These objects are:

The database record

The segments in a database record

The fields within a segment

The hierarchy in a database record

A database is composed of a series of database records. Records contain segments, and the segments are arranged in a hierarchy in the database record.

Numbering sequence in a hierarchy: top to bottom

When a database record is stored in the database, the hierarchical arrangement of segments in the database record is the order in which segments are stored.

Starting at the top of a database record (at the root segment), segments are stored in the database in the sequence shown by the numbers in the following figure.

The sequence goes from the top of the hierarchy to the bottom in the first (left most) *path* or leg of the hierarchy. When the bottom of the database is reached, the sequence is from left to right. When all segments have been stored in that path of the hierarchy, the sequencing begins in the next path to the right, again proceeding from top to bottom and then left to right. (In the second leg of the hierarchy there is nothing to go to at the right.) The sequence in which segments are stored is loosely called "*top to bottom, left to right.*"

The following figure shows sequencing of segment types for the school database shown in "How data is stored in a database" on page 7. The sequence of segment types are stored in the following order:

1. COURSE (top to bottom)
2. INSTR
3. REPORT
4. STUDENT (left to right)

5. GRADE (top to bottom)
6. PLACE (left to right)
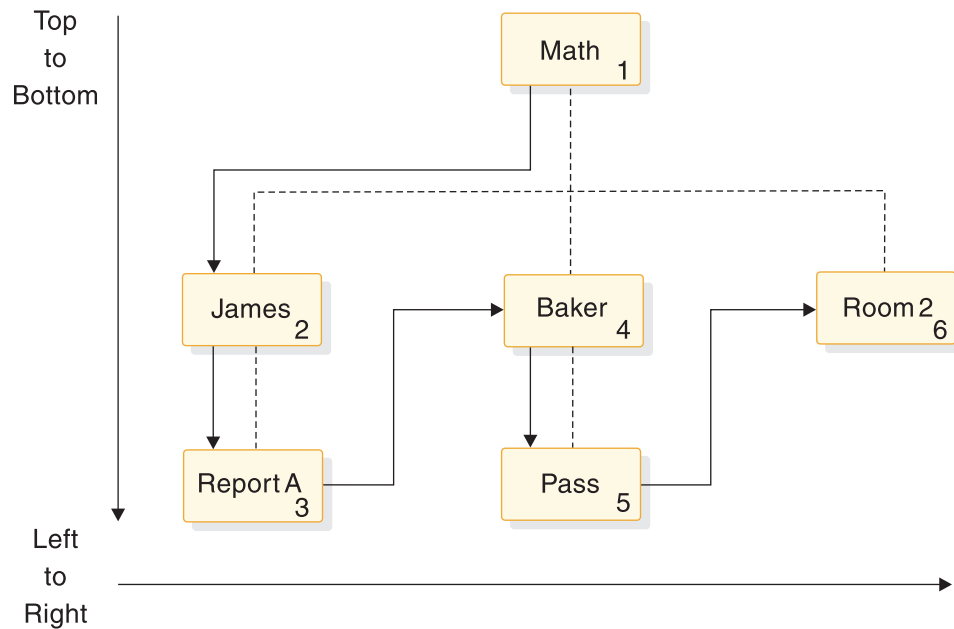
Figure 3. Hierarchical sequence of segment types for a school database

The following figure shows the segment occurrences for the school database record as shown in "Relationship between segments" on page 8. Because there are multiple occurrences of segment types, segments are read "front to back" in addition to "top to bottom, left to right." The segment occurrences for the school database are stored in the following order:

1. Math (top to bottom)
2. James
3. ReportA
4. ReportB (front to back)
5. Baker (left to right)
6. Pass (top to bottom)
7. Coe (front to back)
8. Inc (top to bottom)
9. Room2 (left to right)
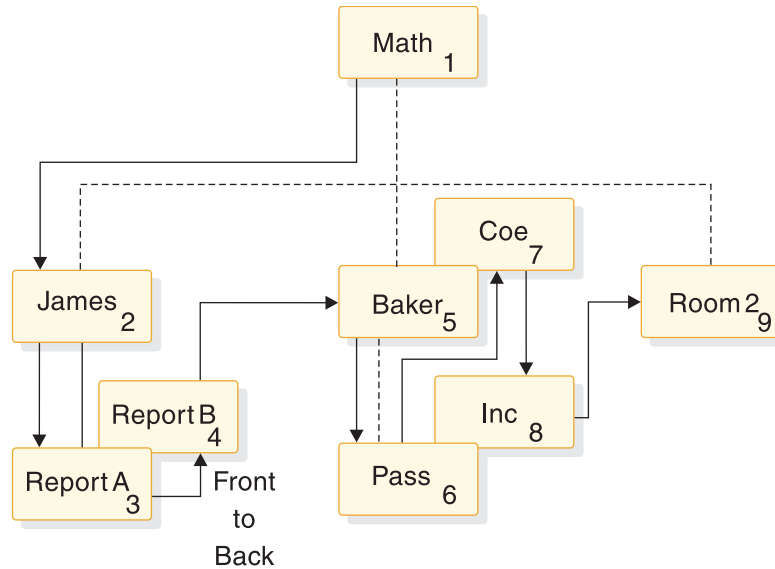
Figure 4. Hierarchical sequence of segment occurrences for school database

Note that the numbering sequence is still initially from top to bottom. At the bottom of the hierarchy, however, observe that there are two occurrences of the REPORT segment.

Because you are at the bottom of the hierarchy, both segment occurrences are picked up before you move to the right in this path of the hierarchy. Both reports relate to the instructor segment James; therefore it makes sense to keep them stored together in the database. In the second path of the hierarchy, there are also two segment occurrences in the student segment. You are not at the bottom of the hierarchical path until you reach the grade segment Pass. Therefore, sequencing is not "interrupted" by the two occurrences of the student segment Baker and Coe. This makes sense because you are keeping student and grade Baker and Pass together.

Note that the grade Inc under student Coe is not considered another occurrence under Baker. Coe and Inc become a separate path in the hierarchy. Only when you reach the bottom of a hierarchical path is the "top to bottom, left to right" sequencing interrupted to pick up multiple segment occurrences. You can refer to sequencing in the hierarchy as "top to bottom, front to back, left to right", but "front to back" only occurs at the bottom of the hierarchy. Multiple occurrences of a segment at any other level are sequenced as separate paths in the hierarchy.

As noted before, this numbering of segments represents the sequence in which segments are stored in the database. If an application program requests all segments in a database record in hierarchical sequence or issues Get-Next (GN) calls, this is the order in which segments would be presented to the application program.

Numbering sequence in a hierarchy: movement and position

The terms *movement* and *position* are used when talking about how segments are accessed when an application program issues a call. They are used to help describe the numbering sequence in a hierarchy.

When talking about movement through the hierarchy, it always means moving in the sequence implied by the numbering scheme. Movement can be forward or backward. When talking about position in the hierarchy, it means being located (positioned) at a specific segment.

A segment is the smallest piece of data IMS can store. If an application program issues a Get-Unique (GU) call for the student segment BAKER (see Figure 4 on page 11), the current position is immediately after the BAKER segment occurrence. If an application program then issues an unqualified GN call, IMS moves forward in the database and returns the PASS segment occurrence.

Numbering sequence in a hierarchy: level

In a hierarchy, *level* is the position of a segment in the hierarchy in relation to the root segment. The root segment is always on level one.

The following figure illustrates levels of the database record shown in "Relationship between segments" on page 8.
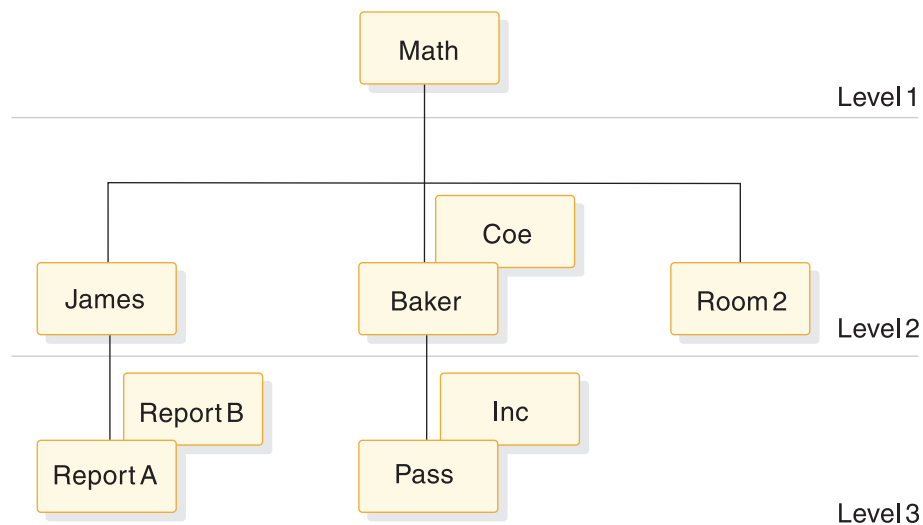
Figure 5. Levels in the database

Types of IMS databases

IMS allows you to define many different database types. You define the database type that best suits your application's processing requirements.

You need to know that each IMS database has its own access method, because IMS runs under control of the z/OS operating system. The operating system does not know what a segment is because it processes logical records, not segments. IMS access methods therefore manipulate segments in a database record. When a logical record needs to be read, operating system access methods (or IMS) are used.

The following table lists the IMS database types you can define, the IMS access methods they use and the operating system access methods you can use with them. Although each type of database varies slightly in its access method, they all use database records.

Table 1. Types of IMS databases and their z/OS access methods

Type of IMS database	Full name of database type	IMS or operating system access methods that can be used
DEDB [1]	Data Entry Database	Media Manager
GSAM	Generalized Sequential Access Method	QSAM/BSAM or VSAM
HDAM	Hierarchical Direct Access Method	VSAM or OSAM
HIDAM	Hierarchical Indexed Direct Access Method	VSAM or OSAM
HISAM	Hierarchical Indexed Sequential Access Method	VSAM
HSAM	Hierarchical Sequential Access Method	BSAM or QSAM
MSDB [2]	Main Storage Database	N/A
PHDAM	Partitioned Hierarchical Direct Access Method	VSAM or OSAM
PHIDAM	Partitioned Hierarchical Indexed Direct Access Method	VSAM or OSAM
PSINDEX	Partitioned Secondary Index	VSAM
SHSAM	Simple Hierarchical Sequential Access Method	BSAM or QSAM
SHISAM	Simple Hierarchical Indexed Sequential Access Method	VSAM

Table notes:

1. For DBCTL, available only to BMPs
2. Not applicable to DBCTL

The databases listed in the above table are divided into two categories: Full-function database types and Fast Path database types. DEDB and MSDB are the only two Fast Path database types. All other databases in the above table are considered full-function database types.

Related concepts:

Part 3, "Database types and functions," on page 113

Chapter 14, "Summary of IMS database types and functions," on page 115

The database record

A database consists of a series of database records, and a database record consists of a series of segments.

Another thing to understand is that a specific database can only contain one kind of database record. In the school database, for example, you can place as many school records as desired. You could not, however, create a different type of database record, such as the medical database record shown in the following figure, and put it in the school database.
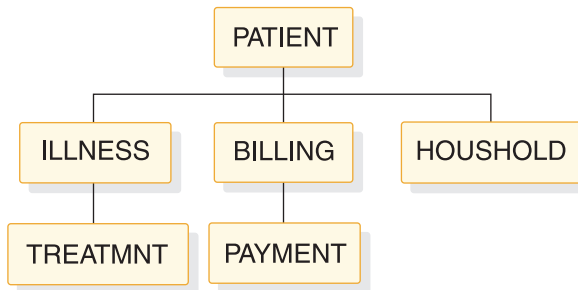
Figure 6. An example of a medical database record

The only other thing to understand is that a specific database record, when stored in the database, does not need to contain all the segment types you originally designed. To exist in a database, a database record need only contain an occurrence of the root segment. In the school database, all four of the records shown in the following figure can be stored.
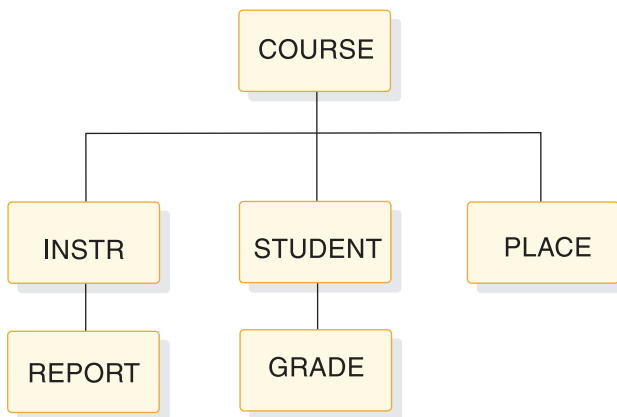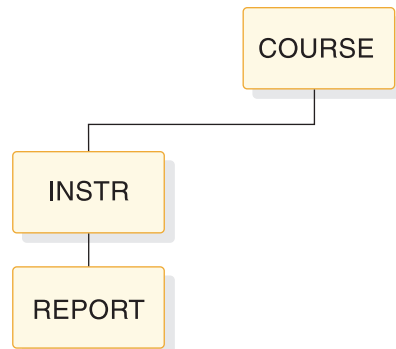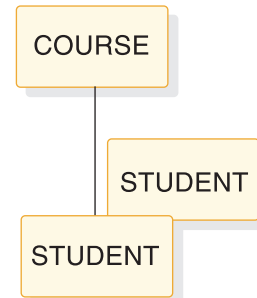
Figure 7. Example of records that can be stored in the school database

However, no segment can be stored unless its parent is also stored. For example, you could not store the records shown in the following figure.
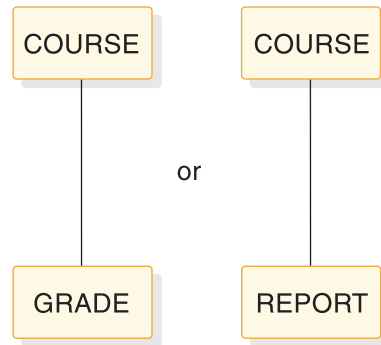
Figure 8. Records that cannot be stored in the school database

Occurrences of any of the segment types can later be added to or deleted from the database.

The segment

A database record consists of one or more segments, and the segment is the smallest piece of data IMS can store.

Here are some additional facts you need to know about segments:
* A database record can contain a maximum of 255 segment types. The space you allocate for the database limits the number of segment occurrences.
* You determine the length of a segment; however, a segment cannot be larger than the physical record length of the device on which it is stored.
* The length of segments is specified by segment type. A segment type can be either variable or fixed in length.

Segments consist of two parts (a prefix and the data), except when using a SHSAM or SHISAM database. In SHSAM and SHISAM databases, the segment consists of only the data. In a GSAM database, segments do not exist.

The following figure shows the format of a fixed-length segment.

Figure 9. Format of fixed-length segments

The following figure shows the format of a variable-length segment.
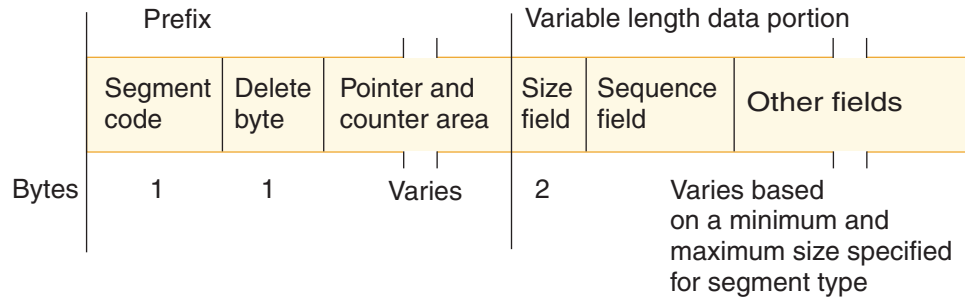
Figure 10. Format of variable-length segments

IMS uses the prefix portion of the segment to "manage" the segment. The prefix portion of a segment consists of: segment code, delete byte, and in some databases, a pointer and counter area. Application programs do not "see" the prefix portion of a segment. The data portion of a segment contains your data, arranged in one or more fields.

Related concepts:

"SHSAM, SHISAM, and GSAM databases" on page 140

"Main storage databases (MSDBs)" on page 218

"Data entry databases" on page 197

Segment code

IMS needs a way to identify each segment type stored in a database. It uses the segment code field for this purpose.

When loading a segment type, IMS assigns it a unique identifier (an integer from 1 to 255). IMS assigns numbers in ascending sequence, starting with the root segment type (number 1) and continuing through all dependent segment types in hierarchical sequence.

Delete byte

When an application program deletes a segment from a database, the space it occupies might or might not be immediately available to reuse.

Deletion of a segment is described in the discussions of the individual database types. For now, know that IMS uses this prefix byte to track the status of a deleted segment.

Related reference:

"Bits in the delete byte" on page 325

Pointer and counter area

The pointer and counter area exists in HDAM, PHDAM, HIDAM, and PHIDAM databases, and, in some special circumstances, HISAM databases.

The pointer and counter area can contain two types of information:

- Pointer information consists of one or more addresses of segments to which a segment points.
- Counter information is used when logical relationships, an optional function of IMS, are defined.

The length of the pointer and counter area depends on how many addresses a segment contains and whether logical relationships are used. These topics are covered in more detail later in this book.

The data portion

The data portion of a segment contains one or more data elements. The data is processed and unlike the prefix portion of the segment, seen by an application program.

The application program accesses segments in a database using the name of the segment type. If an application program needs to reference part of a segment, a field name can be defined to IMS for that part of the segment. Field names are used in segment search arguments (SSAs) to qualify calls. An application program can see data even if you do not define it as a field. But an application program cannot qualify an SSA on the data unless it is defined as a field.

The maximum number of fields that you can define for a segment type is 255. The maximum number of fields that can be defined for a database is 1000. Note that 1000 refers to types of fields in a database, not occurrences. The number of occurrences of fields in a database is limited only by the amount of storage you have defined for your database.

The three data portion field types

You can define three field types in the data portion of a segment: a sequence field, data fields, and for variable-length segments, a size field stating the length of the segment.

The first two field types contain your data, and an application program can use both to qualify its calls. However, the sequence field has some other uses besides that of containing your data.

You can use a sequence field, often referred to as a key, to keep occurrences of a segment type in key sequence under a given parent. For example, in the database record shown in the following figure, there are three segment occurrences of the STUDENT segment, and the STUDENT segment has three data elements.
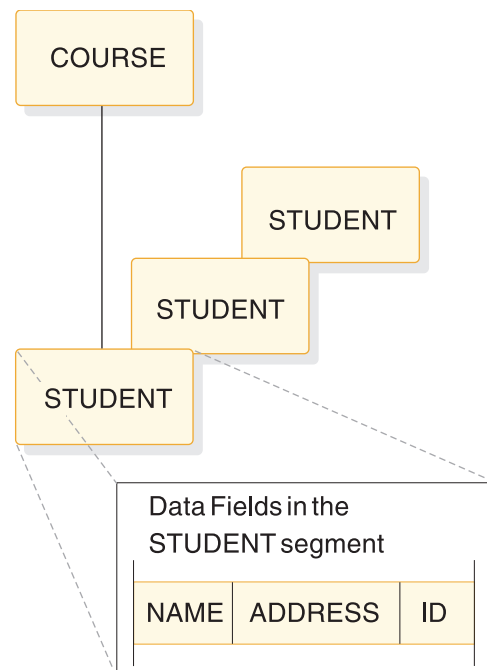
Figure 11. Three segment occurrences and three data elements of the STUDENT segment

Suppose you need the STUDENT segment, when stored in the database, to be in alphabetic order by student name. If you define a field on the NAME data as a *unique* sequence field, IMS stores STUDENT segment occurrences in alphabetical sequence. The following figure shows three occurrences of the STUDENT segment in alphabetical sequence.
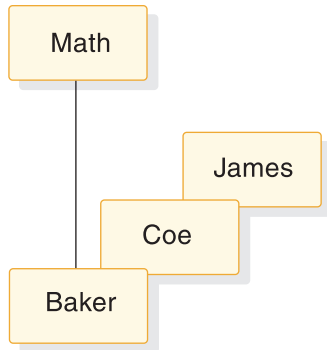
Figure 12. Example of STUDENT segments stored in alphabetic order

When you define a sequence field in a root segment of a HISAM, HDAM, PHDAM, HIDAM, or PHIDAM database, an application program can use it to access a specific root segment, and thus a specific database record. By using a sequence field, an application program does not need to search the database sequentially to find a specific database record, but can retrieve records sequentially (for HISAM, HIDAM, and PHIDAM databases).

You can also use a sequence field in other ways when using the IMS optional functions of logical relationships or secondary indexing. These other uses are discussed in detail later in this book.

The important things to know now about sequence fields are that:
- You do not always need to define a sequence field. This book describes cases where a sequence field is necessary.
- The sequence field value can be defined as unique or non-unique.
- The data or value in the sequence field is called the "key" of the segment.

Overview of optional database functions

IMS has several optional functions you can use for your database.

The functions include:

Logical relationships
> *Logical relationships* is a function you can use to let an application program access a logical database record. A logical database record can consist of segments from one or more physical database records. Physical database records can be stored in one or more databases. Thus, a logical database record lets an application program view a database structure that is different from the physical database structure.
>
> For example, if a logical data structure contains segments from two different physical databases, a segment can be accessed from two different paths:
> - A segment can be physically stored in the path where it is most frequently used and where the most urgent response time is required.

- A pointer containing the location of the segment can be physically stored in the alternate path needed by another application program.

Secondary indexing

Secondary indexing is a function you can use to access segments in a database in a sequence other than the one defined in the sequence field.

Variable-length segments

Variable-length segments is a function you can use to make the data portion of a segment type variable in length. Use variable-length segments when the size of the data portion of a segment type varies greatly from one segment occurrence to the next. With variable-length segments, you define the minimum and maximum length of a segment type. Defining both minimum and maximum length saves space in the database whenever a segment is shorter than the maximum length.

Field-level sensitivity

Field-level sensitivity is a function you can use to:

- Deny an application program access to selected fields in a segment for security purposes.
- Allow an application program to use a subset of the fields that make up a segment (and not process fields it does not use) or use fields in a segment in a different order. Use field-level sensitivity in this way to accommodate the differing needs of your application programs.

Segment edit/compression

Segment edit/compression is a function you can use with segments to:

- Encode or "scramble" segment data when it is on the device so only application programs with access to the segment receive the data in decoded form.
- Edit data so application programs can receive data in a format other than the one in which it is stored.
- Compress data when writing a segment to the device, so the Direct Access Storage Device (DASD) is better used.

A Data Capture exit routine

A Data Capture exit routine is used to capture segment data when an application program updates IMS databases with an insert, replace, or delete call. This is a synchronous activity that happens within the unit of work or application update. Captured data is used for data propagation to Db2 for z/OS databases. You can also use Data Capture exit routines to perform tasks other than data propagation.

Asynchronous Data Capture

Asynchronous Data Capture is a function you use to capture segment data when an application program updates IMS databases with an insert, replace, or delete call. This is an asynchronous activity that happens outside of the unit of work or application update. Captured data is used for data propagation to Db2 for z/OS databases asynchronously. You can also use Asynchronous Data Capture to perform tasks other than data propagation.

IMS DataPropagator allows you to propagate the changed data to or from IMS and Db2 for z/OS both synchronously and asynchronously.

Related reading: For more information on IMS DataPropagator see *IMS DataPropagator for z/OS: An Introduction*.

Multiple data set groups

Multiple data set groups is a function you can use to put some segments in a database record in data sets other than the primary data set. This can be done without destroying the hierarchical sequence of segments in a database record.

One reason to use multiple data set groups is to accommodate the differing needs of your applications. By using multiple data set groups, you can give an application program fast access to the segments in which it is interested. The application program simply bypasses the data sets containing unnecessary segments. Another reason for using multiple data set groups is to improve performance by, for example, separating high-use segments from low-use segments. You might also use multiple data set groups to save space by putting segment types whose size varies greatly from the average in a separate data set group.

Related concepts:

Chapter 20, "Optional database functions," on page 387

How databases are defined to IMS

If your IMSsystem manages runtime application control blocks by using the catalog, you can define databases to IMS by issuing industry standard SQL statements. However, if your IMS system uses ACB libraries, you must code a macro that generates a DBD.

A *DBD (database descriptor)* is a series of macro instructions that describes the organization and access methods for a database, the segments and fields in a database record, and the relationship between types of segments. The use of macros to generate DBDs is required only for IMS systems that use ACB libraries.

If you enable the IMS system to manage application control blocks by using the catalog, you do not need to code and generate a DBD. Instead, you can use the IMS Enterprise Suite Explorer for Development to model the attributes of your database and issue DDL statements to update the IMS catalog. IMS uses the catalog data to manage the database descriptions internally.

Certain databases, such as IMS partitioned hierarchic direct databases, known collectively as High Availability Large Databases (HALDB), require you to define additional database characteristics in the RECON data set.

If you have the IBM DB/DC (database/data communication) Data Dictionary, you can use it to define your database (except for DEDBs and MSDBs). The DB/DC Data Dictionary may contain all the information you need to produce a DBD.

How application programs view the database

You control how an application program views your database.

An application program might not need use of all the segments or fields in a database record. And an application program may not need access to specific segments for security or integrity purposes. An application program may not need to perform certain types of operations on some segments or fields. For example, an application program needs read access to a SALARY segment but not update access.

You control which segments and fields an application can view and which operations it can perform on a segment by defining a program view. You can define program views in one of two ways. If support for SQL DDL and the IMS management of ACBs is enabled in your IMS system, you can define program views by using DDL statements. You can also define program views by coding and generating a PSB (program specification block) with the PSB Generation utility.

A PSB is a series of macro instructions that describe an application program's access to segments in the database. A PSB consists of one or more program communication blocks (PCB), and each PCB describes the application program's ability to read and use the database. For example, an application program can have different views and uses of the same database. An application program can access several different databases and can have several PCBs in its PSB.

A program view that is defined by using DDL is functionally the same as a program view that is defined by using PSB macro instructions. When defining a new program view with DDL, it might be helpful to understand that the PSBGEN macro instruction that is used to define a PSB is generally equivalent to the CREATE PROGRAMVIEW DDL statement. Similarly, the PSB macro instructions that define PCBs are generally equivalent to the CREATE SCHEMA DDL statement.

If you have the IBM DB/DC Data Dictionary, you can use it to define an application program's access to the database. It can contain all the information needed to produce a program view.

Chapter 2. Standards, procedures, and naming conventions for IMS databases

Well planned standards and procedures and a good understanding of IMS conventions provide guidance to administrators, operators, and programmers improve the reliability and efficiency of your installation.

Standards and procedures for database systems

You must develop standards and procedures for your database system.

Adequate standards and procedures improve:
- The quality of application systems, because setting up and following standards and procedures gives you greater control over your entire application development process
- The productivity in application and database design, because guidelines for design decisions exist
- The productivity of application coding, because coding standards and procedures exist
- The communication between you and application developers, because you each have clearly defined responsibilities
- The reliability and recoverability in operations, because you have clear and well-understood operating procedures

You must set up and test procedures and standards for database design, application development, application programs' use of the database, application design, and for batch operation. These standards are guidelines that change when installation requirements change.

You can establish standard practices for the following aspects of database design:
- Database structure and segmentation

  Number of segments within a database

  Placement of segments

  Size of segments

  Use of variable-length segments

  When to use segment edit/compression

  When to use secondary data set groups

  Number of databases within an application

  When and how to use field-level sensitivity

  Database size
- Access methods

  When to use HISAM

  Choice of record size for HISAM

  HISAM organization using VSAM

  When to use GSAM

  Use of physical child/physical twin pointers

  Use of twin backward pointers

Use of child last pointers

HIDAM or PHIDAM index organization using VSAM

HIDAM or PHIDAM pointer options at the root level

Sequencing twin chains

Use of HD free space

When to use HDAM or PHDAM

Processing an HDAM or a PHDAM database sequentially

Use of the "byte limit count" for HDAM or PHDAM

Use of twin backward pointer for HDAM or PHDAM roots

Use of free space with HDAM or PHDAM

When to use DEDBs

Processing DEDBs sequentially

Use of DEDB parameters

Use of subset pointers

Use of multiple area data sets

- Secondary indexing

For sequential processing

On volatile segments

In HISAM databases

Use of unique secondary indexes

Use of sparse indexing

Processing of the secondary index as a separate database

- Logical relationships

Use of direct pointers versus symbolic pointers

Avoidance of long logical twin chains

Sequencing of the logical twin chain

Placement of the real logical child segment

You can also establish standards for the ways in which application programs use the database, for example:

- Requiring update and read functions to be in separate programs
- How many transaction types to allow per application program
- When applications are to issue a deliberate abnormal termination and the range of abend codes that is permitted to applications
- Whether application programs are permitted to issue messages to the master terminal
- The method of referencing data in the IOAREA, and referencing IMS variables (such as PCBs and SSAs)
- Use of predefined structures, such as PCB masks, SSAs, or database segment formats, by applications
- Use of GU calls to the message queue
- Re-usability of MPP and BMP programs
- Use of qualified calls and SSAs
- Use of path calls
- Use of the CHANGE call
- Use of the system calls: PURG, LOG, STAT, SNAP, GCMD, and CMD

Establish procedures to govern the following aspects of application design:
- The interaction between you and the application designer
- Use of the dictionary or COPY or STRUCTURE libraries for data elements and structures
- The requirement of design reviews and inspections

For operations, consider developing:
- Procedures to limit access to computer facilities
- A control point, to ensure that:
  - Jobs contain complete and proper submittal documentation
  - Jobs are executed successfully on schedule
  - Correct input and output volumes are used, and output is properly distributed
  - Test programs are executed only in accordance with a defined test plan
  - An incident report is maintained to ensure that all problems are recorded and reported to the responsible parties
- Normal operating procedures, including operations schedules, procedures for cold start, warm start, and shutdown, and scheduling and execution of batch programs.
- Procedures for emergency situations. During an emergency, the environment is one of stress. Documented procedures provide step-by-step guidance to resolve such situations. Include procedures for emergency restart, database backout, database recovery, log recovery, and batch program restart.
- A master terminal operator's guide for the installation. This guide should be supplemented by *IMS Version 15 Operations and Automation*.
- A master operations log. This log could contain a record of system availability, time and type of failure, cause of the failure, recovery steps taken, and type of system termination if normal.
- A system maintenance log. This log could contain a record of all release and modification levels, release dependencies, program temporary fixes (PTFs) applied, the status of APARs and date submitted, and bypass solutions.

General naming conventions for IMS databases

Naming conventions help users identify and manage the many resources in an IMS system. Some naming conventions are defined by IMS, while many others can be defined by you.

General rules for establishing naming conventions

Good naming conventions are mandatory in a data processing project, especially in an environment with multiple applications.

A good naming convention includes the following general rules:
- Each name must be unique. If names are not unique, unpredictable errors can occur.
- Each name must be meaningful and identify to all personnel the type of resource that the named element is.

The following table provides an example of basic naming conventions. These conventions are only an example, and you can establish your own naming conventions.

Table 2. Example of basic naming conventions.

Resource type	Convention
SYSTEM	S as first letter
JOB	J as first letter
PROGRAM	P as first letter if this is an IMS program (to match PSB) G as first letter otherwise
MODULE	M as first letter
COPY	C as first letter for a member that contains the segment structure A as first letter for a member that contains all the SSAs for the segment Other members must be the same as the segment name
TRANSACTION	T as first letter
PSB	P as first letter
PCB	Same name as PSB **Note:** The PCB occurrence number indicates the position of the PCB in the PSB
DATABASE	D as first letter with the subsequent characters identifying the type of database and its relationship to other databases. For example, D*taaann*, in which the characters *taaann* indicate the following:

	Character	Meaning
	t	Database type. The database can be one of the following types:
		P Physical
		L Logical
		X Primary index
		Y Secondary index
	aaa	A unique database identifier common to all logical and index databases based on the same physical database
	nn	A unique identifier, if there are multiple logical or secondary index databases

Resource type	Convention
SEGMENT	S, R, or O as first letter with the subsequent characters identifying the type of segment and its relationship to its database. An R identifies 'segments' that are non-DL/I file record definitions. An O identifies any other data areas, for example, terminal I/O areas, control blocks, report lines, and so on. For example, S*aaabbbb*, in which the characters *aaabbbb* indicate the following:

	Character	Meaning
	aaa	A unique database identifier; same as the physical database in which the segment occurs **Note:** Concatenated segments should have an aaa value corresponding to the aaa of the logical child segment.
	bbbb	An identifier for the user name

Resource type	Convention
ELEMENT	E as first letter

Naming conventions for HALDB partitions, ddnames, and data sets

HALDB naming conventions for partitions, ddnames, and data set names simplify the management of numerous partitions and data sets in HALDB PHDAM, PHIDAM, and PSINDEX databases.

Related concepts:

"Data set naming conventions for HALDB Online Reorganization" on page 677

Related tasks:

"Allocating logically related database data sets" on page 874

"Allocating the indexed database data sets" on page 866

"Allocating database data sets" on page 854

Naming convention for HALDB partitions

You assign names to each partition. Partition names are 1–7 bytes in length.

These names must be unique among the database names, partition names, and Fast Path area names that are registered in the RECON data set. You can use partition names to describe the data in the partition, but choose such names carefully. If you add or delete partitions or modify their boundaries, data might move from one partition to another. This movement can make the assignment of meaningful names difficult. You cannot change the name of an existing partition without deleting it and redefining it as a new partition.

Naming convention for HALDB data definition names (ddnames)

IMS defines HALDB data definition names (ddnames) by appending a 1-byte suffix to the partition name. The suffix indicates the type of data set and, if you use multiple data set groups, differentiates the data sets within the group.

The following table shows the HALDB data set types and the corresponding ddname suffixes.

Table 3. Suffixes for HALDB ddnames by data set type

Data set type	Ddname suffix	Additional suffixes if HALDB Online Reorganization is used
Database data set	A–J	M–V
Primary index (PHIDAM only)	X	Y
Indirect list data set (PHDAM and PHIDAM only)	L	L (the suffix for the ILDS does not change)

If you use multiple data set groups, the A through J suffixes are the values that you would specify on the DSGROUP parameter in either a SEGM statement or, if support for DDL is enabled, a CREATE or ALTER TABLE DDL statement. The letter A identifies the first database data set (DBDS), the letter B identifies the second, and so forth, up to the letter J. If you do not use multiple data set groups, you do not specify the DSGROUP parameter and the ddname for the single data set that contains the record segments has the suffix A.

The suffixes M–V and Y are created automatically for the integrated HALDB Online Reorganization function of IMS. You do not need to specify them in the DBD. If you have never used the HALDB Online Reorganization function to reorganize a given partition, the suffixes M–V and Y are not used in that partition.

In PSINDEX databases, each partition contains only one data set. The suffix A is used for the ddname that corresponds to that data set.

For example, a PHIDAM database partition named PART1 would have ddnames of PART1A for its first DBDS, PART1B for the second DBDS, up to PART1J for the

tenth DBDS. The indirect list data set (ILDS) and the primary index of partition PART1 would have ddnames of PART1L and PART1X, respectively. And a PSINDEX database partition named PARSI would have a ddname of PARSIA for its data set.

When reorganizing a partition, the integrated HALDB Online Reorganization function of IMS uses an additional data set for each data set that is active prior to starting the online reorganization process. For example, a ddname of PART1M is created to correspond to the active data set PART1A. A PART1N is created for PART1B, and so on, up to PART1V for PART1J, if it exists.

The ddnames must be unique among the database names, partition names, and Fast Path area names that are registered in the RECON data set.

Naming convention for HALDB data set names

You define a part of HALDB data set names and IMS creates the rest.

When you define a partition, you define a data set name prefix of up to 37 characters for the partition. A data set name prefix cannot be a duplicate of a data set name prefix in any other HALDB database, but it can be duplicated within a single HALDB database. Because partition IDs are unique, the suffix that IMS appends to each data set name prefix makes the data set names unique for the different partitions within a HALDB database. There is no required correlation between the partition name and the names of its data sets.

To create the lowest-level qualifier, IMS appends a 6-character suffix to the prefix to form the data set name. The first character of the IMS-supplied suffix is an alphabetic character: either A–J, L, and X, or M–V, L, and Y. The 6-character suffix is separated from the preceding data set name qualifiers by a period.

The first character of the data set name suffix matches the character that is used as the suffix in the ddname. The remaining five digits of the suffix represent the partition ID number, which is assigned by DBRC and you cannot change. For example:

- A suffix of A00001 indicates the first or only DBDS in a partition with partition ID 1
- A suffix of J00004 indicates the tenth DBDS in a partition with partition ID 4
- A suffix of L00007 indicates the ILDS in a partition with partition ID 7
- A suffix of X00011 indicates the primary index in a PHIDAM partition with partition ID 11

Chapter 3. Review process for database development

One of the best ways to make sure a good database design is developed and effectively implemented is to review the design at various stages in its development.

The types of reviews are that are typically conducted during development of a database system are described in the following topics.

Design reviews 1, 2, 3, and 4

Code inspections 1 and 2

Security inspection

Post-implementation review

The design review

Design reviews ensure that the functions being developed are adequate, the performance is acceptable, the installation standards met, and the project is understood and under control.

Hold reviews during development of the initial database system and, afterward, whenever a program or set of programs is being developed to run against it.

Role of the database administrator in design reviews

The role of a database administrator in the review process is to ensure that a good database design is developed and then effectively implemented. The role is ongoing and provides a supporting framework for the other database administration tasks.

The role of database administration in the review process is an important one. Typically, a member of the database administration staff, someone not associated with the specific system being developed, moderates the reviews. The moderator does more than just conduct the meeting. The moderator also looks to see what impact development of this system has on existing or future systems. You, the database administrator responsible for developing the system, need to participate in all reviews.

General information about reviews

During system development, development groups typically hold a series of reviews that are common to most development projects.

For purposes of simplicity, "system" describes the object under review. In actuality, the "system" could be a program, set of programs, or an entire database system. The number of reviews, who attends them, and their specific role in the review will differ slightly from one installation to the next. What you need to understand is the importance of the reviews and the tasks performed at them. Here is some general information about reviews:

- People attending all reviews (in addition to database administrators) include a review team and the system designer. The review team generally has no responsibility for developing the system. The review team consists of a small

group of people whose purpose is to ensure continuity and objectivity from one review to the next. The system designer writes the initial functional specifications.

- At the end of each review, make a list of issues raised during the review. These issues are generally change requirements. Assign each issue to a specific person for resolution, and set a target date for resolution. If certain issues require major changes to the system, schedule other reviews until you resolve all major issues.
- If you have a data dictionary, update it at the end of each review to reflect any decisions that you made. The dictionary is an important aid in keeping information current and available especially during the first four reviews when you make design decisions.

Design review 1

The purpose of design review 1 is to ensure that all user requirements have been identified and that design assumptions are consistent with objectives.

The first design review takes place after initial functional specifications for the system are complete. No detailed design for the system is or should be available at this point. The review of the specifications will determine whether the project is ready to proceed to a more detailed design. When design review 1 concludes successfully, its output is an approved set of initial *functional* specifications.

People who attend design review 1, in addition to the regular attendees, include someone from the organization that developed the requirement and anyone participating in the development of detailed design. You are at the review primarily for information. You also look at:

> The relationship between data elements
>
> Whether any of the needed data already exists

Design review 2

Your role in design review 2 is primarily to gather information.

The second design review takes place after final *functional* specifications for the system are complete. This means the overall logic for each program in the system is defined, as well as the interface and interactions between programs. Audit and security requirements are defined at this point, along with most data requirements. When design review 2 is successfully concluded, its output is an approved set of final functional specifications.

Everyone who attended design review 1 should attend design review 2. People from test and maintenance groups attend as observers to begin getting information for test case design and maintenance. Those concerned with auditing and security can also attend.

Your role in this review is still primarily to gather information. You also look at:
- Whether the specifications meet user requirements
- Whether the relationship between data items is correct
- Whether any of the required data already exists
- Whether audit and security requirements are consistent with user requirements
- Whether audit and security requirements can be implemented

Design review 3

Your role in design review 3 is to ensure that the flow of transactions is consistent with the database design you are creating.

The third design review takes place after initial *logic* specifications for the system are complete. At this point, high level pseudo code or flowcharts are complete. These can only be considered complete when major decision points in the logic are defined, calls or references to external data and modules are defined, and the general logic flow is known. All modules and external interfaces are defined at this point, definition of data requirements is complete, and database and data files are designed. Initial test and recovery plans are available; however, no code has been written. When design review 3 concludes successfully, its output is an approved set of initial logic specifications.

Everyone who attended design review 2 should attend design review 3. If the project is large, those developing detailed design need only be present during the review of their portion of the project.

It is possible now that logic specifications are available.

At this point in the design review process, you are designing hierarchies and starting to design the database.

Related concepts:

Chapter 21, "Analyzing data requirements," on page 421

Chapter 15, "Full-function database types," on page 117

Chapter 20, "Optional database functions," on page 387

Chapter 22, "Designing full-function databases," on page 431

Design review 4

The primary objective of design review 4 is to make sure that system performance will be acceptable.

The fourth design review takes place after design review 3 is completed and all interested parties are satisfied that system design is essentially complete. No special document is examined at this review, although final functional specifications and either initial or final logic specifications are available.

At this point in the development process, sufficient flexibility exists to make necessary adjustments to the design, since no code exists but detailed design is complete. Although some design changes undoubtedly occur once coding is begun, these changes should not impact the entire system. Although no code exists at this point, you can and should run tests to check that the database you have designed will produce the results you expect.

When design review 4 concludes successfully, database design is considered complete.

The people who attend all design reviews (moderator, review team, database administrator, and system designer) should attend design review 4. Others attend only as specific detail is required.

At this point in the review process, you are almost finished with the database administration tasks along with designing and testing your database.

Related concepts:

Code inspection 1

The objective of code inspection 1 is to ensure that the correctly developed logic interprets the functional specification. Code inspection 1 also provides an opportunity to review the logic flow for any performance implications or problems.

The first code inspection takes place after final logic specifications for the system are complete.

At this point, no code is written but the final functional specifications have been interpreted. Both pseudo code and flowcharts have a statement or logic box for every 5 to 25 lines of assembler language code, 5 to 15 lines of COBOL code, or 5 to 15 lines of PL/I code that needs writing. In addition, module prologues are written, and entry and exit logic along with all data areas are defined.

When code inspection 1 successfully concludes, its output is an approved set of final logic specifications.

Who attends code inspection 1

Code inspection 1 is attended primarily by those doing the coding. People who attend all design reviews (moderator, review team, database administrator, and system designer) also attend the code inspection 1. Testing people present the test cases that will be used to validate the code, while maintenance people are there to learn and evaluate maintainability of the database.

Your role in this review is now a less active one than it has been. You are there to ensure that everyone adheres to the use of data and access sequences defined in the previous reviews.

At this point in the review process, you are starting to implement database design, to develop test databases, and to load databases.

Related concepts:

Code inspection 2

The objective of the second code inspection is to make sure module logic matches pseudo code or flowcharts. Interface and register conventions along with the general quality of the code are checked. Documentation and maintainability of the code are evaluated.

The code inspection 2 takes place after coding is complete and before testing by the test organization begins.

Everyone who attended code inspection 1 should attend code inspection 2.

Your role in this review is the same as your role in code inspection 1.

At this point in the review process, you are almost finished with the database administration tasks of developing a test database, implementing the database design, and loading the database.

During your testing of the database, you should run the DB monitor to make sure your database still meets the performance expectations you have established.

Related concepts:

Chapter 29, "Monitoring databases," on page 635

Security inspections

The purpose of a security inspection review is to look for any code that violates the security of system interfaces, secured databases, tables, or other high-risk items.

The security inspection is optional but highly recommended if security is a significant concern. Security inspections can take place at any appropriate point in the system development process. Define security strategy early, and check its implementation during design reviews. This particular security inspection takes place after all unit and integration testing is complete.

People who attend the security inspection review include the moderator, system designer, designated security officer, and database administrator. Because the database administrator is responsible for implementing and monitoring the security of the database, you might, in fact, be the designated security officer. If security is a significant concern, you might prefer that the review team not attend this inspection.

During this and other security inspection, you are involved in the database administration task of establishing security.

Related concepts:

Chapter 4, "Database security," on page 35

Post-implementation reviews

A post-implementation review is typically held about six months after the database system is running. Its objective is to make sure the system is meeting user requirements.

Recommendation: Conduct a post-implementation review.

Everyone who has been involved in design and implementation of the database system should attend the post-implementation review. If the system is not meeting user requirements, the output of this review should be a plan to correct design or performance problems to meet user requirements.

Chapter 4. Database security

Database security has two aspects: user verification and user authority.

User verification refers to how you establish that the person using an online database is in fact the person you have authorized.

User authority refers to how you control what users can see and what the users can do with what they see after you verify the user's identity.

These topics deal primarily with how you can control a user's view of data and the user's actions with respect to the data.

Related reading: If you use CICS, see *CICS Transaction Server for z/OS RACF Security Guide* for information on establishing security.

Related concepts:

"Security inspections" on page 33

Restricting the scope of data access

You can restrict a user's access to (and even knowledge of) elements of a database by limiting the view of a database that you define for an application program.

A schema or a PCB defines a program's (and therefore the user's) view of the database. You can think of a schema as a "mask" that can be placed on top of the defined data structure of a database to hide certain parts of the data structure.

In "Restricting processing authority," the top of the first figure shows the hierarchical structure for a PAYROLL database as seen by you in the definition of the database. For certain applications, it is not necessary (nor desirable) to access the SALARY segment. By omitting sensitivity to the SALARY segment from the DDL SCHEMA statement or PCB macro statement that defines the view of the database for the application, you can make it seem that this segment simply does not exist. By doing this, you have denied unauthorized users access to the segment, and you have denied users knowledge of its very existence.

For this method to be successful, the segment being masked off must not be in the search path of an accessed segment. If it is, then the application is made aware of at least the key of the segment to be "hidden."

With field-level sensitivity, you can achieve the same masking effect at the *field* level. If SALARY and NAME were in the same segment, you could still restrict access to the SALARY field without denying access to other fields in the segment.

Restricting processing authority

After you have controlled the scope of data a user has access to, you can also control authority within that scope.

Controlling authority allows you to decide what processing actions against the data a given user is permitted. For example, you could give some application programs authority only to read segments in a database, while you give others authority to update or delete segments.

Control processing authority with the PROCOPT parameter

You can control the processing actions of a user through the PROCOPT parameter when you define a program view (PSB). The PROCOPT parameter tells IMS what actions you permit against the database. A program can do what is declared in the PROCOPT.

If your IMS system is enabled to support DDL, the PROCOPT parameter can be specified on the CREATE PROGRAMVIEW statement. If you are defining a program view by using macro instructions for the PSB Generation utility, you can specify the PROCOPT parameter on SENSEG and PCB statements.

Limit program sensitivity to only required segments

In addition to restricting access and authority, you can limit the segments to which an application program is sensitive. The number of sensitive segments and the processing option specified can have an impact on data availability. To achieve maximum data availability, the PSB should be sensitive only to the segments required and the processing option should be as restrictive as possible.

For example, the database definition in the following macro instructions for the DBD Generation utility describes a payroll database that stores the name, address, position, and salary of employees. The hierarchical structure of the database record is shown in figure following the code.

Figure 13. Example database definition for a payroll database

```
DBD   NAME=PAYROLL,...
DATASET ...
SEGM  NAME=NAME,PARENT=0...
FIELD NAME=
SEGM  NAME=ADDRESS,PARENT=NAME,...
FIELD NAME=
SEGM  NAME=POSITION,PARENT=NAME,...
FIELD NAME=
SEGM  NAME=SALARY,PARENT=NAME,...
FIELD NAME=
  .
  .
  .
```
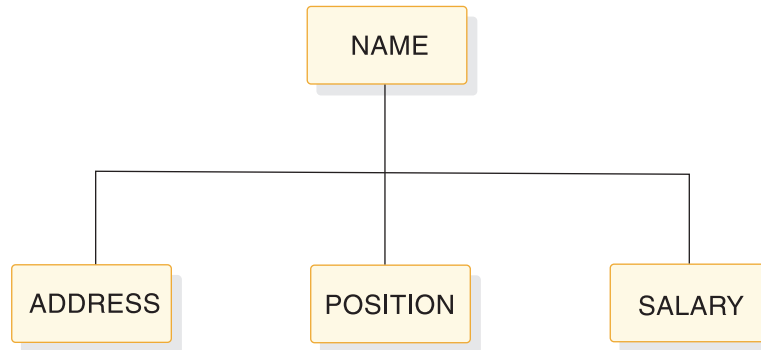
Figure 14. Payroll database record without a mask

If an application needs access to the name, address, and position of employees, but not the salary, you can use the SENSEG statement in the DB PCB macro instructions for the PSB Generation utility to make the application sensitive to only the name, address, and position segments. The SENSEG statements on the DB PCB creates a mask over the database record hiding segments from application. The following code shows the DB PCB that masks the SALARY segment of the payroll database from the application.

Figure 15. Example PCB for a payroll database

```
PCB TYPE=DB.DBDNAME=PAYROLL,...
SENSEG NAME=NAME,PARENT=0,...
SENSEG NAME=ADDRESS,PARENT=NAME,...
SENSEG NAME=POSITION,PARENT=NAME,...
.
.
.
```

If you are using DDL, you can achieve the same result as the preceding macro instruction statements by using the CREATE SCHEMA statement:

```
CREATE SCHEMA pcb01 USING payroll AS pcb01 (
   CREATE SENSEGVIEW name,
   CREATE SENSEGVIEW address,
   CREATE SENSEGVIEW position,

.
.
.
)
```

Figure 16. Example DDL program view schema for a payroll database

The following figure shows what the payroll database record looks like to the application based on the preceding program view definitions. It looks just like the database record in the preceding figure except that the SALARY segment is hidden.
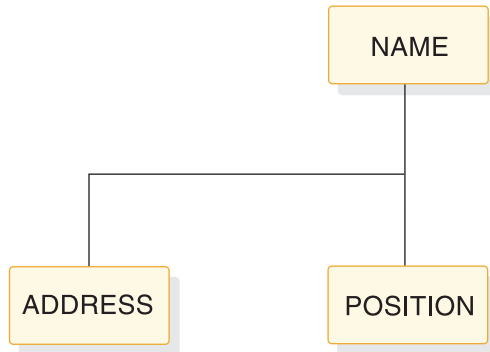
Figure 17. Payroll database record with SALARY segment masked

Restricting access by non-IMS programs

One potential security exposure is from people attempting to access IMS data sets with non-IMS programs. Two methods of protecting against this exposure are data set password protection and database encryption.

Protecting data with VSAM passwords

You can take advantage of VSAM password protection to prevent non-IMS programs from reading VSAM data sets on which you have your IMS databases.

To protect data with VSAM passwords, enable password protection for your VSAM data sets by coding either PASSWD=YES on the DBD statement or, if support for DDL is enabled, PASSWDYES in the CREATE or ALTER DATABASE DDL statement. IMS then passes the DBD name as the password. If you do not enable password protection for your VSAM data sets, the console operator is prompted to provide a password to VSAM each time the data set is opened.

This method is only useful in the batch environment, and VSAM password checking is bypassed entirely in the online system. (If you have RACF® installed, you can use it to protect VSAM data sets.)

Related reference:

↪ DBD statements (System Utilities)

↪ CREATE DATABASE (Application Programming APIs)

↪ ALTER DATABASE (Application Programming APIs)

Encrypting your database

You can encrypt DL/I databases to help prevent non-IMS programs from reading.

You can encrypt DL/I segments using your own encryption routine, entered at the segment edit/compression exit. Before segments are written on the database, IMS passes control to your routine, which encrypts them. Then, each time they are retrieved, they are decrypted by your routine before presentation to the application program.

Do not change the key or the location of the key field in index databases or in root segments of HISAM data bases.

Related concepts:
"Segment Edit/Compression exit routine" on page 390

Security for IMS DDL

There are three different security considerations; security for issuing the DDL, security for activating the DDL, for example issuing the IMPORT command, and security for the resources create, altered, and deleted by DDL.

Security for issuing the DDL

You can set up security around who can issue the DDL and make changes to resources. For example, you might not want Application Developers to make DB changes, only PSB changes.

Security for activating the DDL

You can set up less security around who can issue the DDL to make changes but more security around who activates them. For example, you might have Application Developers make the changes but only System Programmers or DBAs activate the changes after analyzing the impact of the changes, meaning, reorganizing or application outages.

Security for resources

You might want to consider what the DDL does under the covers as part of that change. For example, DDL for DROP DATABASE will eventually try to delete data sets, however, you might have RACF restricting those data sets. In this case, ensure that IMS is authorized so that it can actually perform the task.

When setting up security in a DDL flow, you can:

In IMS Enterprise Suite Explorer for Development, you can restrict who can submit DDL to IMS.
For more information, see Submitting DBD field changes to IMS with IMS 14.

In IMS, you can restrict who can access a given PSB.
This uses existing security mechanisms for PSBs. You can secure the PSB named DFSCP001 to restrict who can issue DDL. For more information, see Security for ODBA application programs (System Administration).

In IMS, you can restrict who can issue the IMPORT DEFN command for IMS Explorer security.
IMPORT is a type-2 command; the command security is the existing type-2 command security in OM. The security is based on IMPORT DEFN, therefore, you will need to write OM security exit to have granular security based on SOURCE(CATALOG). For more information, see CSL OM command security (System Administration).

Using a dictionary to help establish security

A dictionary, such as the IBM DB/DC Data Dictionary, monitors relationships among entities in your computing environment (such as, which programs use which data elements), making it an ideal tool to administer security.

You can use the dictionary to define your authorization matrixes. Through the extensibility feature, you can define terminals, programs, users, data, and their relationships to each other. In this way, you can produce reports that show: dangerous trends, who uses what from which terminal, and which user gets what data. For each user, the dictionary could be used to list the following information:

- Programs that can be used
- Types of transactions that can be entered
- Data sets that can be read
- Data sets that can be modified
- Categories of data within a data set that can be read
- Categories of data that can be modified

Part 2. IMS catalog

The following topics describe the purpose and content of the IMS catalog database and administrative tasks to maintain the IMS catalog.

Chapter 5. Overview of the IMS catalog

The IMS catalog contains trusted metadata and definitions of the IMS databases and application program views that are defined to IMS.

If the IMS management of ACBs is enabled, the IMS catalog also determines the active databases and program views (PSBs) in the IMS system, because ACB libraries are not used. When IMS uses ACB libraries, the ACB library determines which databases and program views are active, and you must ensure that the IMS catalog is always in synch with the ACB library.

The IMS catalog is itself a HALDB PHIDAM database. Each database and application program view that is defined to IMS is stored in a separate record in the IMS catalog. In each record, the root header segment identifies the type of resource that it contains: either a database definition (DBD) or a program view (PSB).

The header segment is followed in the hierarchy by a DBD or a PSB segment. The DBD or PSB segment and its dependent segments store the definition and metadata of the database or program view.

Subsequent definitions of a database or program view are stored in the same record as the previous definitions of the database or program view by inserting more instances of the DBD or PSB segment and its dependents. The different instances of a database or program view are differentiated by a time stamp.

The database and program view instances in a record can include an active instance that is being used by the online IMS system, previously defined instances, and draft instances that were never activated.

If database versioning is used, the DBD segments for a previous version of a database must be retained in the DBD record in the IMS catalog to provide application programs access to the previous version of the database.

Depending on whether you enable the IMS management of application control blocks (ACBs), you have different options for how you define databases and program views, add them to the IMS catalog, and activate them in the IMS system.

When IMS manages the ACBs, you can define databases and program views either by using SQL data definition language (DDL) statements or by using the input macros of the DBD Generation utility and PSB Generation utility.

When you use DDL statements, IMS can add the database and program view definitions to the IMS catalog, build the required runtime control blocks, and, in some cases, load them into the online IMS system automatically.

When you use the DBD and PSB Generation utilities to define databases and program views in an IMS system that manages ACBs, after you run the utilities, you must also run the ACB Generation and Populate utility (DFS3UACB) or equivalent utilities to build the ACBs, update the IMS catalog, and load the ACBs into the IMS system.

In an IMS system that manages ACBs, the IMS catalog completely replaces DBD, PSB, and ACB libraries as the component that determines which database and program view definitions are used by the online IMS system and by batch application programs.

When the IMS management of ACBs is disabled, you cannot use DDL to define databases and program views. Instead, you must define them by using the DBD and PSB Generation utilities, you must generate members into an ACB library, and you must use the online change process to activate the ACB library. You must also make sure that the IMS catalog remains in sync with the active ACB libraries.

Important: If the IMS catalog is not enabled in the DFSDFxxx IMS PROCLIB member for the IMS system, no catalog information is created during ACBGEN. Also, the IMS catalog must be enabled to enable the IMS management of ACBs.

Data is stored in the IMS catalog database as hexadecimal (type X) for numeric values, or Cp1047 EBCDIC character data. In some cases, values are truncated to save space. If a field contains blanks (for character data), that indicates that the field does not apply to that database or program specification block. Individual field definitions might indicate other meanings for a blank field.

IMS provides a skeletal COBOL copybook and skeletal PL/I program for accessing the catalog database. The COBOL copybook is called DFS3DCBL and can be found in IMS.ADFSISRC. The PL/I sample application is named DFS3DPL1 and can be found in the IMS sample library, IMS.ADFSSMPL.

When the catalog is enabled in the DFSDF*xxx* member of the IMS.PROCLIB data set, IMS automatically adds a PCB list for the IMS catalog to each user PSB at run time.

IMS provides two DBDs for the IMS catalog, one for the main IMS catalog database and the other for the secondary index. IMS provides several PSBs for the IMS catalog for different purposes and application program types. The DBDs and PSBs for the IMS catalog are defined as resident.

The catalog database segment types are grouped into four different data set groups (A - D) based on how frequently that segment type is accessed in database queries. The root segment type (HEADER) and the DBD and PSB segment types are in data set group A. The least frequently accessed segment types, such as user remarks, are grouped in data set D.

As a HALDB PHIDAM database, the IMS catalog can be queried with standard DL/I processing, with DL/I processing through the Universal DL/I driver, and with SQL through the Universal JDBC driver. IMS catalog database records cannot be updated, replaced, inserted, or deleted except with the provided utilities.

Related concepts:

➡ IMS catalog definition and tailoring (System Definition)

Related tasks:

➡ Installing the IMS catalog DBDs and PSBs (System Definition)

Related reference:

➡ IMS catalog data set groups (System Definition)

➡ IMS catalog utilities (System Utilities)

➡ The IMS Catalog Redpaper

Chapter 6. Backup and recovery of the IMS catalog

The IMS catalog is a HALDB database, so you can use standard HALDB backup and recovery procedures to back up and recover an IMS catalog, but you must make sure that the records in the recovered IMS catalog match that application control blocks (ACBs) that are active in the IMS system.

For certain IMS functions, such as the IMS management of ACBs or database versioning, the ability to recover the IMS catalog from back up copies is required. If these functions are not used, you can recover the IMS catalog by using one of the population utilities to rebuild it from ACB, DBD, and PSB libraries.

One of the functions that requires backup copies for recovery is the IMS management of ACBs. When IMS manages ACBs, the IMS catalog is the only repository for ACBs, DBDs, and PSBs in the IMS system and contains the only record of which of those resources are active.

Another function that requires backup copies for recovery is database versioning. When database versioning is used in IMS systems that use an ACB library, the DBDs for previous versions of a database exist only in the IMS catalog and cannot be repopulated from the ACB library. Only the version of the database definition (DBD) that is designated as active can be repopulated from the active ACB library.

IMS catalog data sets

The IMS catalog is a HALDB partitioned HIDAM (PHIDAM) database with a primary index data set, an indirect list data set (ILDS). The IMS catalog includes a HALDB partitioned secondary index (PSINDEX) database.

If the IMS management of ACBs is enabled, the IMS catalog also uses one or more IMS directory data sets. If any database or program view changes are pending activation, the IMS directory data sets might also include a staging data set.

Coordinating IMS catalog recovery with the active ACBs

In IMS systems that use IMS-managed ACBs, when you recover the IMS catalog from backup image copies, you must ensure that the DBD and PSB resource instances that are flagged as active in the recovered IMS catalog match the active ACBs in the IMS directory data set.

One way to ensure that the active instances in the recovered IMS catalog match the active ACBs in the IMS directory data set is to rebuild the IMS directory by using the IMS Catalog Directory Recovery utility (DFS3RU00).

However, because rebuilding IMS directory data sets can take a long time, consider creating backup copies of the IMS directory data sets that correspond to the image copies you create of the IMS catalog.

In IMS systems that use ACB libraries, when you recover the IMS catalog from backup image copies, you must ensure that the recovered IMS catalog contains the DBD and PSB instances that match the ACBs in the active ACB library. The time stamps of the records in the IMS catalog must match the time stamps of the corresponding ACB members in the active ACB library.

DBRC, the IMS recovery utilities, and the IMS catalog

All standard IMS utilities can run on the catalog data sets, including the image copy and database recovery utilities that are provided with IMS.

When the IMS catalog is managed by DBRC, the utilities that are provided with IMS create recovery information in the log data sets when the IMS catalog is updated. DBRC manages the logs, image copies, and JCL required for recovery of the IMS catalog. You can perform a full database recovery or a point-in-time recovery for the IMS catalog partitions.

When DBRC is not used with the IMS catalog, the IMS catalog can be recovered by using the standard backup and recovery processes that are used for other HALDB databases. However, you must have processes in place to manage the logs, image copies, JCL, and so on.

Recovery by repopulation

If your installation does not use either the IMS management of ACBs or database versioning, as an alternative to recovering the IMS catalog from image copies and log records, you can re-create the IMS catalog from your ACB, DBD, and PSB libraries by running either the IMS Catalog Populate utility (DFS3PU00) or the ACB Generation and Catalog Populate utility (DFS3UACB) to reload the IMS catalog.

Re-creating the IMS catalog from ACB, DBD, and PSB libraries by using one of the population utilities restores only the record segments in the IMS catalog that are in the libraries. Use this method only if you do not need record segments for past instances of your DBDs and PSBS or the historical metadata that those segment instances contain.

An initial load of the catalog by either of the population utilities does not create any database recovery information in the log data sets. Create an image copy immediately after the IMS catalog is loaded to ensure that the image copy is consistent with the active ACB library.

Recovering the IMS catalog directory data sets

When the IMS management of ACBs is enabled, you can use the IMS Catalog Directory Recovery utility (DFS3RU00) to recover the IMS directory data sets from database image copies after the recovery of IMS catalog. The IMS Catalog Directory Recovery utility rebuilds the IMS directory data sets and recovers the ACBs of IMS databases and programs to the directory by reading active resources from the IMS catalog.

Related concepts:

"Making database backup copies" on page 595

"Recovery of databases" on page 606

Chapter 27, "Database backup and recovery," on page 589

Related reference:

➡ IMS catalog utilities (System Utilities)

Backup methods for the IMS catalog

Because the IMS catalog is a HALDB PHIDAM database, you can use the IMS image copy utilities to back up the partition data sets of the main database and the secondary index.

Like the recovery of other HALDB databases, the recovery of the IMS catalog does not use image copies of the primary index data set or the indirect list data set (ILDS). Instead, these data sets are recovered by rebuilding them by using the HALDB Index/ILDS Rebuild utility (DFSPREC0).

If the IMS management of ACBs is enabled, the IMS catalog also uses additional system data sets to manage the ACBs:

- One more IMS directory data sets that contain the ACBs that are active in the IMS system
- A bootstrap data set
- A staging data set that stores new or changed ACBs that are not yet active in the IMS system.

The names of these data sets are derived by IMS from the high-level qualifier of the IMS catalog.

These additional system data sets are PDSE data sets and cannot be backed up by using the IMS image copy utilities. Instead, you can rebuild them from the recovered IMS catalog database data sets by using the IMS Catalog Directory Recovery utility (DFS3RU00), which recreates the data sets and loads the active ACBs into the IMS directory. Rebuilding the IMS catalog PDSEs by using this utility ensures that they match the current IMS catalog.

Alternatively, you can backup the ACBs in the IMS directory data sets to an ACBLIB data set by using the IMS Catalog Library Builder utility (DFS3LU00). During recovery, you rebuild the IMS directory from the backup ACBLIB by running the IMS Catalog Populate utility (DFS3PU00) with MANAGEDACBS=SETUP specified on the SYSINP DD statement.

You can also make backup copies of the additional IMS catalog data sets by using your preferred method of copying PDSE data sets. If you use backup copies of the IMS directory that are not created by IMS, you are responsible for ensuring that the copies match the current or recovered IMS catalog database.

Regardless of the method you use to back up the IMS directory, the active ACBs in the backup copy of the IMS directory must match exactly the active ACBs that are in the corresponding image copy of the IMS catalog database data sets that you would use for recovery.

When the IMS image copy utilities are used and the IMS catalog is managed by DBRC, the image copies and the required log data sets are managed by DBRC.

If the IMS catalog is shared between IMS systems, the IMS Catalog must be added to a change accumulation (CA) group so that the logs from all of the IMS systems can be merged before a recovery.

Related concepts:

➡️ IMS directory data sets (System Definition)

Related reference:

➡️ Backup utilities (Database Utilities)

➡️ IMS catalog utilities (System Utilities)

Recovering the IMS catalog when IMS manages ACBs

When the IMS management of ACBs is enabled, the IMS directory data sets must be recovered in addition to the other HALDB database data sets that are associated with the IMS catalog.

Depending on your system configuration, you can recover the IMS catalog data sets with either of the two methods:
- Recovering from the backup ACBLIB by using the IMS Catalog Populate (DFS3PU00) utility.
- Recovering from database image copies and the log data sets when the backup ACBLIB cannot be created.

Related reference:

➡️ IMS Catalog Directory Recovery utility (DFS3RU00) (System Utilities)

➡️ IMS Catalog Populate utility (DFS3PU00) (System Utilities)

Recovering the IMS catalog from ACBLIB when IMS manages ACBs

When IMS™ manages ACBs, you can recover the IMS catalog from a backup ACB library by running the IMS Catalog Populate utility (DFS3PU00). The backup ACB library can be created either when the SQL data definition language (DDL) is not used, or under strictly controlled conditions when the DDL is used.

You must have a backup ACBLIB that was created by the IMS Catalog Library Builder utility (DFS3LU00).

1. If DDL is used, check whether the RECON data sets contain any recovery or backout records for the IMS catalog HALDB database. Clean up the related records or flags by issuing the DELETE command to prevent record inconsistency in the RECON data sets.
2. Run the IMS Catalog Populate utility (DFS3PU00) as a batch job with the PSB DFSCPL00 to rebuild the IMS catalog from the backup ACBLIB. Ensure that you specify the MANAGEDACBS=SETUP control statement on the SYSINP DD statement and the backup ACBLIB as input on the IMSACB DD statement.

   The IMS Catalog Populate utility allocates the IMS directory data sets automatically and loads the ACBs into the IMS directory by using load mode.
3. Restart IMS.

Related concepts:

"HALDB partition initialization" on page 185

Related tasks:

"Creating HALDB databases with the HALDB Partition Definition utility" on page 542

Related reference:

➡️ HALDB Partition Definition utility (%DFSHALDB) (Database Utilities)

⇨ Database Description (DBD) Generation utility (System Utilities)

⇨ IMS Catalog Populate utility (DFS3PU00) (System Utilities)

Recovering the IMS catalog from image copies when IMS manages ACBs

When IMS™ manages ACBs, if the SQL data definition language (DDL) is used and you cannot create a backup ACB library, you can recover the IMS™ catalog only from an image copy of the main database data sets combined with the log data sets that are created since that image copy.

You must have an image copy of the main database data sets of the IMS catalog and the log data sets that are created since that image copy.

1. Recover the IMS catalog database by using one of the following PHIDAM recovery procedures that is appropriate to your IMS catalog:

   • "Recovering a PHIDAM database in a non-data-sharing environment" on page 617
   • "Recovering a PHIDAM database in a data sharing environment" on page 620

2. Run the IMS Catalog Directory Recovery utility (DFS3RU00) to rebuild the IMS directory data sets. Before you run the DFS3RU00 utility, ensure that the IMS catalog DBDs (DFSCD000, DFSCX000) and PSBs (DFSCP000, DFSCP001) are recovered so that the utility can be rerun in case of recovery failure.

   The IMS Catalog Directory Recovery utility does not support main storage databases (MSDB) and shared secondary index databases. The utility requires access to the DFSDF*xxx* member of the IMS.PROCLIB data set.

   Note: If you want to recover the IMS catalog HALDB but keep the current directory data sets, you can run the IMS Catalog Directory Recovery utility to check whether the recovered HALDB matches the existing directory data sets.

3. Restart IMS

Related reference:

⇨ IMS Catalog Directory Recovery utility (DFS3RU00) (System Utilities)

Chapter 7. Cleaning up the IMS directory data sets

In an IMS-managed ACBs environment, the Catalog Directory Recovery utility (DFS3RU00) can also be used to clean up the IMS directory data sets when the directory online updates aborted abnormally.

1. Ensure that no DDL definition activation process or the IMPORT DEFN SOURCE(CATALOG) command is in progress of updating the directory data sets. You can use one of the following options:

   - Enable internal resource lock manage (IRLM) in the JCL and in the IMS system to avoid process conflicts. This method is recommended.
   - If IRLM is not used, issue the QUERY MEMBER TYPE(IMS) SHOW() command to check the status of the IMPORT and DDL definition activation processing.

2. Run the IMS Catalog Directory Recovery utility.

Related concepts:

"Activating DDL-defined resources with the IMPORT DEFN command" on page 492

Related reference:

▶ IMS Catalog Directory Recovery utility (DFS3RU00) (System Utilities)

▶ QUERY MEMBER command (Commands)

Chapter 8. Maintaining the IMS catalog

The IMS catalog is an IMS HALDB full-function database, so even though it should not require much maintenance, the same maintenance considerations that apply to other HALDB full-function databases apply to the IMS catalog.

How often you need to reorganize or otherwise maintain the IMS catalog depends on how often your installation adds or updates the PSB and DBD records in the IMS catalog for new or changed databases and application programs in your IMS environment.

To reorganize the IMS catalog, use HALDB Online Reorganization (OLR) to avoid taking the IMS catalog offline. Because the IMS catalog is smaller than the typical HALDB database, running OLR on the IMS catalog should not significantly impact performance or resources.

If you can take the IMS catalog offline, you can reorganize the IMS catalog by using the HD Reorganization Unload utility (DFSURGU0) and the HD Reorganization Reload utility (DFSURGL0).

To clean out instances of PSBs and DBDs that are no longer needed from the IMS catalog, you can use the IMS Catalog Record Purge utility (DFS3PU10).

Related concepts:

Chapter 9, "Removing DBD and PSB instances from the IMS catalog," on page 57

Related reference:

➡ HD Reorganization Unload utility (DFSURGU0) (Database Utilities)

➡ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

➡ IMS Catalog Record Purge utility (DFS3PU10) (System Utilities)

Chapter 9. Removing DBD and PSB instances from the IMS catalog

You can remove the segments that represent individual DBD and PSB instances from the DBD and PSB records in the IMS catalog by using the IMS Catalog Record Purge utility (DFS3PU10).

You can also use the IMS Catalog Record Purge utility to delete all DBD instances of a particular DBD version from a DBD record in the IMS catalog. When a DBD version is deleted, all DBD instances of the DBD version are deleted from the DBD record. After deletion, the version of the DBD no longer exists in the IMS catalog.

When the IMS management of ACBs function is enabled, the IMS Catalog Record Purge utility deletes from the IMS directory the DBD and PSB instances that are also deleted from the IMS catalog. The DBD and PSB instances that are deleted from the IMS directory are also deleted from the directory staging data set.

The IMS Catalog Record Purge utility issues a CHKP (checkpoint) call to commit updates to the IMS catalog after every 200 updates. After 200 updates are made, the updates are committed to the IMS catalog only after the DBDs and PSBs that were identified for deletion since the previous CHKP call are deleted from the IMS directory.

To help avoid the unintentional deletion of DBD or PSB segments that you still need, you can define retention criteria for the DBD and PSB records in the IMS catalog. Based on the retention criteria in effect for each record in the catalog, the analysis function of the IMS Catalog Record Purge utility identifies and creates DELETE statements for the DBD or PSB segment instances that are eligible for deletion.

Retention criteria specific to a DBD or PSB record is set by the UPDATE control statement of the IMS Catalog Record Purge utility and are stored in the HEADER segment of the record. If no retention criteria is specified for a given record, catalog records are subject to the default retention criteria that is set by the RETENTION statement in the CATALOG section of the DFSDF*xxx* member of the IMS.PROCLIB data set.

The retention criteria includes the minimum number of segment instances IMS must retain in a DBD or PSB record and the minimum period of time segment instances must be retained before they can be deleted. By default, IMS retains a minimum DBD and PSB instances a record. There is no default for the period of time an instance must be retained. Periods of time are measured in days.

For example, if the number of instances of DBD or PSB segments in a record is equal to or less than the retention number that is set for the record, no instances can be deleted. If the number of days that a DBD or PSB instance has been in the IMS catalog is equal to or less than the retention period defined for the DBD or PSB record that contains it, the DBD or PSB instance cannot be deleted.

When you define your retention criteria for the segment instances in DBD and PSB records, keep in mind that each additional instance increases the amount of storage that is required for the IMS catalog.

Related reference:

"HEADER segment format" on page 61

➡ IMS Catalog Record Purge utility (DFS3PU10) (System Utilities)

➡ CATALOG and CATALOGxxxx sections of the DFSDFxxx member (System Definition)

Chapter 10. Using HALDB utilities with an unregistered IMS catalog

All HALDB utilities are supported for an IMS catalog database that is registered in the RECON data set and managed by DBRC. Some restrictions apply to an unregistered catalog.

You can use the IMS Catalog Partition Definition Data Set utility to configure an IMS catalog database that is not managed by DBRC. Some of the standard HALDB utilities can be used with an unregistered catalog database, with certain restrictions.

Database Image Copy utility (DFSUDMP0)
> You can use this utility to make batch image copies of an unregistered IMS catalog database. Concurrent image copying is not supported for unregistered IMS catalog databases. Additionally, you must specify the Datain DD statement because dynamic data set allocation is not supported for an unregistered catalog database.
>
> **Note:** See Chapter 6, "Backup and recovery of the IMS catalog," on page 47 for more information about recovering the IMS catalog database.

Batch Backout utility (DFSBBO00)
Database Recovery utility (DFSURDB0)
HALDB Index/ILE Dataset Rebuild utility (DFSPREC0)
HD Reorganization Unload utility (DFSURGU0)
HD Reorganization Reload utility (DFSURGL0)
> You can use these utilities with an unregistered IMS catalog database, but you must include the DFSDF= parameter for the utility EXEC statement. The DFSDF parameter specifies the 3-character suffix of the DFSDF*xxx* member of the IMS.PROCLIB dataset that specifies unregistered IMS catalog databases. The DFSDF*xxx* member specifies unregistered IMS catalog database names with the UNREGCATLG parameter of the DATABASE statement.

HALDB Partition Data Set Initialization utility (DFSUPNT0)
> This utility is not compatible with an unregistered IMS catalog database. The Catalog Populate utility (DFS3PU00) provides analogous support for registered and unregistered IMS catalog databases.

Related reference:

➡ HALDB Index/ILDS Rebuild utility (DFSPREC0) (Database Utilities)

➡ HD Reorganization Unload utility (DFSURGU0) (Database Utilities)

➡ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

➡ Database Recovery utility (DFSURDB0) (Database Utilities)

➡ Database Image Copy utility (DFSUDMP0) (Database Utilities)

➡ IMS catalog utilities (System Utilities)

Chapter 11. Format of records in the IMS catalog database

The IMS catalog database contains a unique record for each PSB and DBD defined during ACB generation. Each type of record, and each type of segment within a record, has a predefined format.

The catalog database segment types are grouped into four different data set groups (A - D) based on how frequently that segment type is accessed in database queries. The most frequently accessed segment types, such as the root segment type (HEADER), are located in data set group A. The least frequently accessed segment types, such as user remarks segments, are grouped in data set D.

Some segment types in the catalog database, such as DBDHXXX, are not currently used and are reserved for future development. The segment definitions are included in the catalog to allow for the implementation of future service and development enhancements without an unload and reload of the catalog database.

Related reference:

▶ IMS catalog data set groups (System Definition)

HEADER segment format

The IMS catalog HEADER segment type, also called the resource header, is the root segment type for the IMS catalog database.

The resource header for an IMS catalog database record contains information about the type of metadata that is stored in that record. The resource header indicates whether a specific IMS catalog record contains DBD or PSB metadata, and includes the IMS name and alias name of the resource that the catalog record describes.

The root key for a catalog record is the value of the RHDRSEQ field in this segment. This key value is generated by the IMS catalog populate utility (DFS3PU00) or the ACB generation and catalog populate utility (DFS3UACB). The value is created by concatenating the record type and the IMS member name of the resource. The record type is eight characters long and is right-padded with blank characters. The IMS member name is always eight characters long.

For example, the root key for a DBD record with the name ACF12000 is the following:

```
DBD     ACF12000
```

The root key for a PSB record with the name MXG88888 is the following:

```
PSB     MXG88888
```

The root key value is also used to sort catalog records into database partitions, if your catalog database consists of more than one partition. The partition high key for the last partition in the database must be high enough to contain the highest-key record in the catalog.

Segment name
> HEADER

Parent name
> Not applicable

> **Sequence field**
>> RHDRSEQ
>
> **Segment length**
>> 56 bytes

Table 4. HEADER segment map.

Offset (bytes)		Length (bytes)	Data type	Field name	Description	Unique key field
1		2	X	LEN	Length of this resource header segment	
3		2	X	CTL	Control field	
5		2	X	SEQNUM	Segment sequence number	
9		16	C	RHDRSEQ	Sequence field, type = U	X
	9	8	C	TYPE	Type of resource metadata in this catalog record	
	17	8	C	IMSNAME	Name of the resource described in this catalog record	
25		4	X	RETNINST	The minimum number of instances of the DBD or PSB that must be kept in this record when record segments are deleted. This value is modified with the UPDATE statement of the IMS Catalog Record Purge utility. If you do not provide a value for this field, the utility uses the value specified on the RETENTION statement in the CATALOG section of the DFSDF*xxx* member of the IMS.PROCLIB data set.	
29		4	X	RETNDAYS	Minimum number of days to keep each instance of the DBD or PSB in this record before the instances is eligible for deletion. This value is modified with the UPDATE statement of the IMS Catalog Record Purge utility. If you do not provide a value for this field, there are two possible scenarios: • If the RETNINST field contains a value of 1 or greater, the IMS Catalog Record Purge utility does not purge instances of this DBD or PSB based on their age. • If the RETINST field contains 0, the IMS Catalog Record Purge utility uses the values specified in the DFSDF*xxx* member to determine the retention criteria for the DBD or PSB instances in this record.	
33		8	C	FILLER1	Reserved	
41		7	C	ACTTS	Active record timestamp. Identifies the timestamp of the active member.	
48		7	C	PNDTS	Pending record timestamp. Identifies the timestamp of the pending member.	
55		2	C	FILLER2	Reserved	

DBD record segment formats

The DBD record segments in the IMS catalog are used to store information about an IMS database definition (DBD).

The following figure shows the high level organization of an IMS catalog record for a DBD:
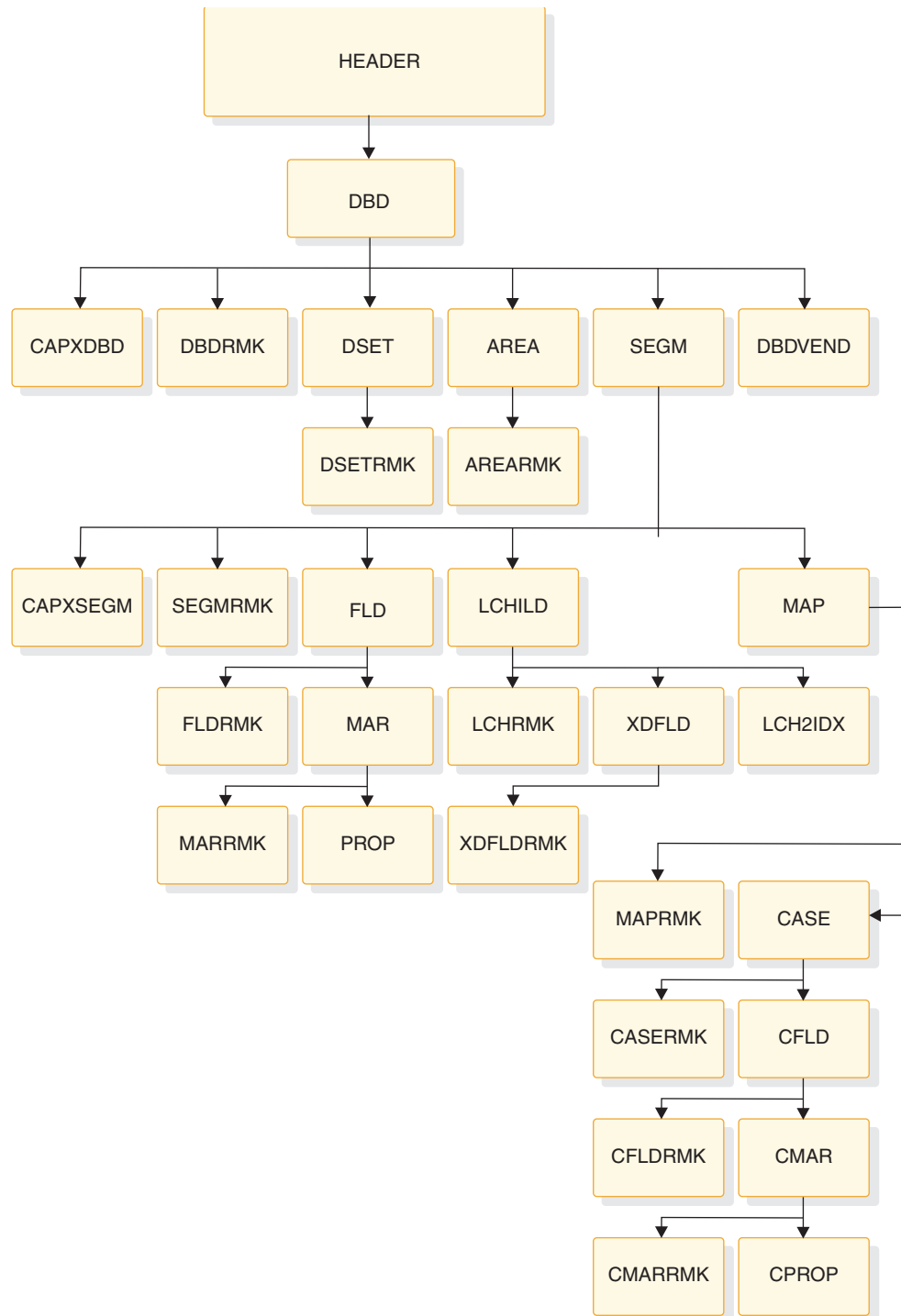
Figure 18. Format of an IMS catalog record for a database description

AREA segment type format

The IMS catalog AREA segment type contains information about a database area in a Fast Path database.

This segment type is used only in IMS catalog records for Fast Path databases.

Segment name
    AREA

Parent name
    DBD

Sequence field
    AREASEQ

Segment length
    40 bytes

Table 5. AREA segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment.	
3	2	X	CTL	Control field.	
5	2	X	AREASEQ	Sequence field, type = U.	X
5	2	X	SEQNUM	Sequence number.	
9	8	C	DD1	The data set name of this database area.	
17	2	X	SIZE	The size of the control interval of this database area in bytes.	
19	2	X	UOW1	The number of control intervals in a unit of work for this database area.	
21	2	X	UOW2	The number of control intervals in the overflow section of a unit of work for this database area.	
23	2	X	ROOT1	The total space allocated to the root addressable section of this database area. This value is given in number of units of work (UOW) for the UOW size given in field UOW1.	
25	2	X	ROOT2	The total space allocated for independent overflow in this database area. This value is given in number of units of work (UOW) for the UOW size given in field UOW2.	
27	14	C	FILLER	Reserved.	

Related concepts:

"AREA statement overview" on page 521

Related reference:

 ☞ AREA statement (System Utilities)

AREARMK segment type format

The IMS catalog AREARMK segment type contains user comments about a database area definition for a Fast Path database.

This segment is a direct child of the AREA segment instance that the comments pertain to.

Segment name
    AREARMK

Parent name
> AREA

Sequence field
> ARCMSEQ

Segment length
> 264 bytes

Table 6. AREARMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	ARCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this database area	

CAPXDBD segment type format

The IMS catalog CAPXDBD segment contains information about a Data Capture exit routine used by a DBD.

The metadata in this segment includes the name of the exit routine and processing options. Multiple DBDs can reference a single data capture exit, and a single DBD can reference multiple data capture exits. In the latter case, there are multiple child instances of this segment type for a single parent DBD segment instance.

Segment name
> CAPXDBD

Parent name
> DBD

Sequence field
> DDCAPSEQ

Segment length
> 32 bytes

Table 7. CAPXDBD segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Segment length	
3	2	X	CTL	Control field	
5	2	X	DDCAPSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	EXITNAME	The module name of this data capture exit routine	
17	1	C	LOG	Indicates if the data capture exit routine control blocks and data are written to the IMS system log	

Table 7. CAPXDBD segment map (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
18	1	C	KEY	Indicates if the data capture exit routine is passed the physical concatenated key of the segment that was updated when the exit routine is called	
19	1	C	PATH	Indicates if the exit routine is passed the data from each segment in the hierarchical path of the physical root segment	
20	1	C	DATA	Indicates if the physical segment data is passed to the data capture exit routine	
21	1	C	BEFORE	Indicates if Before data is included in type X'99' log records written for REPL calls	
22	1	C	DLET	Indicates if type X'99' log records are written for DLET calls	
23	1	C	CASCADE	Indicates if this data capture exit routine is called when a DL/I call deletes this segment as a result of deleting a parent segment	
24	1	C	CKEY	Indicates if the physical concatenated key is passed to the exit routine during a call that resulted from a cascade delete operation	
25	1	C	CPATH	Indicates if the data from each segment in the hierarchical path of the physical root segment is passed to the exit routine during a call that resulted from a cascade delete operation	
26	1	C	CDATA	Indicates if the physical segment data is passed to the exit routine during a call that resulted from a cascade delete operation	
27	1	C	CBEFORE	Indicates if Before data is included in type X'99' log records written for REPL calls for a DEDB	
28	1	C	CDLET	Indicates if type X'99' log records are written for DLET calls for a DEDB	
29	4	C	FILLER	Reserved	

Related tasks:

"DBD parameters for Data Capture exit routines" on page 394

Related reference:

➡ DBD statements (System Utilities)

CAPXSEGM segment type format

The IMS catalog CAPXSEGM segment type contains information about a Data Capture exit routine specified for a database segment.

The metadata in this segment includes the name of the exit routine and processing options. Multiple segments can reference a single data capture exit, and a single segment can reference multiple data capture exits. In the latter case, there are multiple child instances of this segment type for a single parent SEGM segment instance.

Segment name
> CAPXSEGM

Parent name
SEGM

Sequence field
SDCAPSEQ

Segment length
32 bytes

Table 8. CAPXSEGM segment map.

Offset (bytes)		Length (bytes)	Data type	Field name	Description	Unique key field
1		2	X	LEN	Length of this segment	
3		2	X	CTL	Control field	
5		2	X	SDCAPSEQ	Sequence field, type = U	X
	5	2	X	SEQNUM	Sequence number	
9		8	C	EXITNAME	The module name of this exit routine	
17		1	C	LOG	Indicates whether the data capture exit routine control blocks and data are written to the IMS system log	
18		1	C	KEY	Indicates whether the data capture exit routine is passed the physical concatenated key of the segment that was updated when the exit routine is called	
19		1	C	PATH	Indicates whether the exit routine is passed the data from each segment in the hierarchical path of the physical root segment	
20		1	C	DATA	Indicates whether the physical segment data is passed to the data capture exit routine	
21		1	C	BEFORE	Indicates whether before data is included in type X'99' log records written for REPL calls	
22		1	C	DLET	Indicates whether type X'99' log records are written for DLET calls	
23		1	C	CASCADE	Indicates whether this data capture exit routine is called when a DL/I call deletes this segment as a result of deleting a parent segment (during a cascade delete operation)	
24		1	C	CKEY	Indicates whether the physical concatenated key is passed to the exit routine during a call that resulted from a cascade delete operation	
25		1	C	CPATH	Indicates whether the data from each segment in the hierarchical path of the physical root segment is passed to the exit routine during a call that resulted from a cascade delete operation	
26		1	C	CDATA	Indicates whether the physical segment data is passed to the exit routine during a call that resulted from a cascade delete operation	
27		1	C	CBEFORE	Indicates whether Before data is included in type X'99' log records written for REPL calls for a DEDB	
28		1	C	CDLET	Indicates whether type X'99' log records are written for DLET calls for a DEDB	

Table 8. CAPXSEGM segment map (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
29	4	C	FILLER	Reserved	

Related tasks:

"DBD parameters for Data Capture exit routines" on page 394

Related reference:

➡ SEGM statements (System Utilities)

CASE segment type format

The IMS catalog CASE segment type contains information about a specific case for a mapping of an IMS database segment.

Segment name
>   CASE

Parent name
>   MAP

Sequence field
>   CASESEQ

Segment length
>   656 bytes

Table 9. CASE segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	CASESEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	1	C	CASETYPE	The encoding type of the CASEID field. This field specifies either C for Cp1047 EBCDIC encoding or X for a hexadecimal binary representation.	
10	7	C	FILLER01	Reserved	
17	128	C	CASENAME	The name of this case	
145	128	C	CASEID	The unique identifier for this case. Interpret this field based on the value of the CASETYPE field.	
273	128	C	MAPNAME	Name of the segment type mapping that this case belongs to	
401	256	C	FILLER02	Reserved	

Related reference:

➡ DFSCASE statements (System Utilities)

CASERMK segment type format

The IMS catalog CASERMK segment type contains user-specified comments about a case for a segment type mapping.

This segment is a direct child of the CASE segment instance that the comments pertain to.

Segment name
> CASERMK

Parent name
> CASE

Sequence field
> CASCMSEQ

Segment length
> 264 bytes

Table 10. CASERMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	CASCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the case definition described by the parent CASE segment	

CFLD segment type format

The IMS catalog CFLD segment type contains information about a field in a particular segment type format case.

Each instance of the CFLD segment describes a field for one case in a segment type format. The information in a CFLD segment instance is valid for a user database segment only if that segment is mapped with the mapping case defined in the parent CASE segment instance.

Segment name
> CFLD

Parent name
> CASE

Sequence field
> FIELDSEQ

Segment length
> 904 bytes

Table 11. CFLD segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	FIELDSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	IMSNAME	The 8-character IMS name of this field	
17	3	C	NAMESEQ	Indicates if this field is a sequence field	
20	1	C	SEQUM	Indicates whether this field is a unique sequence field (U) or non-unique (M)	
21	2	X	BYTES	The length of this field in bytes	
23	2	X	START	The starting offset of this field from the beginning of the segment, in bytes. If this field contains data, the STARTAFT field is not used.	
25	1	C	TYPE	Indicates what type of binary data IMS uses to pad empty space in this field: **X** Left-padded, X'00'. **P** Left-padded, X'00'. **C** Right-padded, X'40'. **F** Binary fullword data. Only used for MSDBs. **H** Binary halfword data. Only used for MSDBs.	
26	15	C	FILLER01	Reserved	
41	9	C	DATATYPE	The external (non-IMS) data type of the field	
50	3	C	FILLER02	Reserved	
53	2	X	PRECISN	The precision of a field with a decimal data type	
55	2	X	SCALE	The scale of a field with a decimal data type	
57	4	X	MINOCCUR	The minimum number of elements in a DATATYPE=ARRAY field	
61	4	X	MAXOCCUR	The maximum number of elements in a DATATYPE=ARRAY field	
65	4	X	MAXBYTES	The maximum number of bytes in a DATATYPE=ARRAY field or in a DATATYPE=STRUCT field that contains an array	
69	4	X	RELSTART	The relative starting position of the field in bytes	
73	128	C	NAME	The external alias name of this field	
201	128	C	PARENT	The external alias name of another field that this field is nested under	
329	128	C	REDEFINE	The external alias name of another field that this field can be redefined as. The field defined by this instance of the CFLD segment type and the field with the name specified in the REDEFINE field can be processed with a REDEFINES statement in a COBOL application.	

Table 11. CFLD segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
457	128	C	DEPENDON	The mapping selector field that the field defined by this instance of the CFLD segment type depends on	
585	128	C	CASENAME	The name of the mapping case that this field belongs to	
713	128	C	STARTAFT	The external alias name of the field that directly precedes this field in the segment. If this field contains data, the START field does not.	
841	64	C	FILLER03	Reserved	

Related reference:

➡ FIELD statements (System Utilities)

➡ DFSMARSH statements (System Utilities)

CFLDRMK segment type format

The IMS catalog CFLDRMK segment type contains user comments for a database field that is part of a specific segment type format case.

This segment is a direct child of the CFLD segment instance that the comments pertain to.

Segment name
CFLDRMK

Parent name
CFLD

Sequence field
CFLDCSEQ

Segment length
264 bytes

Table 12. CFLDRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	CFLDCSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Segment code	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the parent CFLD segment	

CMAR segment type format

The IMS catalog CMAR segment type contains information about a field marshaller definition in an IMS database that applies only to a specific case of a segment type format.

Each CASEFLD segment can have a CMAR child segment that contains data marshalling properties for the field.

Segment name
     CMAR

Parent name
     CFLD

Sequence field
     MARSHSEQ

Segment length
     704 bytes

Table 13. CMAR segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	MARSHSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	OVERFLOW	Reserved	
17	1	C	SIGN	For data with the data type of DECIMAL (data that uses either the PACKEDDECIMAL or ZONEDDECIMAL internal type converter), this field indicates if the data is a signed decimal value.	
18	6	C	FILLER01	Reserved	
24	25	C	ENCODING	Identifies the encoding type (code page) of the data in the field identified by the parent FLD segment	
49	50	C	PATTERN	Identifies the pattern mask to convert the data in the field identified by the parent FLD segment into a Java date object	
99	30	C	ITYPCONV	Identifies the internal type converter for the parent FLD segment. If this field contains data, the UTYPCONV field contains blanks. The internal type converter is used to convert IMS data to a specific type of Java data object.	
129	256	C	UTYPCONV	Identifies the user type converter for the parent FLD segment. If this field contains data, the ITYPCONV field contains blanks.	
385	256	C	URL	Reserved	
641	64	C	FILLER02	Reserved	

Related reference:

➡ DFSMARSH statements (System Utilities)

➡ DFSCASE statements (System Utilities)

CMARRMK segment type format

The IMS catalog CMARRMK segment type contains user-specified comments about a field marshaller definition in a specific case of a segment type mapping.

This segment is a direct child of the CMAR segment instance that the comments pertain to.

Segment name
> CMARRMK

Parent name
> CMAR

Sequence field
> CMARCSEQ

Segment length
> 264 bytes

Table 14. CMARRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	CMARCSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for marshalling properties that are defined in the parent CMAR segment	

CPROP segment type format

The IMS catalog CPROP segment type contains user-defined marshaller properties for a particular case of an IMS segment type mapping.

Segment name
> CPROP

Parent name
> CMAR

Sequence field
> CPROSEQ

Segment length
> 304 bytes

Table 15. CPROP segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	CPROSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	

Table 15. CPROP segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
9	128	C	NAME	Name of this user-defined marshalling property	
137	128	C	VALUE	User-defined value of this marshalling property	
265	40	C	FILLER	Reserved	

Related reference:

➡ DFSCASE statements (System Utilities)

➡ DFSMARSH statements (System Utilities)

DBD segment type format

The IMS catalog DBD segment type contains metadata about an IMS user database.

This information is collected from the parameters submitted to the DBDGEN utility during system definition.

Segment name
DBD

Parent name
HEADER

Sequence field
DBDSEQ

Segment length
552 bytes

Table 16. DBD segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SEQNUM	Sequence number	
9	17	X	DBDSEQ	Sequence field, type = U	X
9	4	X	CATVERS	Version number of the database definition that is recorded by this DBD segment and its dependents.	
13	13	C	TSVERS	ACB generation timestamp for this version, in the following format: *yyDDDHHmmssff*	
26	1	C	FILLER	Reserved	
27	2	X	RLVL	ACB generation utility release level	
29	7	C	ACCESS	DL/I database type for this database	
36	4	C	OSACC	Access method for this database	

Table 16. DBD segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
40	6	C	PROT	In a secondary index database, this field indicates if integrity protection is used for index pointer segments. • If this field contains PROT, a delete operation that removes an index pointer segment also removes the target segment pointer but the source segment is not deleted. • If this field contains NOPROT, an application program can replace all fields within a pointer segment except the constant, search, and subsequence control fields.	
46	7	C	DOSCOMP	Indicates that this database is a DLI/DOS index and that a DLI/DOS segment code is included in the prefix of segments in this database. IMS preserves the code during segment processing and provides a new code when segments are inserted.	
53	8	C	PSNAME	The name of the HALDB Partition Selection exit routine for this database.	
61	8	C	RMNAME	The module name of the randomizing exit routine for an HDAM or PHDAM database, or a Fast Path data entry database (DEDB).	
69	4	X	RMRBN	The maximum relative block number that the randomizing exit routine produces for this HDAM or PHDAM database. This value is also the number of control blocks or intervals in the root addressable area of the database.	
73	4	X	RMBYTES	The maximum number of bytes of user data that can be stored in the root addressable area of this database by an unbroken sequence of insert operations. A database record that exceeds this size is partially stored in the overflow area.	
77	2	X	RMANCH	The meaning of this field is different for Fast Path DEDB databases and full-function HDAM or PHDAM databases. For DEDB databases, indicates the type of randomizer: a value of 1 indicates a one-stage randomizer. A value of 2 indicates a two-stage randomizer. For HDAM and PHDAM databases, this value indicates the number of root anchor points in each control block or interval in the root addressable area of the HDAM or PHDAM database.	
79	1	C	RMXCI	Indicates if this DEDB uses the Extended Call Interface (XCI) when it calls the randomizing exit routine.	
80	3	C	FILLER01	Reserved	

Table 16. DBD segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
83	1	C	PASSWD	Indicates if this database uses a default VSAM password (the DBD name) to prevent accidental database operations by programs other than IMS.	
84	1	C	DATXEXIT	Indicates if this database uses the Data Conversion user exit routine DFSDBUX1.	
85	1	C	FPI	Indicates if this database is a secondary index for a Fast Path database.	
86	255	C	VERSION	User-supplied version information for this database	
341	2	C	FILLER02	Reserved	
343	2	X	IDXCNT	Number of shared secondary indexes	
345	8	C	IDXNM01	Shared secondary index name	
353	8	C	IDXNM02	Shared secondary index name	
361	8	C	IDXNM03	Shared secondary index name	
369	8	C	IDXNM04	Shared secondary index name	
377	8	C	IDXNM05	Shared secondary index name	
385	8	C	IDXNM06	Shared secondary index name	
393	8	C	IDXNM07	Shared secondary index name	
401	8	C	IDXNM08	Shared secondary index name	
409	8	C	IDXNM09	Shared secondary index name	
417	8	C	IDXNM10	Shared secondary index name	
425	8	C	IDXNM11	Shared secondary index name	
433	8	C	IDXNM12	Shared secondary index name	
441	8	C	IDXNM13	Shared secondary index name	
449	8	C	IDXNM14	Shared secondary index name	
457	8	C	IDXNM15	Shared secondary index name	
465	8	C	IDXNM16	Shared secondary index name	
473	8	C	CREATEBY	Reserved	
481	25	C	ENCODING	Code page used to encode all character data in this database. Individual segment and field definitions can override this value.	
506	47	C	FILLER03	Reserved	

DBDRMK segment type format

The IMS catalog DBDRMK segment type contains user-specified comments about a database definition.

This segment is a direct child of the DBD segment instance that the comments pertain to.

Segment name
DBDRMK

Parent name
DBD

Sequence field
DBDCMSEQ

Segment length
264 bytes

Table 17. DBDRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	DBDCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this DBD	

DBDVEND segment type format

The IMS catalog DBDVEND segment type contains a short header followed by a large block of unformatted space.

This segment type is reserved for use by vendor-supplied tools.

Segment name
DBDVEND

Parent name
DBD

Sequence field
DVNDSEQ

Segment length
4000 bytes

Table 18. DBDVEND segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SEQNUM	Segment sequence number	
7	2	C	FILLER	Reserved	
9	3992	X	DATA	Vendor product DBD data	
5	2	X	DVNDSEQ	Sequence field, type = U	X

DSET segment type format

The IMS catalog DSET segment type contains metadata about a data set group specification for an IMS database.

The information in this segment is generated based on the parameters of the DATASET statement of the DBDGEN utility. All DBD catalog records have at least one child DSET segment instance.

Segment name
DSET

Parent name
DBD

Sequence field
DSETSEQ

Segment length
96 bytes

Table 19. DSET segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	DSETSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	DD1	The name of the first data set in this data set group. • For HSAM, SHSAM and GSAM databases, this field is the name of the input data set. • For HISAM. SHISAM and INDEX databases, this field is the name of the primary data set. • For Fast Path databases, this field is the name of a defined data area. MSDBs and logical databases do not use this field.	
17	8	C	DD2	The name of the output data set for HSAM, SHSAM, and GSAM databases. If no name is specified for a GSAM database, DD1 is used as the output data set.	
25	8	C	OVERFLOW	The name of the overflow data set in this group.	
33	2	X	BLOCK1	Blocking factor 1 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	
35	2	X	BLOCK2	Blocking factor 2 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	
37	2	X	SIZE1	Block size 1 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	
39	2	X	SIZE2	Block size 2 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	

Table 19. DSET segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
41	2	X	RECORD1	Record size 1 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	
43	2	X	RECORD2	Record size 2 for the data set group. See the DATASET statement of the DBDGEN utility for usage information.	
45	2	X	SCAN	The number of DASD cylinders that are scanned for free storage during a segment insert operation. Used only for HIDAM and HDAM databases.	
47	2	X	SEARCHA	The type of algorithm used to search for free storage during a segment insert operation. Used only for HIDAM and HDAM databases. The different type codes and meanings are: **0** IMS chooses which algorithm to use. **1** IMS does not search for space in the second-most desirable block or control interval. **2** IMS includes a search for space in the second-most desirable block or control interval.	
49	2	C	RECFM	The format of records in this data set group for a GSAM database: **F** Fixed-length **FB** Fixed-length and blocked **V** Variable-length **VB** Variable-length and blocked **U** Undefined length	
51	2	X	FRSPFBFF	The number data blocks per block of free space that are allocated in an HDAM or HIDAM database.	
53	2	X	FRSPFSPF	The minimum percentage of free space in each control block or interval in this data set group. Used only for HDAM and HIDAM databases.	

Table 19. DSET segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
55	8	C	REL1	The terminal relationship type and segment ownership type in an MSDB:	
				NO Non-terminal-related without terminal keys	
				TERM Non-terminal-related with the LTERM name as the key	
				FIXED Terminal-related with the LTERM name as the key, with segment insertions and deletions disabled	
				DYNAMIC Terminal-related with the LTERM name as a key, with segment insertions and deletions enabled	
63	8	C	REL2	The name of the pseudo-sequence field for a keyed MSDB. Segment search arguments can use the name of this pseudo-field and the LTERM name as the key value.	
71	26	C	FILLER	Reserved	

DSETRMK segment type format

The IMS catalog DSETRMK segment type contains user-specified comments about a data set group definition.

This segment is a direct child of the DSET segment instance that the comments pertain to.

Segment name
DSETRMK

Parent name
DSET

Sequence field
DSCMSEQ

Segment length
264 bytes

Table 20. DSETRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	DSCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this DSET	

FLD segment type format

The IMS catalog FLD segment type contains metadata about a field in an IMS database.

Each instance of the FLD segment describes a field for one segment in a database. This information is collected during system generation from the FIELD statement of the DBDGEN utility.

Segment name
> FLD

Parent name
> SEGM

Sequence field
> FLDSEQ

Segment length
> 904 bytes

Table 21. FLD segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	FLDSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	IMSNAME	The 8-character IMS name of this field	
17	3	C	NAMESEQ	Indicates if this field is a sequence field	
20	1	C	SEQUM	Indicates whether this field is a unique sequence field (U) or a non-unique sequence field (M)	
21	2	X	BYTES	The length of this field in bytes	
23	2	X	START	The starting offset of this field from the beginning of the segment, in bytes. If this field contains data, the STARTAFT field is not used.	
25	1	C	TYPE	Indicates what type of binary data IMS uses to pad empty space in this field:	
				X Left-padded, X'00'	
				P Left-padded, X'00'	
				C Right-padded, X'40'	
				F Binary fullword data. Used only for MSDBs.	
				H Binary halfword data. Used only for MSDBs.	
26	15	C	FILLER01	Reserved	
41	9	C	DATATYPE	The external data type of the field	
50	3	C	FILLER02	Reserved	
53	2	X	PRECISN	The precision of a field with a decimal data type	
55	2	X	SCALE	The scale of a field with a decimal data type	

Table 21. FLD segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
57	4	X	MINOCCUR	The minimum number of elements in a DATATYPE=ARRAY field	
61	4	X	MAXOCCUR	The maximum number of elements in a DATATYPE=ARRAY field	
65	4	X	MAXBYTES	The maximum number of bytes in a DATATYPE=ARRAY field or in a DATATYPE=STRUCT field that contains an array	
69	4	X	RELSTART	The relative starting offset in bytes from the end of a dynamic array or struct if an array or struct occurs in the parent segment before this field	
73	128	C	NAME	The external alias name of this field	
201	128	C	PARENT	The external alias name of another field that this field is nested under	
329	128	C	REDEFINE	The external alias name of another field that this field can be redefined as. This field and the field indicated with the REDEFINE field can be processed with a REDEFINES statement in a COBOL application.	
457	128	C	DEPENDON	This field contains blanks	
585	128	C	CASENAME	This field contains blanks	
713	128	C	STARTAFT	The external alias name of the field that directly precedes this field in the segment. If this field contains data, the START field does not.	
841	64	C	FILLER03	Reserved	

FLDRMK segment type format

The IMS catalog FLDRMK segment type contains user comments for a database field.

This segment is a direct child of the FLD segment instance that the comments pertain to.

Segment name
> FLDRMK

Parent name
> FLD

Sequence field
> FLDCMSEQ

Segment length
> 264 bytes

Table 22. FLDRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique Key Field
1	2	X	LEN	Length of this segment	

Table 22. FLDRMK segment map (continued).

Offset (bytes)		Length (bytes)	Data type	Field name	Description	Unique Key Field
3		2	X	CTL	Control field	
5		2	X	FLDCMSEQ	Sequence field, type = U	X
	5	2	X	SEQNUM	Sequence number	
7		2	C	FILLER	Reserved	
9		256	C	REMARKS	User comments for the parent FLD	

LCH2IDX segment type format

The IMS catalog LCH2IDX segment type contains information about a Fast Path secondary index specified on the LCHILD statement defined in the parent LCHILD segment.

Segment name
> LCH2IDX

Parent name
> LCHILD

Sequence field
> LCH2ISEQ

Segment length
> 24 bytes

Table 23. LCH2IDX segment map.

Offset (bytes)		Length (bytes)	Data type	Field name	Description	Unique key field
1		2	X	LEN	Length of this segment	
3		2	X	CTL	Control field	
5		2	X	LCH2ISEQ	Sequence field, type = U	X
	5	2	X	SEQNUM	Sequence number	
7		2	C	FILLER01	Reserved	
9		8	C	IMSNAME	IMS name	
17		8	C	DBDNAME	Target secondary index name	

LCHILD segment type format

The IMS catalog LCHILD segment type contains information about a relationship between segment types.

Segment name
> LCHILD

Parent name
> SEGM

Sequence field
> LCHLDSEQ

Segment length
72 bytes

Table 24. LCHILD segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	LCHILDSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	IMSNAME	Name of the segment type that is associated with the parent segment type in this logical relationship	
17	8	C	DBNAME	Name of the database that contains the segment identified by the IMSNAME field	

Table 24. LCHILD segment map (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
25	4	C	PTR	Identifies the type of pointer used in this logical relationship:	
				SNGL Indicates that a logical child first pointer field is reserved in the parent segment type. This field is used in one of three ways:	
				• For a logical parent relationship, the pointer field contains a direct address pointer to the first occurrence of a logical child segment type.	
				• For a HIDAM primary index database relationship, the pointer field contains a direct address pointer to a HIDAM root database segment.	
				• For a secondary index relationship, the pointer field contains a direct address pointer to an index target segment.	
				DBLE Indicates that two 4-byte pointer fields are reserved in the logical parent segment type. The first pointer field contains the address of the first occurrence of the logical child segment type, and the second pointer field contains the address of the last occurrence of the logical child segment type.	
				NONE No pointer fields are reserved in the logical parent segment type. The relationship between the logical parent and logical child is either not implemented or is maintained with physically paired segments.	
				INDX For the first logical child relationship in a HIDAM database, this value indicates that the parent segment is the root segment type in a HIDAM database and the target segment is the root segment of the primary index for the database. For subsequent logical child relationships in a HIDAM database and for other databases, this value indicates that the target segment type is a secondary index target for this database.	
				SYMB This value indicates that the pointer field in the primary database does not contain direct target addresses to the target segments in the secondary index database. Instead, the pointer field contains the concatenated key of the target segment. In a secondary index database, this value indicates that no space is reserved in the index pointer segments for the address of the target segment.	

Table 24. LCHILD segment map (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
29	8	C	PAIR	Name of the segment paired with the segment identified in the IMSNAME field in a bidirectional logical relationship	
37	8	C	INDEX	The name of the sequence field of the root segment type of the HIDAM database that this database is the primary index for. This field contains data only if this database is the primary index for a HIDAM database.	
45	5	C	RULES	Indicates how virtual logical child segments are sequenced during a DL/I insert operation when they do not include a sequence field or use a non-unique sequence field:	
				FIRST If no sequence field exists, new segment instances are inserted before the first existing instance of the logical child. If a non-unique sequence field exists, new segment instances are inserted before all existing instances that have the same key value as the new instance.	
				LAST If no sequence field exists, new segment instances are inserted after the last existing instance of the logical child. If a non-unique sequence field exists, new segment instances are inserted after all existing instances with the same key value as the new instance.	
				HERE The new instance is inserted at the location of the cursor after the last DL/I call. If there is no current position, FIRST is used instead.	
50	1	C	MULTI	Indicates whether the LCHILD statement is a member of a multiple secondary index segment group	
51	2	C	FILLER01	Reserved	
53	4	X	RKSIZE	The root key size of the target databases. This field is only used for partitioned secondary index databases.	
57	16	C	FILLER02	Reserved	

Related concepts:

"Logical relationship types" on page 248

"The logical child segment" on page 263

Related reference:

➦ LCHILD statements (System Utilities)

LCHRMK segment type format

The IMS catalog LCHRMK segment type contains user-specified comments about a logical child relationship in an IMS database.

This segment is a direct child of the LCHILD segment that comments pertain to.

Segment name
> LCHRMK

Parent name
> LCHILD

Sequence field
> LCHCMSEQ

Segment length
> 264 bytes

Table 25. LCHRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	LCHCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this logical relationship	

MAP segment type format

The IMS catalog MAP segment type contains information about a segment type mapping in an IMS database segment.

Segment name
> MAP

Parent name
> SEGM

Sequence field
> MAPSEQ

Segment length
> 520 bytes

Table 26. MAP segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	MAPSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	128	C	NAME	Name of this map	

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
137	128	C	DEPENDON	The external name of the control field within this segment (found in the NAME field of the FLD catalog record) that determines which map case is used for each mapped segment instance in the user database.	
265	256	C	FILLER	Reserved	

Related reference:

◢ DFSMAP statements (System Utilities)

MAPRMK segment type format

The IMS catalog MAPRMK segment type contains user comments for a segment type mapping definition.

Segment name
> MAPRMK

Parent name
> MAP

Sequence field
> MAPCMSEQ

Segment length
> 264 bytes

Table 27. MAPRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	MAPCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the parent MAP segment	

MAR segment type format

The IMS catalog MAR segment type contains information about a field marshaller definition in an IMS database.

Each FLD segment can have a MAR child segment that contains data marshalling properties for the field. The information in this segment type is generated from the input parameters of the DFSMARSH statement of the DBDGEN utility.

Segment name
> MAR

Parent name
> FLD

Sequence field
> MARSEQ

Segment length
> 704 bytes

Table 28. MAR segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	MARSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	8	C	OVERFLOW	Reserved	
17	1	C	SIGN	For data with the data type of DECIMAL (data that uses either the PACKEDDECIMAL or ZONEDDECIMAL internal type converter), this field indicates if the data is a signed decimal value.	
18	6	C	FILLER01	Reserved	
24	25	C	ENCODING	Identifies the encoding type (code page) of the data in the field identified by the parent FLD segment	
49	50	C	PATTERN	Identifies the pattern mask to convert the data in the field identified by the parent FLD segment into a Java date object	
99	30	C	ITYPCONV	Identifies the internal type converter for the parent FLD segment. If this field contains data, the UTYPCONV field contains blanks. The internal type converter is used to convert IMS data into a specific type of Java data object.	
129	256	C	UTYPCONV	Identifies the user type converter for the parent FLD segment. If this field contains data, the ITYPCONV field contains blanks.	
385	256	C	URL	Reserved	
641	64	C	FILLER02	Reserved	

MARRMK segment type format

The IMS catalog MARRMK segment type contains user comments for a field marshaller definition.

Segment name
> MARRMK

Parent name
> MAR

Sequence field
> MARCMSEQ

Segment length
> 264 bytes

Table 29. MARRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	MARCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the parent MAR segment	

PROP segment type format

The IMS catalog PROP segment type contains a user-defined marshaller property definition.

Segment name
PROP

Parent name
MAR

Sequence field
PROPSEQ

Segment length
304 bytes

Table 30. PROP segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	PROPSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	128	C	NAME	Name of this user-defined marshaller property	
137	128	C	VALUE	User-defined marshaller property information	
265	40	C	FILLER	Reserved	

Related reference:

➡ DFSMARSH statements (System Utilities)

SEGM segment type format

The IMS catalog SEGM segment type contains metadata about an IMS database segment.

This information is generated based on the parameters of the SEGM statement of the DBDGEN utility.

Segment name
SEGM

Parent name
DBD

Sequence field
SEGMSEQ

Segment length
376 bytes

Table 31. SEGM segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment.	
3	2	X	CTL	Control field.	
5	2	X	SEGMSEQ	Sequence field, type = U.	X
5	2	X	SEQNUM	Sequence number.	
9	8	C	IMSNAME	Name of this segment.	
17	8	C	PARPHY	Name of the logical parent of this segment.	
25	4	C	PARTYPE	Type of physical child pointers (SNGL or DBLE) that are included in all occurrences of the physical parent of this segment.	
29	8	C	PARLOG	Name of the logical parent of this segment.	
37	8	C	PARCHK	Indicates whether the concatenated key of the logical parent is virtual or physical.	
45	8	C	DBNAME	Name of the database that the logical parent of this segment is defined in.	
53	4	X	BYTE1	Maximum length of a variable-length segment in bytes, or the number of bytes in the data area of a fixed-length segment.	
57	4	X	BYTE2	Minimum length of a variable-length segment in bytes. If this field contains data, this segment type is variable-length.	
61	4	X	FREQ	Estimated number of times that this segment occurs for each instance of the physical parent. This value is used by IMS to determine the logical record length and physical storage block sizes for data set groups in the database that contains this segment type.	
65	3	C	RULE1	Path type that must be used to insert, delete, or replace an instance of this segment type. This field contains three characters. The first character is the path type for insert operations, the second character is the path type for delete operations, and the third character is the path type for replace operations.	
68	5	C	RULE2	The rule that IMS uses when adding a new instance of a segment type that does not have a unique sequence field:	

Table 31. SEGM segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
73	8	C	SRCSEG1	In a catalog record for a virtual logical child segment type, this is the name of the real logical child that corresponds with this virtual logical child.	
				In a catalog record for a segment type in a logical database, this is the name of the source segment in a physical database that is being defined as a logical segment type.	
81	4	C	SRCFBK1	In a catalog record for a virtual logical child segment type, this field indicates if only the key of the real logical child or both the key and data portions of the real logical child are used to construct this segment type in the user I/O area.	
				This field is only used for segments in logical databases.	
85	8	C	SRCDBN1	Name of the database that contains the segment type identified in the SRCSEG1 field.	
93	8	C	SRCSEG2	In a catalog record for a concatenated virtual logical child segment type, this is the name of the physical parent of the real logical child segment.	
				In a catalog record for a segment in a logical database, this is the name of the logical or physical parent segment in a physical database that is used to construct the destination parent section of this logical concatenated segment.	
				If this field contains data, this segment is a logical concatenated segment.	
101	4	C	SRCFBK2	In a catalog record for a concatenated virtual logical child segment type, this field indicates if only the key of the real logical child or both the key and data portions of the real logical child are used to construct this segment type in the user I/O area.	
				The key value of a concatenated segment is either the value in the physical twin sequence field or the logical twin sequence field, depending on which path the logical child is accessed from.	
				This field is used only for concatenated segments in logical databases.	
105	8	C	SRCDBN2	Name of the database that contains the segment type identified in the SRCSEG2 field.	
113	8	C	COMPRTN	Name of the Segment Edit/Compression exit routine used for this segment.	

Table 31. SEGM segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
121	4	C	COMPDATA	Indicates whether the Segment Edit/Compression exit routine for this segment only condenses or modifies data fields and not sequence fields.	
125	4	C	COMPINIT	Indicates if initialization and termination processing control is required for the Segment Edit/Compression exit routine identified in the COMPRTN field.	
129	4	X	COMPMAX	Indicates the maximum expansion size (in bytes) for this segment when it is modified by the Segment Edit/Compression exit routine identified in the COMPRTN field.	
133	3	C	COMPPAD	Indicates that a segment instance will be padded to the size given in the COMPMAX field if the Segment Edit/Compression exit routine compresses it to a smaller size.	
136	1	C	FILLER01	Reserved.	
137	7	C	PTR1	Indicates if pointer fields are reserved in the segment prefix for a HIER (hierarchic forward pointer), HIERBWD (hierarchic forward and backward pointers), TWIN (twin forward pointer), TWINBWD (twin forward and backward pointers), or NOTWIN (no reserved field for physical twin pointers) relationship. See the POINTER= parameter of the SEGM statement in the DBDGEN utility for more details about these values.	
144	8	C	PTR2	Indicates if pointer fields are reserved in the segment prefix for a LTWIN (logical twin forward pointer) or LTWINBWD (logical twin forward and backward pointers) relationship.	
152	6	C	PTR3	Indicates if a pointer field is reserved in the segment prefix for a LPARNT (pointer to a logical parent segment) relationship.	
158	3	C	PTR4	Indicates if a 4-byte field is reserved in the segment prefix for a logical relationship counter.	
161	6	C	PTR5	Indicates if this segment type is part of a bidirectional logical relationship.	
167	2	X	SSPTR	The number of subset pointers. A value of 0 in this field indicates that subset pointers are not used in this segment type.	
169	3	C	TYPE	Indicates the type of DEDB dependent segment for this segment type, either sequential or dependent.	
172	1	C	DSGRP	The data set group identifier for this segment. This field is only used for segments in a HALDB.	
173	2	X	DSGHAL	Reserved for internal use.	
175	12	C	FILLER02	Reserved.	
187	128	C	NAME	The external alias name for this segment.	

Table 31. SEGM segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
315	25	C	ENCODING	The code page used to encode all character data in this segment.	
340	37	C	FILLER03	Reserved.	

Related concepts:

"SEGM statement overview" on page 521

Related reference:

SEGM statements (System Utilities)

SEGMRMK segment type format

The IMS catalog SEGMRMK segment type contains user comments for a database segment.

Segment name
>SEGMRMK

Parent name
>SEGM

Sequence field
>SGMCMSEQ

Segment length
>264 bytes

Table 32. SEGMRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SGMCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the parent SEGM	

XDFLD segment type format

The IMS catalog XDFLD segment type contains metadata about an indexed field in a secondary index relationship.

Each XDFLD segment instance is a direct child of an LCHILD segment instance that defines a secondary index relationship.

Segment name
>XDFLD

Parent name
>LCHILD

Sequence field
XDFLDSEQ

Segment length
200 bytes

Table 33. XDFLD segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment.	
3	2	X	CTL	Control field.	
5	2	X	XDFLDSEQ	Sequence field, type = U.	X
5	2	X	SEQNUM	Sequence number.	
9	8	C	NAME	Name of this field.	
17	8	C	SEGMENT	Indicates the name of the index source segment type for the secondary index relationship.	
25	8	C	SRCH1	Indicates one of up to five fields in the index source segment that can be used as search fields in the secondary index.	
33	8	C	SRCH2	Indicates one of up to five fields in the index source segment that can be used as search fields in the secondary index.	
41	8	C	SRCH3	Indicates one of up to five fields in the index source segment that can be used as search fields in the secondary index.	
49	8	C	SRCH4	Indicates one of up to five fields in the index source segment that can be used as search fields in the secondary index.	
57	8	C	SRCH5	Indicates one of up to five fields in the index source segment that can be used as search fields in the secondary index.	
65	8	C	SUBSEQ1	Indicates one of up to five fields in the index source segment that are used as the subsequence field of the secondary index.	
73	8	C	SUBSEQ2	Indicates one of up to five fields in the index source segment that are used as the subsequence field of the secondary index.	
81	8	C	SUBSEQ3	Indicates one of up to five fields in the index source segment that are used as the subsequence field of the secondary index.	
89	8	C	SUBSEQ4	Indicates one of up to five fields in the index source segment that are used as the subsequence field of the secondary index.	
97	8	C	SUBSEQ5	Indicates one of up to five fields in the index source segment that are used as the subsequence field of the secondary index.	
105	8	C	DDATA1	Indicates one of up to five fields in the index source segment that are used as the duplicate data field of the secondary index.	

Table 33. XDFLD segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
113	8	C	DDATA2	Indicates one of up to five fields in the index source segment that are used as the duplicate data field of the secondary index.	
121	8	C	DDATA3	Indicates one of up to five fields in the index source segment that are used as the duplicate data field of the secondary index.	
129	8	C	DDATA4	Indicates one of up to five fields in the index source segment that are used as the duplicate data field of the secondary index.	
137	8	C	DDATA5	Indicates one of up to five fields in the index source segment that are used as the duplicate data field of the secondary index.	
145	8	C	EXITRTN	The name of the user-supplied Secondary Index Database Maintenance exit routine for this secondary index relationship.	
153	8	C	PSELRTN	The name of the user-supplied Partition Selection exit routine that is used when user partitioning is requested for this HISAM or SHISAM database that is defined as secondary index for a Fast Path primary database.	
161	1	C	PSELOPT	Indicates how partition databases in a user partition group are logically grouped for GN calls that can process past the end of the first partition: **M (multiple grouping)** The selected user partition and subsequent partitioned databases are included in the group as they are defined in the NAME field of the LCHILD segment instance in the primary DEDB catalog record. **S (single grouping)** Only the selected user partition database is used.	
162	3	C	FILLER01	Reserved.	
165	5	C	CONSTANT	Indicates a character that identifies every index pointer in a particular secondary index. This value differentiates pointers for different secondary indexes that are stored in the same database.	
170	5	X	NULLVAL	The pointer suppression value for index search fields. No index pointers are created when all of the SRCH fields of the index source segment contain this value.	
175	26	C	FILLER	Reserved.	
175	26	C	FILLER02	The external alias name of this XDFLD.	

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

Related reference:

➡ XDFLD statements (System Utilities)

➡ Secondary Index Database Maintenance exit routine (Exit Routines)

XDFLDRMK segment type format

The IMS catalog XDFLDRMK segment type contains user-specified comments about the XDFLD segment type defined by the parent XDFLD segment instance.

Segment name
XDFLDRMK

Parent name
XDFLD

Sequence field
XDFRMSEQ

Segment length
264 bytes

Table 34. XDFLDRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	XDFRMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	X	RMKLEN	Remarks length	
9	256	C	REMARKS	User-specified comments for the parent XDFLD statement definition	

PSB record segment formats

The PSB record segments in the IMS catalog are used to store information about application program views and schemas of IMS databases, otherwise known as program specification blocks (PSBs) and program communication blocks (PCBs).

The following figure shows the high-level organization of an IMS catalog record for a PSB:
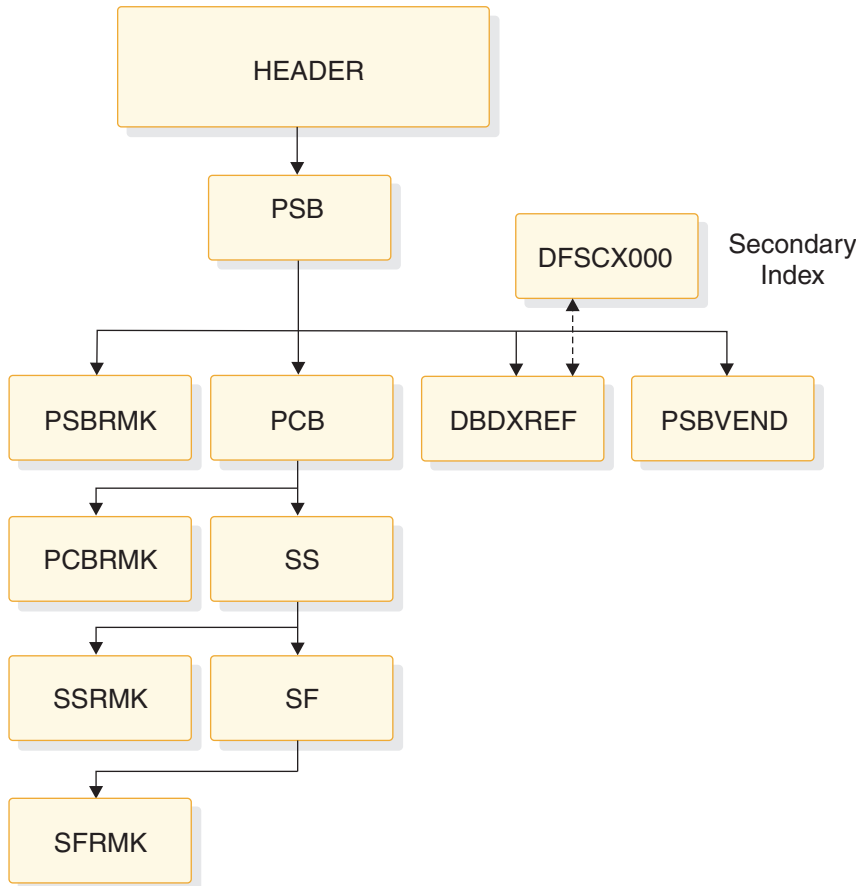
Figure 19. Format of an IMS catalog record for a program specification block

DBDXREF segment type format

The IMS catalog DBDXREF segment type contains metadata about a DBD in the intent list of a program specification block (PSB).

Segment name
DBDXREF

Parent name
PSB

Sequence field
DBDXSEQ

Segment length
48 bytes

Table 35. DBDXREF segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length	
3	2	X	CTL	Control field	
5	2	X	DBDXSEQ	Sequence field, type = U	X

Table 35. DBDXREF segment map (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
5	2	X	SEQNUM	Sequence number	
9	4	X	CATVERS	Catalog version number	
13	13	C	TSVERS	ACB generation timestamp for this version, in the following format: *yyDDDDHHmmssff*	
26	3	C	FILLER	Reserved	
29	8	C	IMSNAME	Name of the DBD	
37	8	C	PSBNAME	Name of the PSB that includes the DBD named in the IMSNAME field in the PSB intent list	

Related concepts:

PCB segment type format

The IMS catalog PCB segment type contains metadata about a program control block definition.

Information in this segment type is generated based on the parameters of the PCB statement of the PSBGEN utility.

Segment name
> PCB

Parent name
> PSB

Sequence field
> PCBSEQ

Segment length
> 288 bytes

Table 36. PCB segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	PCBSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	IMSNAME	The name of the physical or logical DBD that is the primary source of database segments for this PCB. Other databases may be added to the logical data structure for this PCB with secondary indexes and cross references. In an alternate PCB, this field is the name of the LTERM that the output message is sent to.	
17	8	C	PCBNAME	8-character IMS name of this PCB based on the PCBNAME or LABEL parameter of the PCB statement.	

Table 36. PCB segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
25	8	C	LABEL	8-character IMS name of this PCB based on the PCB label parameter. If this field contains data, the PCBNAME field contains blanks.	
33	4	C	TYPE	Identifies whether this PCB is a standard or alternate PCB. A standard PCB returns its output message to the source of the input message. An alternate PCB returns the message to a different destination such as a terminal or transaction queue. **DB** Standard PCB **TP** Alternate PCB	
37	4	C	PROCOPT	Identifies the processing options for this PCB. Processing options define what types of operations an application program using the PCB can perform. There can be up to four options for one PCB.	
41	8	C	PROCSEQ	The name of a secondary index for the database identified in the IMSNAME field. Application programs that use this PCB use the processing sequence of the secondary index rather than the primary database.	
49	8	C	PROCSEQD	The name of a secondary index for the Fast Path database identified in the IMSNAME field. Application programs that use this PCB use the processing sequence of the secondary index rather than the primary Fast Path database.	
57	2	X	KEYLEN	The number of bytes in the longest concatenated key for a hierarchical path of sensitive segments used in the data structure accessed with this PCB.	
59	2	X	COPIES	The number of runtime copies that exist for this PCB. This value is used for XQUERY processing.	
61	4	C	VIEW	Identifies that this PCB for a Fast Path database uses either the DEDB commit view or the MSDB commit view.	
65	1	C	ALTRESP	Identifies if this alternate PCB can be used instead of the standard I/O PCB for terminal response messages in response mode, conversational mode, or exclusive mode.	
66	1	C	EXPRESS	Identifies whether queued messages are sent (Y) or backed out (N) by this alternate PCB if the application program using it abends.	
67	1	C	MODIFY	Specifies if the destination name for this alternate PCB can be dynamically modified.	
68	1	C	SAMETRM	Identifies if IMS verifies that the destination logical terminal for this alternate PCB is the same as the logical terminal that sent the input message.	
69	1	C	SB	Identifies if this PCB is buffered with sequential buffering when possible.	

Table 36. PCB segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
70	1	C	POS	Identifies whether this PCB uses single (S) or multiple (M) positioning in the target data structure.	
71	1	C	LIST	Identifies if this PCB is included in the PCB list passed to an application program when it is given control.	
72	1	C	PSELOPT	Indicates how this PCB logically groups user partition databases for qualified GN calls without SSA processing before the end of the data is reached in the user partition databases: **M** The selected user partition database and subsequent user partition databases within a user data partition are grouped as they are physically defined in the NAME field of the LCHILD definition of the primary Fast Path database DBD. **S** Only the selected user partition database is used by the PCB. Subsequent user partition databases are not added to the logical group. This field is only used for Fast Path secondary index databases.	
73	1	C	FILLER01	Reserved	
74	7	C	ACCESS	Indicates whether this PCB accesses the target database using the normal secondary index or a separate logical database.	
81	128	C	NAME	External alias name of this PCB	
209	13	C	DBDTS	DBDGEN timestamp	
222	4	X	DBVER	Requested DBD version	
226	63	N/A	N/A	Reserved bytes	

Related concepts:

"Coding program specification blocks as input to the PSBGEN utility" on page 525

Related reference:

➡ Full-function or Fast Path database PCB statement (System Utilities)

➡ Alternate PCB statement (System Utilities)

PCBRMK segment type format

The IMS catalog PCBRMK segment type contains user-specified comments about an IMS program control block.

This segment is a direct child of the PCB segment instance that the comments pertain to.

Segment name
PCBRMK

Parent name
PCB

Sequence field
PCBCMSEQ

Segment length
264 bytes

Table 37. PCBRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	PCBCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this PCB	

PSB segment type format

The IMS catalog PSB segment type contains metadata about an IMS program specification block.

Segment name
PSB

Parent name
HEADER

Sequence field
PSBSEQ

Segment length
88 bytes

Table 38. PSB segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SEQNUM	Sequence number	
9	17	X	PSBSEQ	Sequence field, type = U	X
9	4	X	CATVERS	Catalog version number	
13	13	C	TSVERS	ACB generation timestamp for this version, in the following format: *yyDDDHHmmssff*	
26	1	C	FILLER	Reserved	
27	2	X	RLVL	ACB generation utility release level	
37	6	C	LANG	Compiler language for the message or batch processing program used by this application	

Table 38. PSB segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
43	2	X	IOERCC	The condition code returned to the operating system when IMS terminates normally and one or more errors occurred on any database during the execution of a program that used this PSB. If this value is 451 and the IMS abend code is U451, IMS terminates with a U451 abend instead of issuing a condition code. If the IMS abend code is not U451, IMS also issues a DFS0426I message.	
45	4	C	IOERWTOR	Indicates if IMS issues a WTOR DFS0451A error message and waits for the operator to respond with the ABEND command before terminating after a database error.	
49	2	X	MAXQ	The maximum number of database calls with Qx command codes between synchronization points that can be issued with this PSB.	
51	2	X	LOCKMAX	The maximum number of locks that can application program can obtain at one time with this PSB. The value is in thousands of locks. A value of 0 indicates that there is no limit on the number of locks that an application program can obtain with this PSB.	
53	1	C	CMPAT	Indicates if the PSB is always treated as if it has an I/O PCB even if it is being executed in Batch-DL/I	
54	1	C	OLIC	Indicates if users of this PSB can execute the Online Database Image Copy utility or the Surveyor utility	
55	1	C	GSROLBOK	Indicates whether an internal ROLB call (Y) or a type 777 user abend (N) is issued for non-GSAM databases when the following conditions are true: • The application is a non-message-driven BMP • The PSB contains a GSAM PCB • DB2® for z/OS reports a deadlock either on a thread create or on an SQL call	
56	1	C	DBLEVEL	Requested default DBD version	
57	8	C	FILLER01	Reserved	
65	8	C	CREATEBY	Reserved	
73	16	C	FILLER03	Reserved	

Related concepts:

"Coding program specification blocks as input to the PSBGEN utility" on page 525

Related reference:

 Program Specification Block (PSB) Generation utility (System Utilities)

PSBVEND segment type format

The IMS catalog PSBVEND segment type contains a short header followed by a large block of unformatted space.

This segment type is reserved for use by vendor-supplied tools.

Segment name
PSBVEND

Parent name
PSB

Sequence field
PVNDSEQ

Segment length
4000 bytes

Table 39. PSBVEND segment map

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SEQNUM	Segment sequence number	
7	2	C	FILLER	Reserved	
9	3992	X	DATA	Vendor product PSB data	
5	2	X	PVNDSEQ	Sequence field, type = U	X

PSBRMK segment type format

The IMS catalog PSBRMK segment type contains user-specified comments about an IMS program specification block.

This segment is a direct child of the PSB segment instance that the comments pertain to.

Segment name
PSBRMK

Parent name
PSB

Sequence field
PSBCMSEQ

Segment length
264 bytes

Table 40. PSBRMK segment map.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	PSBCMSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this PSB	

SF segment type format

The IMS catalog SF segment type contains information about a sensitive field definition for a sensitive segment in a program control block (PCB).

Segment name
    SF

Parent name
    SS

Sequence field
    SENFLSEQ

Segment length
    40 bytes

Table 41. SF segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SENFLSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
9	8	C	IMSNAME	The IMS name of the field as defined in the FLD catalog record	
17	2	X	START	The offset of the field from the beginning of the segment as returned in the user I/O area	
19	1	C	REPL	Indicates if the field can be altered on a replace call	
20	21	C	FILLER	Reserved	

SFRMK segment type format

The IMS catalog SFRMK segment type contains user comments for a sensitive field definition.

This segment is a direct child of the SF segment instance that the comments pertain to.

Segment name
    SFRMK

Parent name
    SF

Sequence field
    SENFLSEQ

Segment length
    264 bytes

Table 42. SFRMK segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SENFLSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for the parent SEGM	

SS segment type format

The IMS catalog SS segment type contains information about a sensitive segment definition for a program control block (PCB).

Segment name
   SS

Parent name
   PCB

Sequence field
   SENSGSEQ

Segment length
   328 bytes

Table 43. SS segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment.	
3	2	X	CTL	Control field.	
5	2	X	SENSGSEQ	Sequence field, type = U.	X
5	2	X	SEQNUM	Sequence number.	
9	8	C	IMSNAME	Name of the sensitive segment type.	
17	8	C	PARENT	Name of the direct parent of the sensitive segment type. If the value of this field is 0, this is a sensitive root segment type.	
25	4	C	PROCOPT	The processing options that are valid for use with this sensitive segment.	
29	2	X	IDXCNT	The number of secondary indexes with a valid path to this sensitive segment type.	
31	2	X	SSPTRCNT	The number of subset pointers for this sensitive segment.	
33	256	C	INDICES	A list of up to 32 DBD names of secondary index databases that have valid path to this sensitive segment type.	
289	2	X	SSPNUM01	Order in which this subset pointer was specified in the PSB source.	

Table 43. SS segment type format (continued).

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
291	1	C	SSPSEN01	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
292	1	C	SSPFILL1	Reserved.	
293	2	X	SSPNUM02	Order in which this subset pointer was specified in the PSB source.	
295	1	C	SSPSEN02	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
296	1	C	SSPFILL2	Reserved.	
297	2	X	SSPNUM03	Order in which this subset pointer was specified in the PSB source.	
299	1	C	SSPSEN03	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
300	1	C	SSPFILL3	Reserved.	
301	2	X	SSPNUM04	Order in which this subset pointer was specified in the PSB source.	
303	1	C	SSPSEN04	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
304	1	C	SSPFILL4	Reserved.	
305	2	X	SSPNUM05	Order in which this subset pointer was specified in the PSB source.	
307	1	C	SSPSEN05	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
308	1	C	SSPFILL5	Reserved.	
309	2	X	SSPNUM06	Order in which this subset pointer was specified in the PSB source.	
311	1	C	SSPSEN06	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
312	1	C	SSPFILL6	Reserved.	
313	2	X	SSPNUM07	Order in which this subset pointer was specified in the PSB source.	
315	1	C	SSPSEN07	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
316	1	C	SSPFILL7	Reserved.	
317	2	X	SSPNUM08	Order in which this subset pointer was specified in the PSB source.	
319	1	C	SSPSEN08	Sensitivity type for this subset pointer: read (R) or update (U) sensitivity.	
320	1	C	SSPFILL8	Reserved.	
321	8	C	FILLER	Reserved.	

Related concepts:

"The SENSEG statement" on page 527

Related reference:

SENSEG statement (System Utilities)

SSRMK segment type format

The IMS catalog SSRMK segment type contains user-specified comments about a sensitive segment definition.

This segment is a direct child of the SS segment instance that the comments pertain to.

Segment name
  SSRMK

Parent name
  SS

Sequence field
  SENSGSEQ

Segment length
  264 bytes

Table 44. SSRMK segment type format.

Offset (bytes)	Length (bytes)	Data type	Field name	Description	Unique key field
1	2	X	LEN	Length of this segment	
3	2	X	CTL	Control field	
5	2	X	SENSGSEQ	Sequence field, type = U	X
5	2	X	SEQNUM	Sequence number	
7	2	C	FILLER	Reserved	
9	256	C	REMARKS	User comments for this sensitive segment definition	

Chapter 12. IMS catalog secondary index

The IMS catalog secondary index provides a short processing path for determining which PSBs refer to a specific DBD.

Overview

The IMS application environment can contain an arbitrary number of databases that are each accessed through one or many IMS PSBs. Before an IMS database can be safely changed or removed, the database administrator must know which applications have a dependency on the database. The IMS catalog secondary index provides a fast processing path for determining which PSBs have a dependency on a specific DBD.

The IMS catalog secondary index (DFSCX000) root segment type, DBDPSB, is logically linked to the DBDXREF segment type in the IMS catalog database. The *DFSC* prefix is replaced with the catalog alias prefix, if one is defined to IMS.

The IMSNAME field of the DBDXREF segment type is indexed as the DBD2PSB XDFLD for the secondary index relationship. The IMSNAME field contains the 8-character IMS name for the DBD that is referenced by the PSB described by the catalog record. You can search the DBDXREF segment on the IMSNAME (the DBD name), PSBNAME, or TSVERS fields.

Usage

You can use the IMS catalog secondary index to process the IMS catalog metadata in the following ways:
- Use the DFSCATSX PCB to process the primary catalog database (DFSCD000) with PROCSEQ=DFSCX000. The DBDXREF segment type is the only segment type defined as a sensitive segment in this PCB.
- Use the DFSCATX0 PCB to directly process the catalog secondary index. The secondary index root segment type (DBDPSB) is the only segment type in the secondary index.

Both PCBs are included in the IMS catalog PSBs, DFSCP*xxx*, that are dynamically attached to user PSBs when the IMS catalog is active.

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

 Application programming with the IMS catalog (Application Programming)

Related reference:

"DBDXREF segment type format" on page 98

Chapter 13. IMS catalog activity records

Certain types of activity against the IMS catalog can be logged in z/OS System Management Facility (SMF) as a type-29, subtype-3 record.

System programmers and administrators can use the activity record in SMF to review IMS catalog activities.

The type-29 record is exclusive to IMS, and its subtype-3 record is exclusive to the IMS catalog activity record feature.

Related concepts:

Activity records for the IMS catalog (System Definition)

Part 3. Database types and functions

IMS databases come in two general classes: full-function and Fast Path. Each class includes different types of databases and each database type can have different functions and characteristics.

Related concepts:

Chapter 14. Summary of IMS database types and functions

The following table provides a summary of characteristics, functions, and options of the different types of IMS databases.

Table 45. Summary of database characteristics and options for database types

Characteristic	HSAM	HISAM	HDAM	PHDAM	HIDAM	PHIDAM	DEDB	MSDB
Hierarchical structures	Y	Y	Y	Y	Y	Y	Y	N
Direct access storage	Y	Y	Y	Y	Y	Y	Y	N
Multiple data set groups	N	N	Y	Y	Y	Y	N	N
Logical relationships	N	Y	Y	Y	Y	Y	N	N
Variable-length segments	N	Y	Y	Y	Y	Y	Y	N
Segment Edit/Compression	N	Y	Y	Y	Y	Y	Y	N
Data Capture exit routines	N	Y	Y	Y	Y	Y	Y	N
Field-level sensitivity	Y	Y	Y	Y	Y	Y	N	N
Primary index	N	Y	N	N	Y	Y	N	N
Secondary index	N	Y	Y	Y	Y	Y	Y	N
Logging, recovery, offline reorganization	N	Y	Y	Y	Y	Y	Y	Y
VSAM	N	Y	Y	Y	Y	Y	Y	N/A
OSAM	N	N	Y	Y	Y	Y	N	N/A
QSAM/BSAM	Y	N	N	N	N	N	N	N/A
Boolean operators	Y	Y	Y	Y	Y	Y	Y	N
Command codes	Y	Y	Y	Y	Y	Y	Y	N
Subset pointers	N	N	N	N	N	N	Y	N
Uses main storage	N	N	N	N	N	N	N	Y
High parallelism (field call)	N	N	N	N	N	N	N	Y
Compaction	Y	Y	Y	Y	Y	Y	Y	N
DBRC support	Y	Y	Y	Required	Y	Required[1]	Y	N/A
Partitioning support	N	N	N	Y	N	Y	Y	N
Data sharing	Y	Y	Y	Y	Y	Y	Y	N
Partition sharing	N	N	N	Y	N	Y	Y	N
Block level sharing	Y	Y	Y	Y	Y	Y	Y	N
Area sharing	N/A	N/A	N/A	N/A	N/A	N/A	Y	N/A
Record deactivation	N	N	N	N	N	N	Y	N/A
Database size	med	med	med	lg	med	lg	lg	sml
Online utilities	N	N	N	N	N	N	Y	N
Online reorganization	N	N	N	Y	N	Y	Y	N
Batch	Y	Y	Y	Y	Y	Y	N	N

Table notes:

1. The IMS catalog is a PHIDAM database. Unlike other HALDB databases, the IMS catalog PHIDAM database does not require DBRC support. However, DBRC support is strongly recommended outside of test and development environments.

Related concepts:

Chapter 15. Full-function database types

IMS full-function databases are hierarchical databases that are accessed through DL/I calls. IMS makes it possible for application programs to retrieve, replace, delete, and add segments to IMS databases.

IMS allows you to define twelve database types. Each type has different organization processing characteristics. Except for DEDB and MSDB, all the database types are discussed in this chapter.

Understanding how the database types differ enables you to choose the type that best suits your application's processing requirements.

Each database type has its own access method. The following table shows each database type and its access method:

Table 46. Database types and their access methods

Type of database	Access method
HSAM	Hierarchical Sequential Access Method
HISAM	Hierarchical Indexed Sequential Access Method
SHSAM	Simple Hierarchical Sequential Access Method
SHISAM	Simple Hierarchical Indexed Sequential Access Method
GSAM	Generalized Sequential Access Method
HDAM	Hierarchical Direct Access Method
PHDAM	Partitioned Hierarchical Direct Access Method
HIDAM	Hierarchical Indexed Direct Access Method
PHIDAM	Partitioned Hierarchical Indexed Direct Access Method
PSINDEX	Partitioned Secondary Index Database
DEDB	Data Entry Database (Hierarchical Direct Access)
MSDB	Main Storage Database (Hierarchical Direct Access)

Based on the access method used, the various databases can be classified into two groups: sequential storage and direct storage.

Related concepts:

Chapter 14, "Summary of IMS database types and functions," on page 115

"Data entry databases" on page 197

"Main storage databases (MSDBs)" on page 218

"Design review 3" on page 31

"Design review 4" on page 31

Sequential storage method

HSAM, HISAM, SHSAM, and SHISAM databases use the sequential method of accessing data.

With this method, the hierarchical sequence of segments in the database is maintained by putting segments in storage locations that are physically adjacent to each other. GSAM databases also use the sequential method of accessing data, but no concept of hierarchy, database record, or segment exists in GSAM databases.

Direct storage method

HDAM, PHDAM, HIDAM, DEDB, MSDB, and PHIDAM databases use the direct method of accessing data. With this method, the hierarchical sequence of segments is maintained by putting direct-address pointers in each segment's prefix.

Related concepts:

"Performance considerations overview" on page 119

Databases supported with DBCTL

Database Control (DBCTL) configuration of IMS supports all IMS full-function databases.

The full-function databases supported by DBCTL include:

> HSAM
>
> HISAM
>
> SHSAM
>
> SHISAM
>
> HDAM
>
> PHDAM
>
> HIDAM
>
> PHIDAM
>
> PSINDEX

Databases can be accessed through DBCTL from IMS BMP regions, as well as from independent transaction-management subsystems. Only batch-oriented BMP programs are supported because DBCTL provides no message or transaction support.

CICS online programs can access the same IMS database concurrently; however, an IMS batch program must have exclusive access to the database (if you are not participating in IMS data sharing).

If you have batch jobs that currently access IMS databases through IMS data sharing, you can convert them to run as BMPs directly accessing databases through DBCTL, thereby improving performance. You can additionally convert current batch programs to BMPs to access DEDBs.

Related concepts:

Batch processing online: batch-oriented BMPs (Application Programming)

Related reference:

EXEC parameters for IMS batch message processing regions (System Definition)

Databases supported with DCCTL

The DCCTL configuration of IMS supports several database and dependent region combinations.

The database and dependent region combinations supported by the DCCTL configuration of IMS include:

- GSAM databases for BMP regions
- Db2 for z/OS databases for BMP, MPP, and IFP regions through the External Subsystem attachment facility (ESAF)
- Db2 for z/OS databases for JMP and JBP regions through the DB2 Recoverable Resource Manager Services attachment facility (RRSAF)

Restriction: DCCTL does not support full-function or Fast Path databases.

Related reading: For more information on RRSAF, see *DB2 for z/OS Application Programming and SQL Guide*.

Related concepts:

"GSAM databases" on page 142

➦ External Subsystem Attach Facility (ESAF) (Communications and Connections)

Related tasks:

➦ DB2 Attach Facility (Communications and Connections)

Related reference:

➦ IMS system exit routines (Exit Routines)

Performance considerations overview

The functional and performance characteristics of IMS databases vary from one type of IMS databases to another. You will want to make an informed decision regarding the type of database organizations which will best serve your purposes.

The following lists briefly summarize the performance characteristics of the various full-function database types, highlighting efficiencies and deficiencies of hierarchical sequential, hierarchical direct, and general sequential databases.

General sequential (GSAM)

- Supported by DCCTL
- No hierarchy, database records, segments, or keys
- No DLET or REPL
- ISRT adds records at end of data set
- GN and GU processed in batch or BMP applications only
- Allows IMS symbolic checkpoint calls and restart from checkpoint (except VSAM-loaded databases)
- Good for converting data to IMS and for passing data
- Not accessible from an MPP or JMP region
- Space efficient
- Not time efficient

  VSAM

  - Fixed- or variable-length records are usable

- VSAM ESDS DASD stored
- IMS symbolic checkpoint call allowed
- Restart from checkpoint not allowed

BSAM/QSAM

- Fixed-, variable-, or undefined-length records are usable
- BSAM/QSAM DS tape or DASD stored
- Allows IMS symbolic checkpoint calls and restart from checkpoint

Hierarchical sequential

Segments are linked by physical contiguity

HSAM

- Supported by DBCTL
- Physical sequential access to roots and dependents stored on tape or DASD
- ISRT allowed only when database is loaded
- GU, GN, and GNP allowed
- Database update done by merging databases and writing new database
- QSAM and BSAM accessible
- Space efficient but not time efficient
- Sequential access

HISAM

- Supported by DBCTL
- Hierarchical indexed access to roots
- Sequential access to dependents
- Stored on DASD
- VSAM accessible
- All DL/I calls allowed
- Index is on root segment sequence field
- Good for databases not updated often
- Not space efficient with many updates
- Time efficient with SSA-qualified calls

SHSAM

- Supported by DBCTL
- Simple hierarchical sequential access method to root segments only
- ISRT allowed only when database is loaded
- GU, GN, and GNP allowed
- Database update done by reloaded database
- QSAM and BSAM accessible
- Allows IMS symbolic checkpoint calls and restart from checkpoint (except VSAM-loaded databases)
- Good for converting data to IMS and for passing data
- Not accessible from an MPP or JMP region
- Space efficient

- Not time efficient

SHISAM

- Supported by DBCTL
- Simple hierarchical indexed access to roots only
- Stored on DASD
- VSAM accessible
- All DL/I calls allowed
- Good for converting data to IMS and for passing data
- Not space efficient
- Time efficient

Hierarchical direct

Segments are linked by pointers

HDAM

- Supported by DBCTL
- Hashing access to roots
- Sequential access by secondary index to segments
- All DL/I calls allowed
- Stored on DASD in VSAM ESDS or OSAM data set
- Good for direct access to records
- Hierarchical pointers allowed
    - Hierarchical sequential access to dependent segments
    - Better performance than child and twin pointers
    - Less space required than child and twin pointers
- Child and twin pointers allowed
    - Direct access to pointers
    - More space required by additional index VSAM ESDS database

HIDAM

- Supported by DBCTL
- Indexed access to roots
- Pointer access to dependent segments
- All DL/I calls allowed
- Stored on DASD in VSAM ESDS or OSAM data set
- Good for random and sequential access to records
- Good for random access to segment paths
- Hierarchical pointers allowed
    - Hierarchical sequential access to dependent segments
    - Better performance than child and twin pointers
    - Less space required than child and twin pointers
- Child and twin pointers allowed
    - Direct access to pointers
    - More space required by additional index VSAM ESDS database

HALDB partitioned hierarchical direct
> Segments are linked by pointers. HALDB databases contain one to 1 001 partitions. HALDB databases are the best choice for large databases

PHDAM
- Supported by DBCTL
- Supports up to 1 001 partitions
- Partitions support up to 10 database data sets and one indirect list data set (ILDS)
- Maximum size for OSAM data sets is 4 or 8 GB and for VSAM data sets is 4 GB
- Partitions within the database can be allocated, authorized, processed, reorganized, and recovered independently of the other partitions in the database
- Parallel processing of partitions reduces reorganization times
- Each partition can have a different root addressable area (RAA)
- Indirect pointers are used for logical relationships and secondary indexes, which:
  - Allow for the automatic update, or *self healing*, of indirect pointers after database reorganizations
  - Require an ILDS for each partition
- Hashing access to roots
- Sequential access by secondary index to segments
- All DL/I calls allowed
- Stored on DASD in VSAM ESDS or OSAM data sets
- Good for direct access to records
- Direct pointers are used in logical relationships, and symbolic pointers are not supported
- No hierarchical pointers
- Child and twin pointers allowed
  - Direct access to pointers
  - More space required by additional index VSAM ESDS database

PHIDAM
- Supported by DBCTL
- Supports up to 1 001 partitions
- Partitions support up to 10 database data sets, one primary index data set, and one indirect list data set (ILDS)
- Maximum size for OSAM data sets is 4 or 8 GB and for VSAM data sets is 4 GB
- Partitions within the database can be allocated, authorized, processed, reorganized, and recovered independently of the other partitions in the database
- Parallel processing of partitions reduces reorganization times
- Indirect pointers are used for logical relationships and secondary indexes, which:
  - Allow for the automatic update, or *self healing*, of indirect pointers after database reorganizations
  - Require an ILDS for each partition

- Indexed access to roots
- Primary index is a nonrecoverable database, so database update logs are smaller, even before they are compressed when moved to the SLDS
- Record keys are stored in sequence within each partition; whether the sequence of records is maintained across partitions depends on the method of partition selection used
- Pointer access to dependent segments
- All DL/I calls allowed
- Stored on DASD in a VSAM ESDS or OSAM data set
- Good for random and sequential access to records
- Good for random access to segment paths
- Direct pointers are used in logical relationships and symbolic pointers are not supported
- No hierarchical pointers
- Child and twin pointers allowed
  - Direct access to pointers
  - More space required by additional index VSAM ESDS database

HALDB partitioned secondary index

PSINDEX

- Supported by DBCTL
- Supports up to 1 001 partitions
- Partitions support only a single data set
- Stored on DASD in VSAM KSDS data set
- Maximum size of the VSAM data set is 4 GB
- Do not need to rebuild after reorganizations of the indexed database because of the HALDB self-healing pointer process
- Partitions within the partitioned secondary index (PSINDEX) can be allocated, authorized, processed, reorganized, and recovered independently of the other partitions in the database
- Segments have a larger prefix than non-partitioned secondary indexes to accommodate both a 28-byte extended pointer set (EPS) and the length of the root key of the secondary index target segment
- Does not support shared secondary indexes
- Does not support symbolic pointers
- Requires that the secondary index record segments have unique keys

Related concepts:

Nonrecoverable full-function databases

You can define a full-function database as *nonrecoverable* in the RECON data set by using DBRC commands.

When a full-function database is defined as nonrecoverable, each time the data in the database is updated, IMS logs only the data as it exists before the update. IMS does not log the data as it exists after the update. For this reason, you can backout updates to a nonrecoverable full-function database, but you cannot recover a database by reapplying updates to a prior image copy of the database.

This "before" image of the data from nonrecoverable full-function databases is logged in type X'50' log records.

You can use the NONRECOV keyword on either of the DBRC commands INIT.DB or CHANGE.DB to define a database as nonrecoverable.

Related tasks:

Making databases recoverable or nonrecoverable (Operations and Automation)

Related reference:

INIT.DB command (Commands)

CHANGE.DB command (Commands)

HSAM databases

Hierarchical sequential access method (HSAM) databases use the sequential method of accessing data. All database records and all segments within each database record are physically adjacent in storage.

An HSAM database can be stored on tape or on a direct-access storage device. They are processed using either basic sequential access method (BSAM) or queued sequential access method (QSAM) as the operating system access method. Specify your access method on the PROCOPT= parameter in the PCB. If you specify PROCOPT=GS, QSAM is always used. If you specify PROCOPT=G, BSAM is used.

HSAM data sets are loaded with root segments in ascending key sequence (if keys exist for the root) and dependent segments in hierarchical sequence. You do not need to define a key field in root segments. You must, however, present segments to the load program in the order in which they must be loaded. HSAM data sets use a fixed-length, unblocked record format (RECFM=F), which means that the logical record length is the same as the physical block size.

HSAM databases can only be updated by rewriting them. Delete (DLET) and replace (REPL) calls are not allowed, and insert (ISRT) calls are only allowed when the database is being loaded. Although the field-level sensitivity option can be used with HSAM databases, the following options cannot be used with HSAM databases:

- Multiple data set groups
- Logical relationships
- Secondary indexing
- Variable-length segments
- Segment edit/compression exit routine

- Data Capture exit routines
- Asynchronous data capture
- Logging, recovery, or reorganization

Multiple positioning and multiple PCBs cannot be used in HSAM databases.

When to use HSAM

HSAM is used for applications requiring sequential processing only.

The uses of HSAM are limited because of its processing characteristics. Typically, HSAM is used for low-use files. These are files containing, for example, audit trails, statistical reports or files containing historical or archive data that has been purged from the main database.

How an HSAM record is stored

Segments in an HSAM database are loaded in the order in which you present them to the load program.

You should present all segments within a database record in hierarchical sequence. If a sequence field has been defined for root segments, you should present database records to the load program in ascending root key sequence.

The following figure shows an example HSAM database.
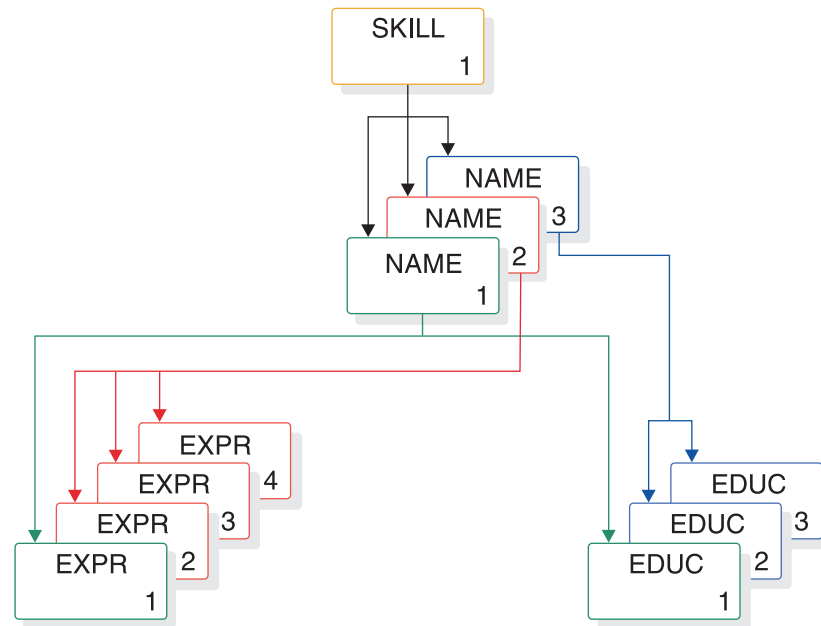
Figure 20. Example HSAM database

The following figure shows how the example HSAM database shown in the preceding figure would be stored in blocks.
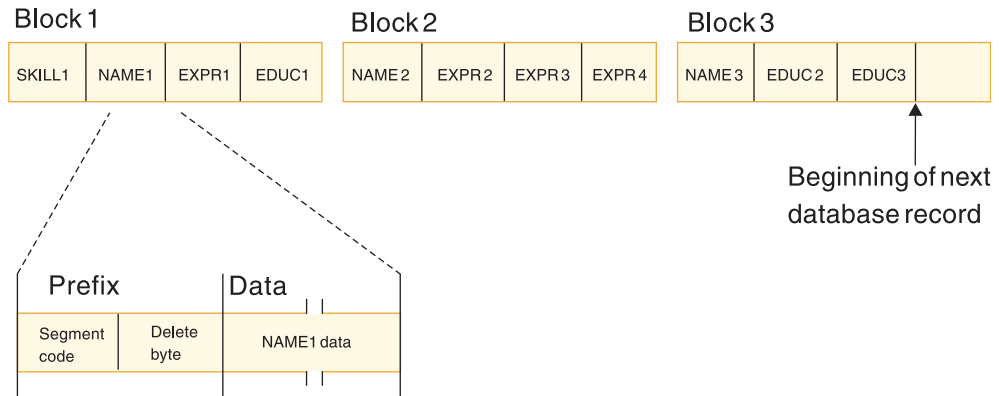
Figure 21. Example HSAM database stored in blocks

In the data set, a database record is stored in one or more consecutive blocks. You define what the block size will be. Each block is filled with segments of the database record until there is not enough space left in the block to store the next segment. When this happens, the remaining space in the block is padded with zeros and the next segment is stored in the next consecutive block. When the last segment of a database record has been stored in a block, any unused space, if sufficient, is filled with segments from the next database record.

In storage, an HSAM segment consists of a 2-byte prefix followed by user data. The first byte of the prefix is the segment code, which identifies the segment *type* to IMS. This number can be from 1 to 255. The segment code is assigned to the segment by IMS in ascending sequence, starting with the root segment and continuing through all dependents in hierarchical sequence. The second byte of the prefix is the delete byte. Because DLET calls cannot be used against an HSAM database, the second byte is not used.

DL/I calls against an HSAM database

Initial entry to an HSAM database is through GU or GN calls. When the first call is issued, the search for the desired segment starts at the beginning of the database and passes sequentially through all segments stored in the database until the desired segment is reached.

After the desired segment is reached, its position is used as the starting position for any additional calls that process the database in a forward direction.

After position in an HSAM database has been established, the way in which GU calls are handled depends on whether a sequence field is defined for the root segment and what processing options are in effect. The following figure shows a flow chart of the actions taken based on whether a sequence field is defined and what processing options are in effect.
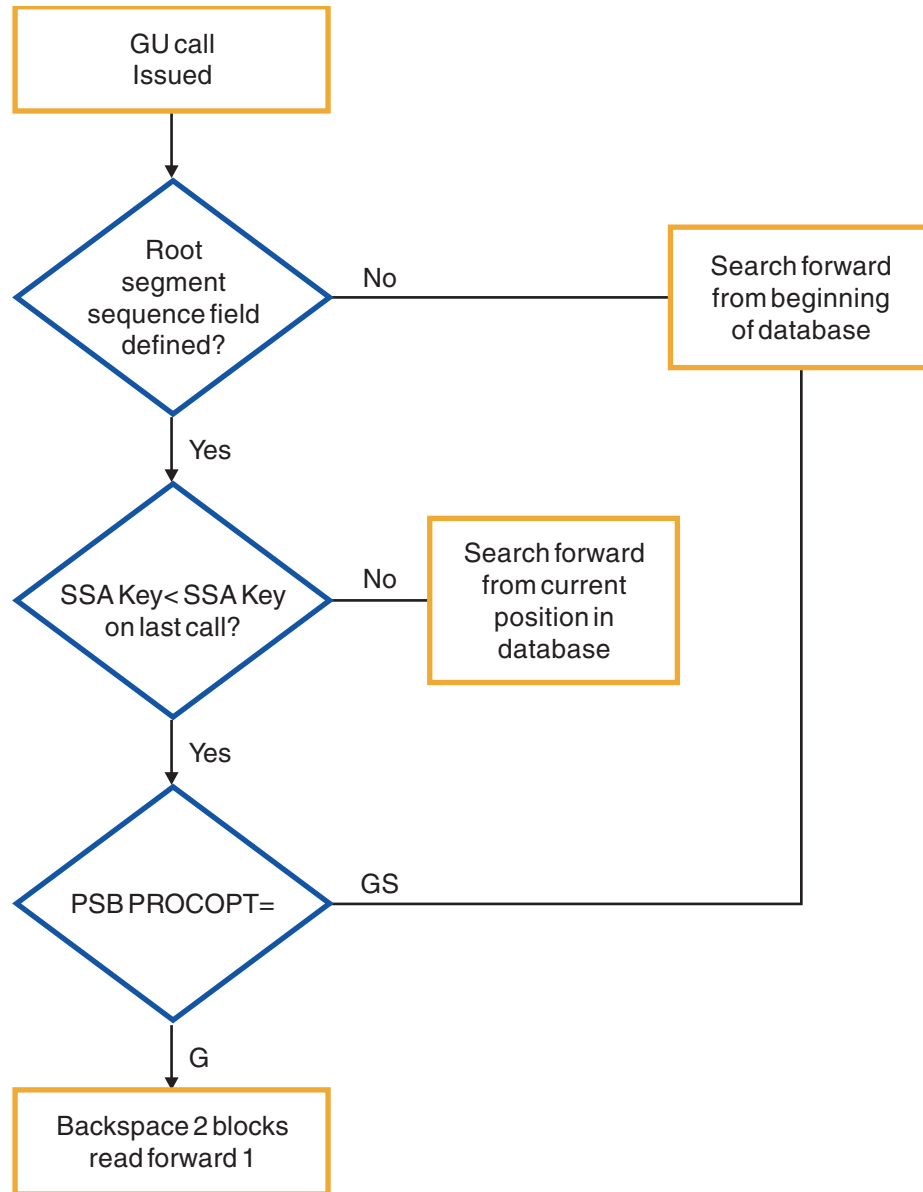
Figure 22. GU calls against an HSAM database

No sequence field defined

If no sequence field has been defined, each GU call causes the search for the desired segment to start at the beginning of the database regardless of current position.

This allows direct processing of the HSAM database. The processing, however, is restricted to one volume.

Sequence field defined

If a sequence field has been defined and the GU call retrieves a segment that is forward in the database, the search starts from the current position and moves forward to the desired segment.

If access to the desired segment requires backward movement in the database, the PROCOPT= parameters G or GS (specified during PSBGEN) determine how backward movement is accomplished. If you specify PROCOPT=GS (that is, the database is read using QSAM), the search for the desired segment starts at the beginning of the database and moves forward. If you specify PROCOPT=G (that is, the database is read using BSAM), the search moves backward in the database. This is accomplished by backspacing over the block just read and the block previous to it, then reading this previous block forward until the wanted segment is found.

Because of the way in which segments are accessed in an HSAM database, it is most practical to access root segments sequentially and dependent segments in hierarchical sequence within a database record. Other methods of access, involving backspacing, rewinding of the tape, or scanning the data set from the beginning, can be time consuming.

As stated previously, DLET and REPL calls cannot be issued against an HSAM database. ISRT calls are allowed only when the database is being loaded. To update an HSAM database, you must write a program that merges the current HSAM database and the update data. The update data can be in one or more files. The output data set created by this process is the new updated HSAM database. The following figure illustrates this process.
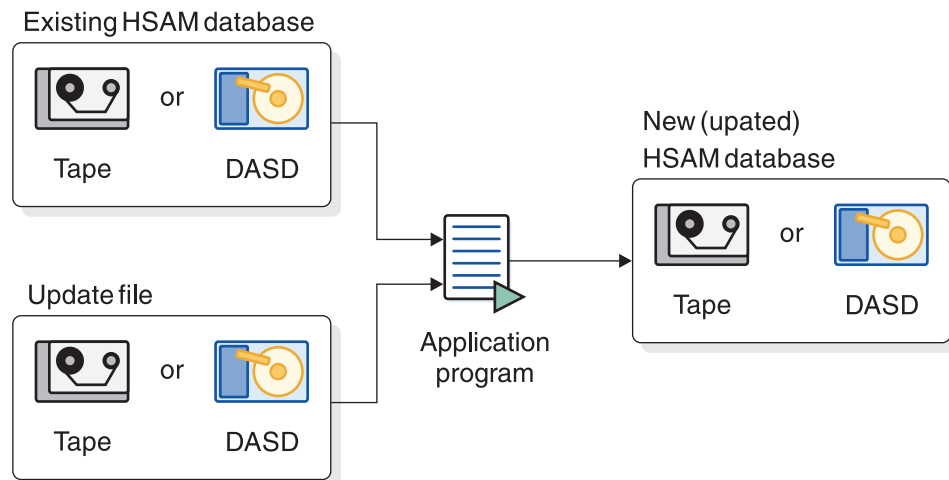
Figure 23. Updating an HSAM database

HISAM databases

In a hierarchical indexed sequential access method (HISAM) database, as with an HSAM database, segments in each database record are related through physical adjacency in storage.

Unlike HSAM, however, each HISAM database record is indexed, allowing direct access to a database record. In defining a HISAM database, you must define a unique sequence field in each root segment. These sequence fields are then used to construct an index to root segments (and therefore database records) in the database.

HISAM databases are stored on direct-access devices. They can be processed using the virtual storage access method (VSAM) utility. Unlike HSAM, all DL/I calls can be issued against a HISAM database. In addition, the following options are available for HISAM databases:

- Logical relationships
- Secondary indexing
- Variable-length segments
- Segment edit/compression exit routine
- Data Capture exit routines
- Field-level sensitivity
- Logging, recovery, and reorganization

Criteria for selecting HISAM

You should use HISAM when you need sequential or direct access to roots and sequential processing of dependent segments in a database record.

HISAM is a good choice of data organization when your database has most, or all, of the following characteristics.

- Each root has few dependents.

  Root segment access is indexed, and is therefore fast. Dependent segment access is sequential, and is therefore slower.

- You have a small number of delete operations against the database.

  Except for deleting root segments, all delete operations result in the creation of space that is unusable until the database is reorganized.

- Your applications depend on a small volume of root segments being inserted within a narrow key range (VSAM).

  Root segments inserted after initial load are inserted in root key sequence in the appropriate CI in the KSDS. If many roots have keys within a narrow key range, many CI splits can occur. This will degrade performance.

- Most of your database records are about the same size.

  The similar sizes allow you to pick logical record lengths and CI sizes so most database records fit on the primary data set. You want most database records to fit on the primary data set, because additional read and seek operations are required to access those parts of a database record on the overflow data set. Additional reads and seeks degrade performance. If, however, most of the processing you do against a database record occurs on segments in the primary data set (in other words, your high-use segments fit on the primary data set), these considerations might not be as important.

  Having most of your database records the same size also saves space. Each database record starts at the beginning of a logical record. All space in the logical records not used by the database record is unusable. This is true of logical records in both the primary and overflow data set. If the size of your database records varies tremendously, large gaps of unused space can occur at the end of many logical records.

How a HISAM record is stored

HISAM database records are stored in two data sets: a primary data set and an overflow data set.

The *primary data set* contains an index and all segments in a database record that can fit in one logical record. The index provides direct access to the root segment

(and therefore to database records). The *overflow data set*, contains all segments in the database record that cannot fit in the primary data set. A key-sequenced data set (KSDS) is the primary data set and an entry-sequenced data set (ESDS) is the overflow data set.

There are several things you need to know about storage of HISAM database records:

- You define the logical record length of both the primary and overflow data set (subject to the rules listed in this topic). The logical record length can be different for each data set. This allows you to define the logical record length in the primary data set as large enough to hold an "average" database record or the most frequently accessed segments in the database record. Logical record length in the overflow data set can then be defined (subject to some restrictions) as whatever is most efficient given the characteristics of your database records.
- Logical records are grouped into control intervals (CIs). A control interval is the unit of data transferred between an I/O device and storage. You define the size of CIs.
- Each database record starts at the beginning of a logical record in the primary data set. A database record can only occupy one logical record in the primary data set, but overflow segments of the database record can occupy more than one logical record in the overflow data set.
- Segments in a database record cannot be split and stored across two logical records. Because of this and because each database record starts a new logical record, unused space exists at the end of many logical records. When the database is initially loaded, IMS inserts a root segment with a key of all X'FF's as the last root segment in the database.

The following figure shows four HISAM database records.
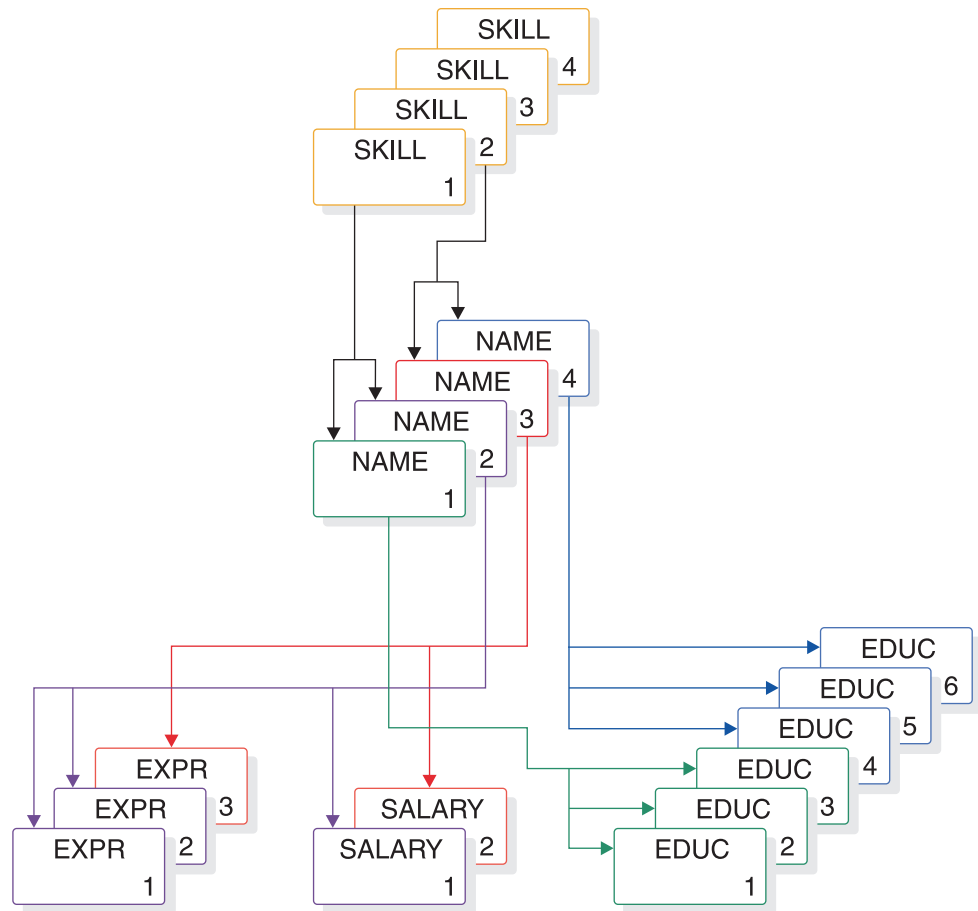
Figure 24. Example HISAM database records

The following figure shows the four records from the preceding figure as they are initially stored on the primary and overflow data sets. In storage, a HISAM segment consists of a 2-byte prefix followed by user data. The first byte of the prefix is the segment code, which identifies the segment *type* to IMS. This number can be from 1 to 255. The segment code is assigned to the segment by IMS in ascending sequence, starting with the root segment and continuing through all dependents in hierarchical sequence. The second byte of the prefix is the delete byte.
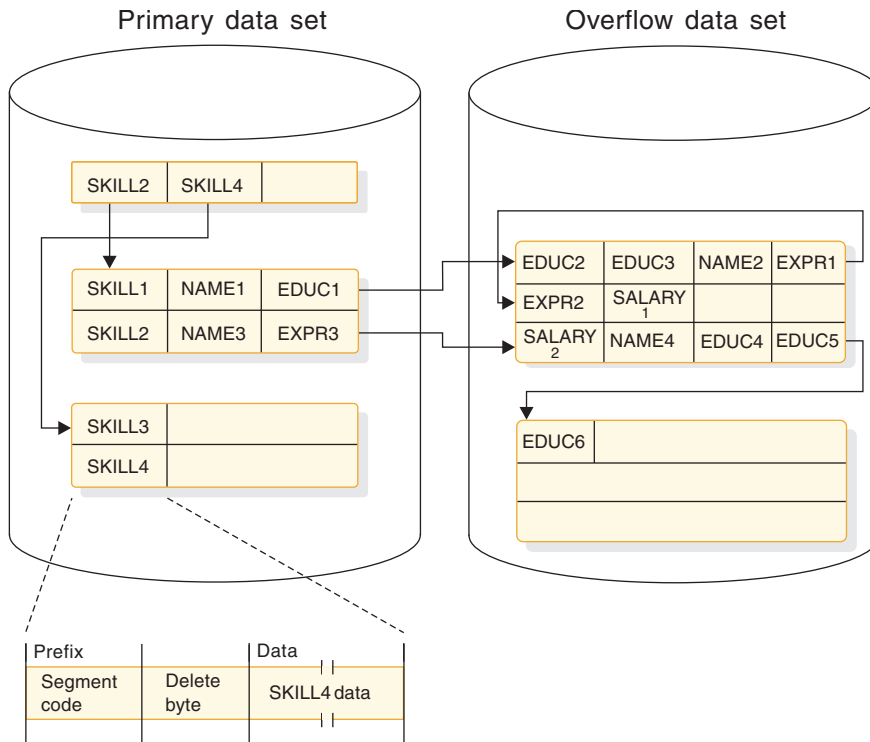
Figure 25. Example HISAM database records in storage

Each logical record in the primary data set contains the root plus all dependents of the root (in hierarchical sequence) for which there is enough space. The remaining segments of the database record are put in the overflow data set (again in hierarchical sequence). The two "parts" of the database record are chained together with a direct-address pointer. When overflow segments in a database record use more than one logical record in the overflow data set, as is the case for the first and second database records in the preceding figure, the logical records are also chained together with a direct-address pointer. Note in the figure that HISAM indexes do not contain a pointer to each root segment in the database. Rather, they point to the highest root key in each block or CI.

The following figure illustrates the following points regarding the structure of a logical record in a HISAM database:

- In a logical record, the first 4 bytes are a direct-address pointer to the next logical record in the database record. This pointer maintains all logical records in a database record in correct sequence. The last logical record in a database record contains zeros in this field.
- Following the pointer are one or more segments of the database record in hierarchical sequence.
- Following the segments is a 1-byte segment code of 0. It says that the last segment in the logical record has been reached.

RBA	Segment		Segment	Segment code of 0	Unused space
Bytes 4		Varies		1	Varies

Figure 26. Format of a logical record in a HISAM database

Accessing segments

When accessing a segment in a HISAM database, the application program follows a set search sequence.

In HISAM, when an application program issues a call with a segment search argument (SSA) qualified on the key of the root segment, the segment is found by:

1. Searching the index for the first pointer with a value greater than or equal to the specified root key (the index points to the highest root key in each CI)
2. Following the index pointer to the correct CI
3. Searching this CI for the correct logical record (the root key value is compared with each root key in the CI)
4. When the correct logical record (and therefore database record) is found, searching sequentially through it for the specified segment

If an application program issues a GU call with an unqualified SSA for a root segment or with an SSA qualified on other than the root key, the HISAM index cannot be used. The search for the segment starts at the beginning of the database and proceeds sequentially until the specified segment is found.

Inserting root segments using VSAM

After an initial load, root segments inserted into a HISAM database are stored in the primary data set in ascending key sequence.

The CI might or might not contain a free logical record into which the new root can be inserted. Both situations are described next.

A free logical record exists

This example shows how insertion takes place when a free logical record exists.

In the following figure, the new root is inserted into the CI in root key sequence. If there are logical records in the CI containing roots with higher keys, they are "pushed down" to create space for the new logical record.
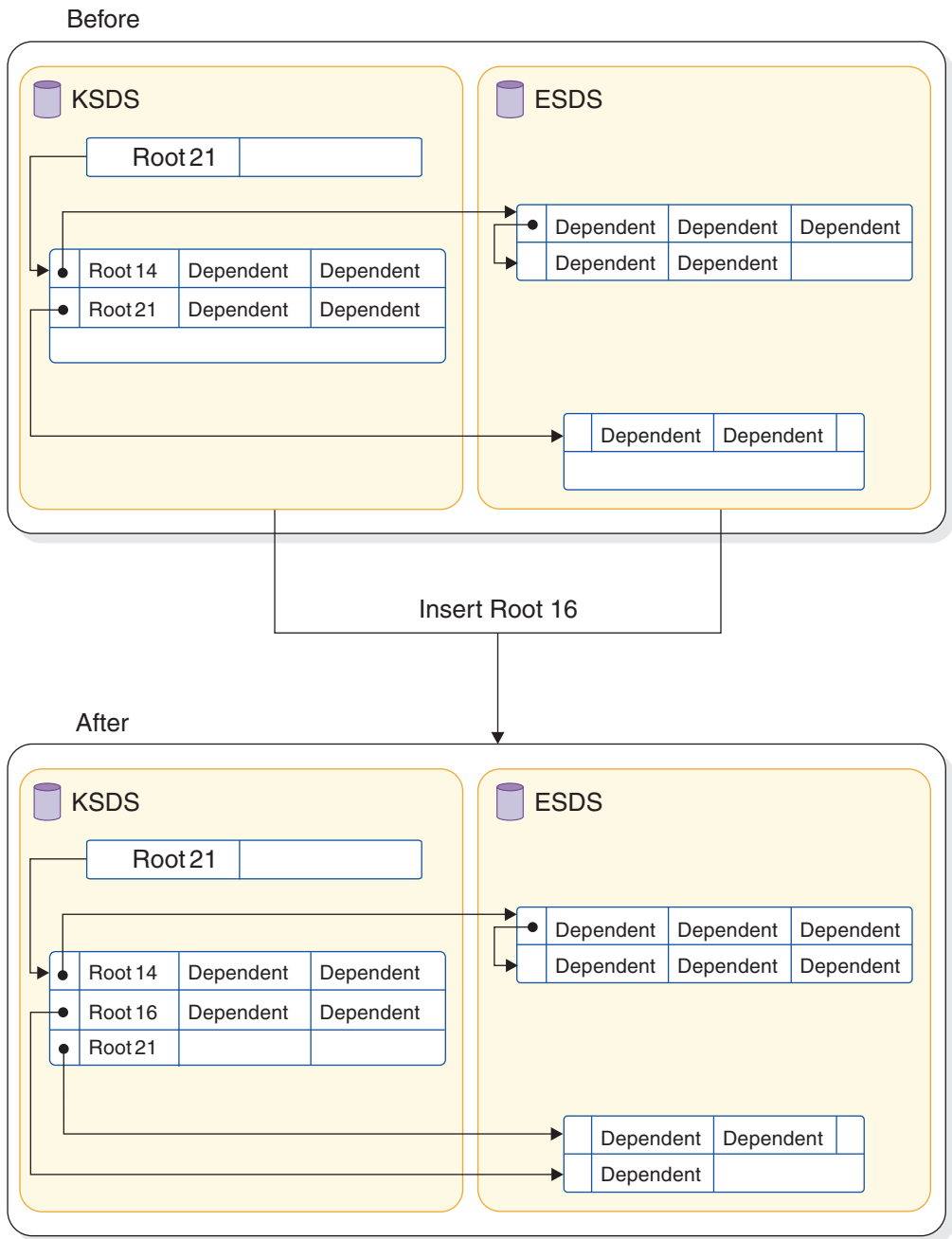
Figure 27. Inserting a root segment into a HISAM database (free logical record exists in the CI)

No free logical record exists

This example shows how insertion takes place when no free logical record exists in the CI.

The CI is split forming two new CIs, both equal in size to the original one. Where the CI is split depends on what you have coded in the INSERT=parameter on the OPTIONS statement in the DFSVSAMP data set for batch environments or the DFSVSMxx PROCLIB member for online environments.

The split can occur at the point at which the root is inserted or midpoint in the CI. After the CI is split, free logical records exist in each new CI and the new root is inserted into the proper CI in root key sequence. If, as was the case in the figure shown in "A free logical record exists" on page 133, logical records in the new CI contained roots with higher keys, those logical records would be "pushed down" to create space for the new logical record.

When adding new root segments to a HISAM database, performance can be slightly improved if roots are added in ascending key sequence.
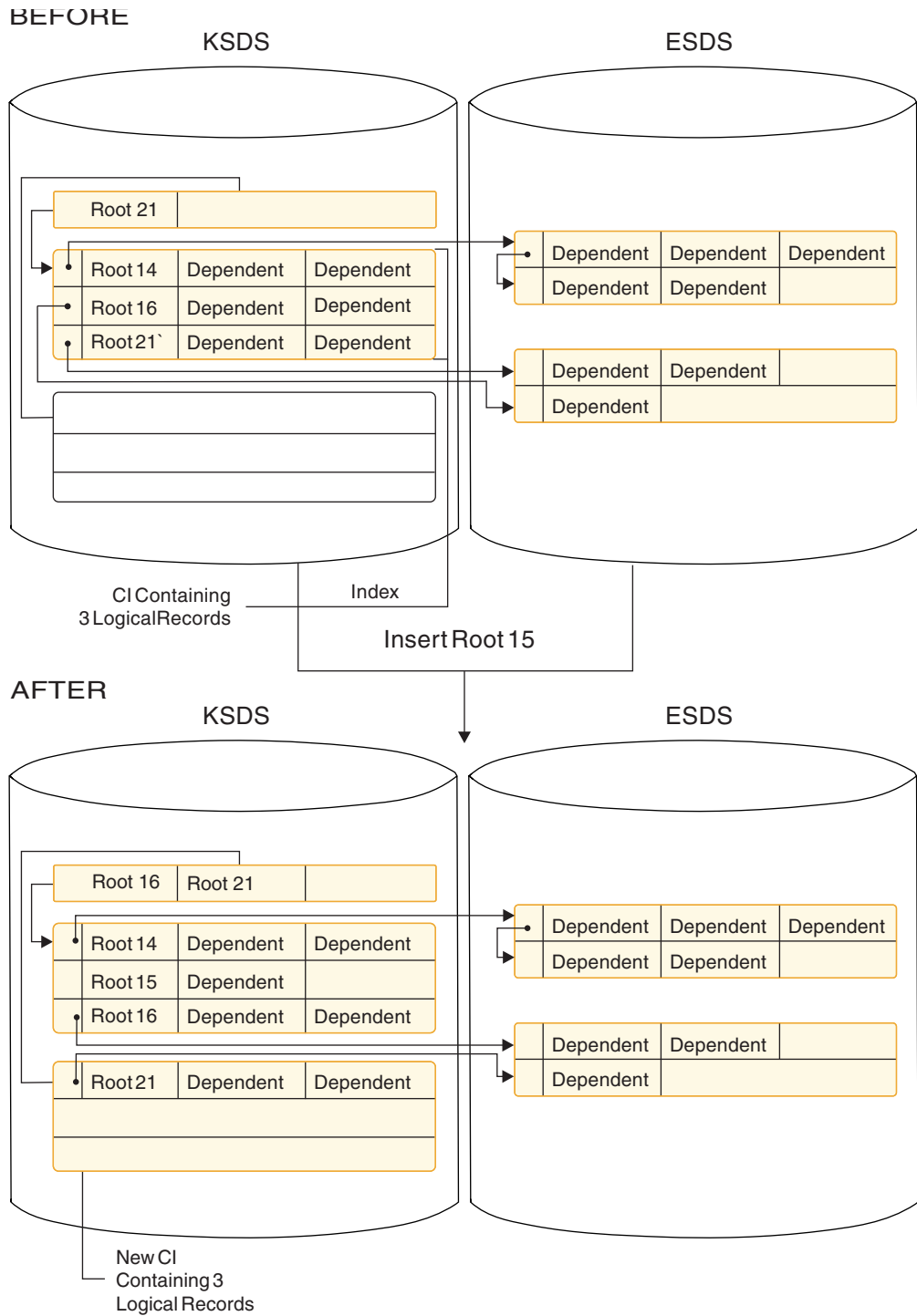
Figure 28. Inserting a root segment into a HISAM database (no free logical record exists in the CI)

Related reference:

➡ DFSVSMxx member of the IMS PROCLIB data set (System Definition)

➡ DD statements for IMS procedures (System Definition)

Inserting dependent segments

Dependent segments inserted into a HISAM database after initial load are inserted in hierarchical sequence. IMS decides where in the appropriate logical record the new dependent should be inserted.

Two situations are possible. Either there is enough space in the logical record for the new dependent or there is not.

The following figure shows how segment insertion takes place when there is enough space in the logical record. The new dependent is stored in its proper hierarchical position in the logical record by shifting the segments that hierarchically follow it to the right in the logical record.
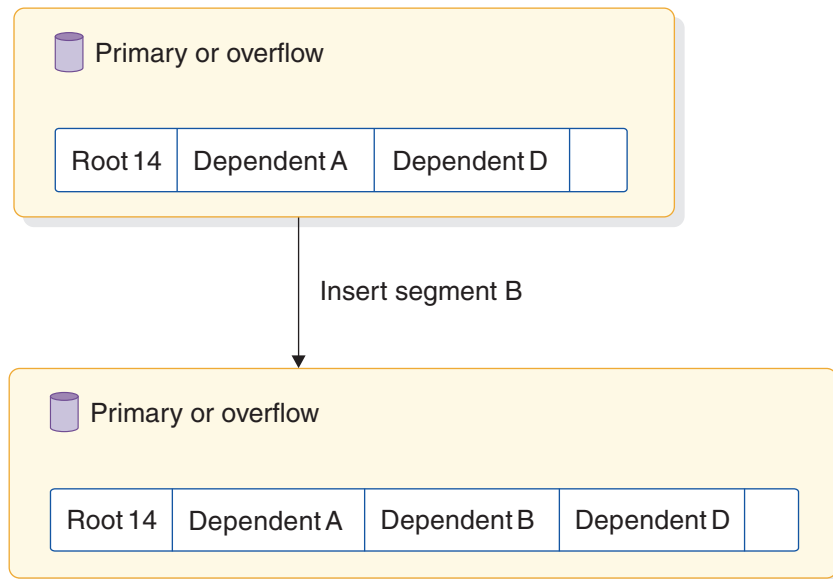
Figure 29. Inserting a dependent segment into a HISAM database (space exists in the logical record)

The following figure shows how segment insertion takes place when there is not enough space in the logical record. As in the previous case, new dependents are always stored in their proper hierarchical sequence in the logical record. However, all segments to the right of the new segment are moved to the first empty logical record in the overflow data set.

Before

CI or block containing two logical records

Primary

| Root 14 | Dependent A | Dependent B | Dependent D | • |
| Root 16 | Dependent A | Dependent B | Dependent D | • |

Overflow

| | Dependent I | Dependent H | Dependent F |
| • | Dependent M | Dependent K | Dependent J |

| Dependent X | |
| | |

Insert dependent segment G
for root 14

After

Primary

| Root 14 | Dependent A | Dependent B | Dependent D | • |
| Root 16 | Dependent A | Dependent B | Dependent D | • |

Overflow

| • | | Dependent G | Dependent F |
| • | Dependent M | Dependent K | Dependent J |

| Dependent X | |
| Dependent H | Dependent I | |

Figure 30. Inserting a dependent segment into a HISAM database (no space exists in the logical record)

Deleting segments

When segments are deleted from a HISAM database, they are marked as deleted in the delete byte in their prefix. They are not physically removed from the database; the one exception to this is discussed later in this topic.

Dependent segments of the deleted segment are not marked as deleted, but because their parent is, the dependent segments cannot be accessed. These unmarked segments (as well as segments marked as deleted) are deleted when the database is reorganized.

One thing you should note is that when a segment is accessed that hierarchically follows deleted segments in a database record, the deleted segments must still be "searched through". This concept is shown in the following figures.

Segment B2 is deleted from this database record. This means that segment B2 and its dependents (C1, C2, and C3) can no longer be accessed, even though they still exist in the database.
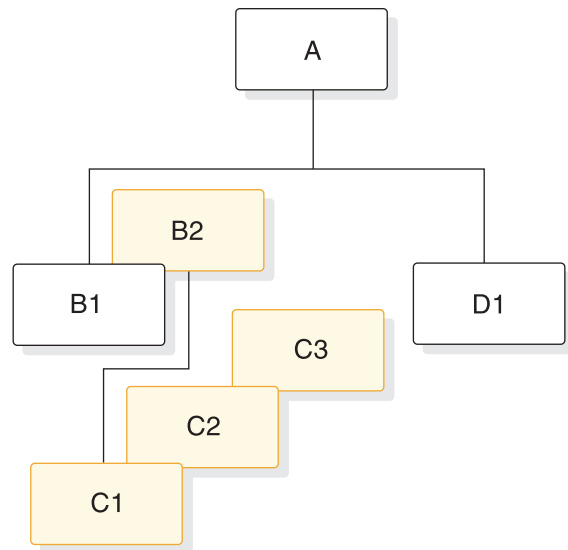
Figure 31. The hierarchical segment layout on the database

A request to access segment D1 is made. Although segments B2, C1, C2, and C3 cannot be accessed, they still exist in the database. Therefore they must still be "searched through" even though they are inaccessible as shown in the following figure.
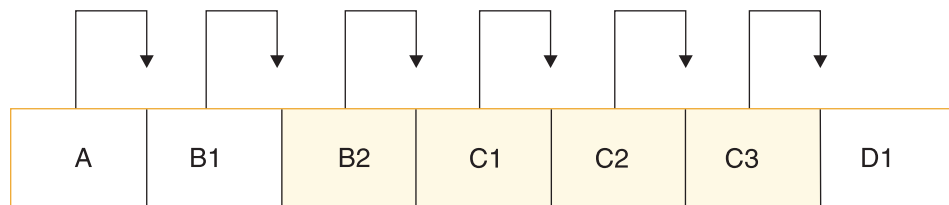
Figure 32. Accessing a HISAM segment that hierarchically follows deleted segments

In one situation, deleted segments are physically removed from the database. If the deleted segment is a root, the logical record containing the root is erased, provided neither the root nor any of its dependents is involved in a logical relationship. The default is ERASE=YES, and no "mark buffer altered" takes place. Thus a PROCOPT=G read job will not have to wait for locks after another job has set the delete byte, and will return a segment not found condition. To be consistent with

other DB types, use ERASE=NO to cause a wait for physical delete prior to attempted read. The ERASE parameter is specified on the DBD statement of the DFSVSMxx PROCLIB member.

After the logical record is removed, its space is available for reuse. However, any overflow logical record containing dependents of this root is not available for reuse. Except for this special condition, you must unload and reload a HISAM database to regain space occupied by deleted segments.

Related concepts:

"Locking to provide program isolation" on page 179

Replacing segments

Replacing segments in a HISAM database is straightforward as long as fixed length segments are being used. The data in the segment, once changed, is returned to its original location in storage. The key field in a segment cannot be changed.

When variable-length segments are used there are other implications to consider.

Related concepts:

"Variable-length segments" on page 387

SHSAM, SHISAM, and GSAM databases

You typically use simple hierarchical sequential access method (SHSAM), simple hierarchical indexed sequential access method (SHISAM), and generalized sequential access method (GSAM) databases either when converting a non-database system to IMS or when passing data from one application program to another.

When converting from a non-database system to IMS, SHSAM, SHISAM, and GSAM databases allow existing programs, using z/OS access methods, to remain usable during the conversion to IMS. This is possible because the format of the data in these databases is the same as in the z/OS data sets.

When a database (or non-database) application program passes data to a database (or non-database) application program, it first puts the data in a SHSAM, SHISAM, or GSAM database. The database (or non-database) application program then accesses the data from these databases.

If you have application programs that need access to both IMS and z/OS data sets, you can use SHSAM, SHISAM, or GSAM. Which one you use depends on what functions you need. The following table compares the characteristics and functions available for each of the three types of databases.

Table 47. Comparison of SHSAM, SHISAM, and GSAM databases

Characteristics and functions	SHSAM	SHISAM	GSAM
Uses hierarchical structure	NO	NO	NO
Uses segment prefixes	NO	NO	NO
Supports variable-length records	NO	NO	YES
Supports checkpoint/restart	NO	YES[1]	YES[1]
Compatible with non-IMS data sets	YES	YES	YES

Table 47. Comparison of SHSAM, SHISAM, and GSAM databases (continued)

Characteristics and functions	SHSAM	SHISAM	GSAM
Supports VSAM as the operating system access method	NO	YES	YES
Supports BSAM as the operating system access method	YES	NO	YES
Accessible from a batch region	YES	YES	YES
Accessible from a batch message processing region	YES	YES	YES
Accessible from a message processing region	YES	YES	NO
Supports logging support	NO	YES	NO
Supports GET calls	YES	YES	YES
Supports ISRT calls	YES[2]	YES	YES[3]
Supports CICS-DBCTL	YES	YES	NO
Supports DCCTL	NO	NO	YES

Note:

1. Using symbolic checkpoints
2. To load database only
3. Allowed only at the end of the data set

Related concepts:

"The segment" on page 15

SHSAM databases

A simple HSAM (SHSAM) database is an HSAM database containing only one type of segment, a root segment. The segment has no prefix, because no need exists for a segment code (there is only one segment type) or for a delete byte (deletes are not allowed).

SHSAM databases can be accessed by z/OS BSAM and QSAM because SHSAM segments contain user data only (no IMS prefixes). The ISRT, DLET, and REPL calls cannot be used to update. However, ISRT can be used to load an SHSAM database. Only GET calls are valid for processing an SHSAM database. These allow retrieval only of segments from the database. To update an SHSAM database, it must be reloaded. The situations in which SHSAM is typically used are explained in the introduction to this topic. Before deciding to use SHSAM, read the topic on GSAM databases, because GSAM has many of the same functions as SHSAM. Unlike SHSAM, however, GSAM files cannot be accessed from a message processing region. GSAM does allow you to take checkpoints and perform restart, though.

Although SHSAM databases can use the field-level sensitivity option, they *cannot* use any of the following options:
- Logical relationships
- Secondary indexing
- Multiple data set groups
- Variable-length segments
- Segment edit/compression exit routine
- Data Capture exit routines

- Logging, recovery, or reorganization

SHISAM databases

A simple HISAM (SHISAM) database is a HISAM database containing only one type of segment, a root segment.

The segment has no prefix, because no need exists for a segment code (there is only one segment type) or for a delete byte (deletes are done using a VSAM erase operation). SHISAM databases must be KSDSs; they are accessed through VSAM. Because SHISAM segments contain user data only (no IMS prefixes), they can be accessed by VSAM macros and DL/I calls. All the DL/I calls can be issued against SHISAM databases.

SHISAM IMS symbolic checkpoint call

SHISAM is also useful if you need an application program that accesses z/OS data sets to use the IMS symbolic checkpoint call.

The IMS symbolic checkpoint call makes restart easier than the z/OS basic checkpoint call. If the z/OS data set the application program is using is converted to a SHISAM database data set, the symbolic checkpoint call can be used. This allows application programs to take checkpoints during processing and then restart their programs from a checkpoint. The primary advantage of this is that, if the system fails, application programs can recover from a checkpoint rather than lose all processing that has been done. One exception applies to this: An application program for initially loading a database that uses VSAM as the operating system access method cannot be restarted from a checkpoint. Application programs using GSAM databases can also issue symbolic checkpoint calls. Application programs using SHSAM databases cannot.

Before deciding to use SHISAM, you should read the next topic on GSAM databases. GSAM has many of the same functions as SHISAM. Unlike SHISAM, however, GSAM files cannot be accessed from a message processing region.

SHISAM databases can use field-level sensitivity and Data Capture exit routines, but they *cannot* use any of the following options:
- Logical relationships
- Secondary indexing
- Multiple data set groups
- Variable-length segments
- Segment edit/compression exit routine

GSAM databases

GSAM databases are sequentially organized databases that are designed to be compatible with z/OS data sets.

GSAM databases have no hierarchy, database records, physical segments, or keys. GSAM databases can be in a data set previously created or in one later accessed by the z/OS access methods VSAM or QSAM/BSAM. GSAM data sets can use fixed-length or variable-length records when VSAM is used, or fixed-length, variable-length, or undefined-length records when QSAM/BSAM is used.

Optionally, you can define a single *virtual segment* with multiple non-searchable fields in a GSAM DBD to add field metadata from COBOL copybooks to the GSAM DBD record in the IMS catalog. IMS ignores the virtual segment and the

non-searchable fields, but application programs and products, such as IMS Enterprise Suite Explorer for Development, can retrieve the metadata from the IMS catalog when they access the GSAM database through the IMS Universal drivers.

If VSAM is used to process a GSAM database, the VSAM data set must be entry sequenced and on a DASD. If QSAM/BSAM is used, the physical sequential (DSORG=PS) data set can be placed on a DASD or tape unit. If BSAM is used, the GSAM data sets can be defined as z/OS large format data sets by specifying DSNTYPE=LARGE on the DD statements.

GSAM supports DFSMS striped extended-format data sets for both VSAM and BSAM.

GSAM database data sets can be allocated in the extended addressing space (EAS) of an extended address volume (EAV).

Restriction: GSAM databases cannot be used with CICS applications.

Because GSAM databases are supported in a DCCTL environment, you can use them when you need to process sequential non-IMS data sets using a BMP program.

GSAM databases are loaded in the order in which you present records to the load program. You cannot issue DLET and REPL calls against GSAM databases; however, you can issue ISRT calls after the database is loaded but only to add records to the end of the data set. Records are not randomly added to a GSAM data set.

Although random processing of GSAM and SHSAM databases is possible, random processing of a GSAM database is done using a GU call qualified with a record search argument (RSA). This processing is primarily useful for establishing position in the database before issuing a series of GN calls.

Although SHSAM and SHISAM databases can be processed in any processing region, GSAM databases can only be processed in a batch or batch message processing region.

The following IMS options do not apply to GSAM databases:
- Logical relationships
- Secondary indexing
- Segment edit/compression exit routine
- Field-level sensitivity
- Data Capture exit routines
- Logging or reorganization
- Multiple data set groups

For more information about GSAM data sets and access methods, including information about the GSAM use of striped extended-format data sets, see "Processing GSAM databases" in *IMS Version 15 Application Programming*.

For more information about z/OS data sets, see *z/OS DFSMS: Using Data Sets*, as well as the z/OS DFSMShsm, DFSMSdss, and DFSMSdfp storage administration guides and references.

Related concepts:

GSAM IMS symbolic checkpoint call

Among its other uses, GSAM is also useful if you need an application program that accesses z/OS data sets to use the IMS symbolic checkpoint call.

The IMS symbolic checkpoint call makes restart easier than the z/OS basic checkpoint call. This IMS symbolic checkpoint call allows application programs to take checkpoints during processing, thereby allowing programs to restart from a checkpoint. A checkpoint call forces any GSAM buffers with inserted records to be written as short blocks. The primary advantage of taking checkpoints is that, if the system fails, the application programs can recover from a checkpoint rather than lose all your processed data. However, any application program that uses VSAM as an operating system access method and initially loads the database cannot be restarted from a checkpoint.

In general, always use DISP=OLD for GSAM data sets when restarting from a checkpoint even if you used DISP=MOD on the original execution of the job step. If you use DISP=OLD, the data set is positioned at its beginning. If you use DISP=MOD, the data set is positioned at its end.

HDAM, PHDAM, HIDAM, and PHIDAM databases

A hierarchical direct (HD) database is a database that maintains the hierarchical sequence of its segments by having segments point to one another (instead of by physically storing the segments in the hierarchical sequence).

HD databases are stored on direct-access devices in either a VSAM ESDS or an OSAM data set.

In most cases, each segment in an HD database has one or more direct-address pointers in its prefix. When direct-address pointers are used, database records and segments can be stored anywhere in the database. After segments are inserted into the database, they remain in their original positions unless the segments are deleted or until the database is reorganized. During database update activity, pointers are updated to reflect the hierarchical relationships of the segments.

HD databases also differ from sequentially organized databases because space in HD databases can be reused. If part or all of a database record is deleted, the deleted space can be reused when new database records or segments are inserted.

HD databases access the root segments that they contain in one of two ways: by using a randomizing module or by using a primary index. HD databases that use a randomizing module are referred to as *hierarchical direct access method (HDAM)* databases. HD databases that use a primary index are referred to as *hierarchical indexed direct access method (HIDAM)* databases.

HD databases can also be partitioned. A partitioned HD database that uses a randomizing module to access its root segments is referred to as a *partitioned HDAM (PHDAM)* database. A partitioned HD database that uses a primary index to access its root segments is referred to as a *partitioned HIDAM (PHIDAM)* database. PHDAM and PHIDAM databases, along with *partitioned secondary index (PSINDEX)* databases, are collectively referred to as *High Availability Large Database (HALDB)* type databases.

The storage organization in HD databases that use a randomizing module and in HD databases that use a primary index is basically the same. The primary difference is in how their root segments are accessed. In HDAM or PHDAM databases, the randomizing module examines the root's key to determine the address of a pointer to the root segment. In HIDAM or PHIDAM databases, each root segment's storage location is found by searching the index. In HIDAM databases, the primary index is a database that IMS loads and maintains. In PHIDAM databases, the primary index is a data set that IMS loads and maintains. The advantage of a randomizing module is that the I/O operations that are required to search an index are eliminated.

In PHDAM and PHIDAM databases, before IMS uses either the randomizing module or the primary index, IMS must determine which partition the root segments are stored in by using a process called *partition selection*. You can have IMS perform partition selection by assigning a range of root keys to a partition or by using a partition selection exit routine.

The following figure compares a logical view of an HDAM database with the logical view of a PHDAM database.
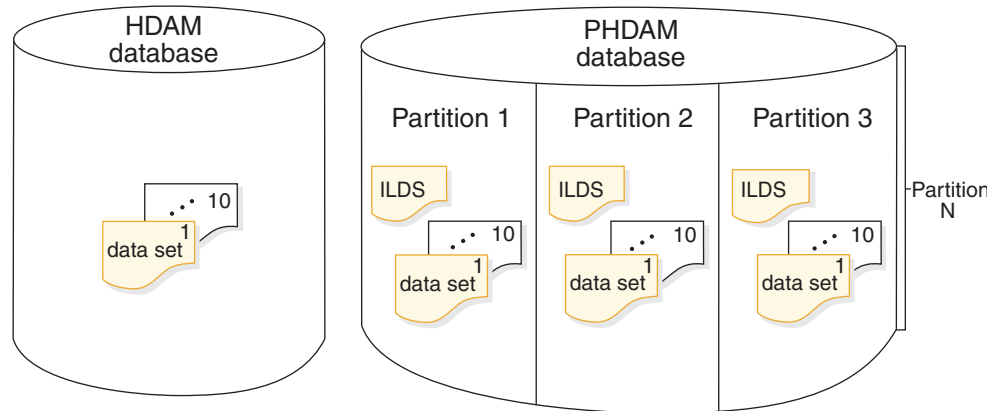
Figure 33. A comparison of the logical views of HDAM and PHDAM databases

The following figure compares a logical view of a HIDAM database with the logical view of a PHIDAM database.
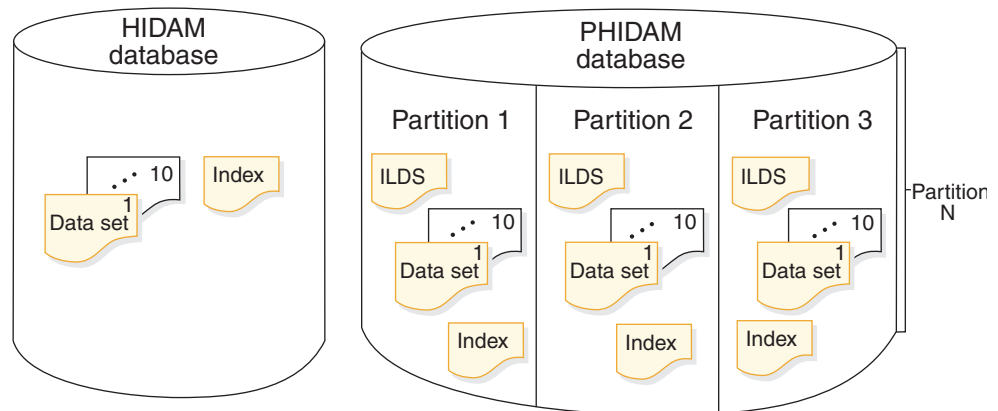
Figure 34. A comparison of the logical views of HIDAM and PHIDAM databases

Related concepts:

Maximum sizes of HD databases

The maximum possible size of HDAM, PHDAM, HIDAM, and PHIDAM databases is based on the number of data sets the database can hold and the size of the data sets. The maximum possible size of a data set differs depending on whether VSAM or OSAM is used and whether the database is partitioned.

The following table lists the maximum data set size, maximum number of data sets, and maximum database size for HDAM, PHDAM, HIDAM, and PHIDAM databases.

Table 48. Maximum sizes for HDAM, HIDAM, PHDAM, and PHIDAM databases

Data set type	Maximum data set size	Maximum number of data sets	Maximum database size
OSAM HDAM or HIDAM Database	8 GB	10 data sets	80 GB
VSAM HDAM or HIDAM Database	4 GB	10 data sets	40 GB
OSAM PHDAM or PHIDAM Database	4 or 8 GB[1]	10 010 data sets (10 data sets per partition; 1001 partitions per database)	40 040 or 80 080 GB
VSAM PHDAM or PHIDAM Database	4 GB	10 010 data sets (10 data sets per partition; 1001 partitions per database)	40 040 GB

Note:

1. The maximum size depends on how the HALDB is registered with DBRC. By default, the maximum size of the OSAM data sets is 4 GB.

Related concepts:

"Using OSAM as the access method" on page 567

DL/I calls that can be issued against HD databases

All DL/I calls can be issued against HD databases.

In addition, the following options are available:
- Multiple data set groups
- Logical relationships
- Secondary indexing
- Variable-length segments
- Segment edit/compression exit routine
- Data Capture exit routines
- Field-level sensitivity
- Logging, recovery, and offline reorganization
- Online reorganization for HALDB partitions

Related concepts:

➡ Logging (System Administration)

When to use HDAM and PHDAM

HDAM and PHDAM databases are typically used for direct access to database records.

The randomizing module provides fast access to the root segment (and therefore the database record). HDAM and PHDAM databases also give you fast access to paths of segments as specified in the DBD in a database record. For example, in the following figure, if physical child pointers are used, they can be followed to reach segments B, C, D, or E. A hierarchical search of segments in the database record is bypassed. Segment B does not need to be accessed to get to segments C, D, or E. And segment D does not need to be accessed to get to segment E. Only segment A must be accessed to get to segment B or C. And only segments A and C must be accessed to get to segments D or E.
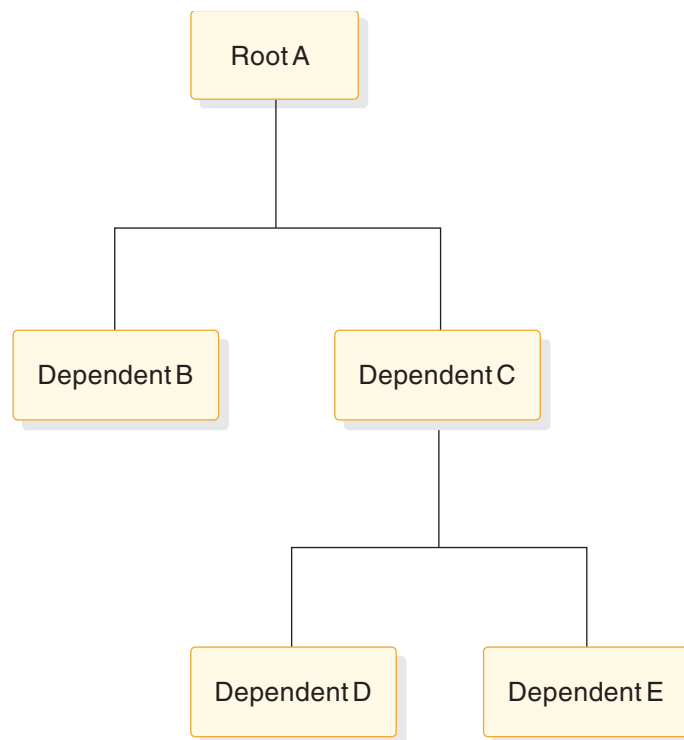
Figure 35. Example database record

When to use HIDAM and PHIDAM

HIDAM and PHIDAM databases are typically used when you need both random and sequential access to database records and random access to paths of segment in a database record.

Access to root segments (and therefore database records) is not as fast as with HDAM (or PHDAM), because the HIDAM (or PHIDAM) index database has to be searched for a root segment's address. However, because the index keeps the address of root segments stored in key sequence, database records can be processed sequentially.

Pointers in HD databases

Before looking in detail at how HD databases are stored and processed, you need to become familiar with the pointers that are used in HD databases.

Types of pointers you can specify

In the HD access methods, segments in a database record are kept in hierarchical sequence using direct-address pointers.

Except for a few special cases, each prefix in an HD segment contains one or more pointers. Each pointer is 4 bytes long and consists of the relative byte address of the segment to which it points. Relative, in this case, means relative to the beginning of the data set.

Several different types of direct-address pointers exist, and you will see how each works in the topics that follow in this section. However, there are three basic types:

- Hierarchical pointers, which point from one segment to the next in either forward or forward and backward hierarchical sequence
- Physical child pointers, which point from a parent to each of its first or first and last children, for each child segment type
- Physical twin pointers, which point forward or forward and backward from one segment occurrence of a segment type to the next, under the same parent

When segments in a database record are typically processed in hierarchical sequence, use hierarchical pointers. When segments in a database record are typically processed randomly, use a combination of physical child and physical twin pointers. One thing to keep in mind while reading about pointers is that the different types, subject to some rules, can be mixed within a database record. However, because pointers are specified by segment type, all occurrences of the same segment type have the same type of pointer.

Each type of pointer is examined separately in this topic. In the subtopics in this topic, each type of pointer is illustrated, and the database record on which each illustration is based is shown in the following figure.
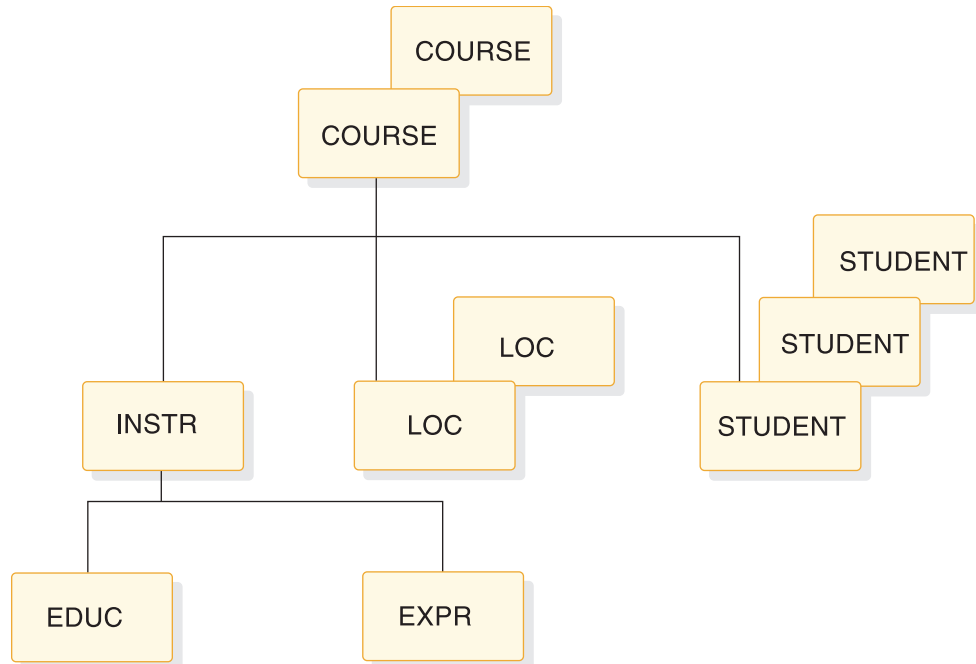
Figure 36. Example database record for illustrating pointers

Related concepts:

"Mixing pointers" on page 157

Chapter 32, "Converting database types," on page 841

"Physical child first pointers" on page 152

"Physical child first and last pointers" on page 153

Related tasks:

"Converting a database from HIDAM to HDAM" on page 844

Hierarchical forward pointers

With hierarchical forward (HF) pointers, each segment in a database record points to the segment that follows it in the hierarchy.

The following figure shows hierarchical forward pointers:
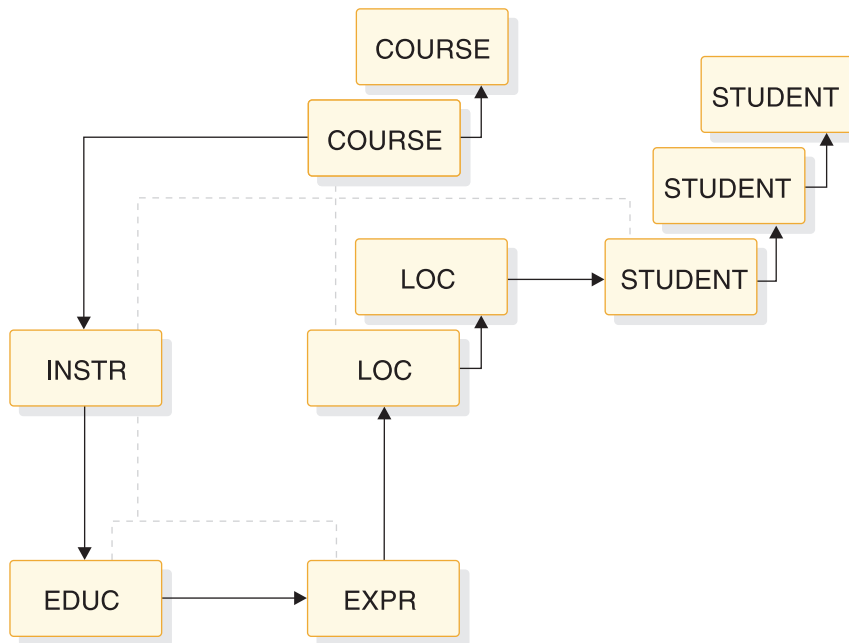
Figure 37. Hierarchical forward pointers

When an application program issues a call for a segment, HF pointers are followed until the specified segment is found. In this sense, the use of HF pointers in an HD database is similar to using a sequentially organized database. In both, to reach a dependent segment all segments that hierarchically precede it in the database record must be examined. HF pointers should be used when segments in a database record are typically processed in hierarchical sequence and processing does not require a significant number of delete operations. If there are a lot of delete operations, hierarchical forward and backward pointers (explained next) might be a better choice.

Four bytes are needed in each dependent segment's prefix for the HF pointer. Eight bytes are needed in the root segment. More bytes are needed in the root segment because the root points to both the next root segment and first dependent segment in the database record.

If you are defining your database with DBD generation macro statements, HF pointers are specified by coding PTR=H in the SEGM statement in the DBD.

If you are defining your database by using DDL statements, HF pointers are specified by coding the HIER parameter in a CREATE or ALTER TABLE statement.

Restriction: HALDB databases do not support HF pointers.

Hierarchical forward and backward pointers

With hierarchical forward and backward pointers (HF and HB), each segment in a database record points to both the segment that follows and the one that precedes it in the hierarchy (except dependent segments do not point back to root segments).

HF and HB pointers must be used together, since you cannot use HB pointers alone. The following figure shows how HF and HB pointers work.
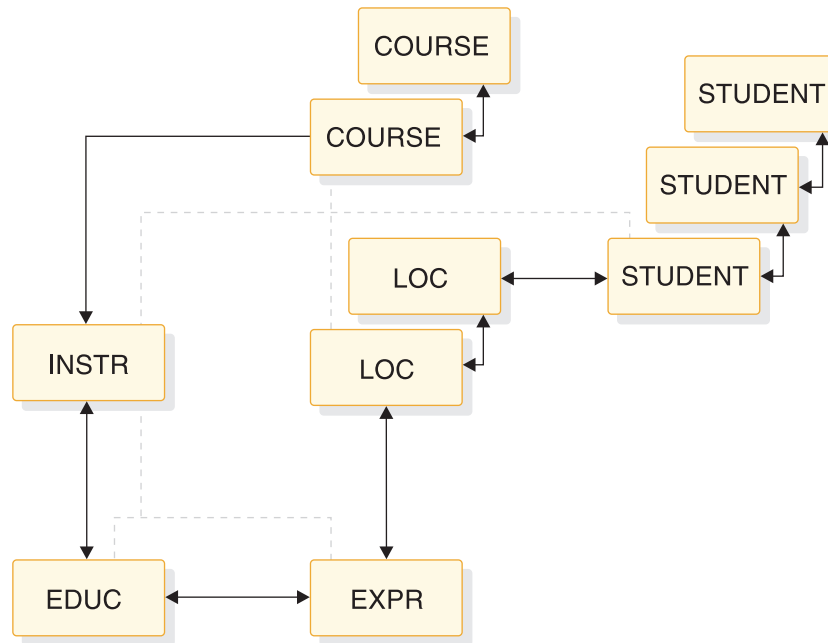
Figure 38. Hierarchical forward and backward pointers

HF pointers work in the same way as the HF pointers that are described in "Hierarchical forward pointers" on page 149.

HB pointers point from a segment to one immediately preceding it in the hierarchy. In most cases, HB pointers are not required for delete processing. IMS saves the location of the previous segment retrieved on the chain and uses this information for delete processing. The backward pointers are useful for delete processing if the previous segment on the chain has not been accessed. This happens when the segment to be deleted is entered by a logical relationship.

The backward pointers are useful only when all of the following are true:
- Direct pointers from logical relationships or secondary indexes point to the segment being deleted or one of its dependent segments.
- These pointers are used to access the segment.
- The segment is deleted.

Eight bytes are needed in each dependent segment's prefix to contain HF and HB pointers. Twelve bytes are needed in the root segment. More bytes are needed in the root segment because the root points:
- Forward to a dependent segment
- Forward to the next root segment in the database
- Backward to the preceding root segment in the database

If you are defining your database with DBD generation macro statements, HF and HB pointers are specified by coding PTR=HB in the SEGM statement in the DBD.

If you are defining your database by using DDL statements, HF and HB pointers are specified by coding the HIERBWD parameter in a CREATE or ALTER TABLE statement.

Restriction: HALDB databases do not support HF and HB pointers.

Physical child first pointers

With physical child first (PCF) pointers, each parent segment in a database record points to the first occurrence of each of its immediately dependent child segment types.

The following figure shows PCF pointers:
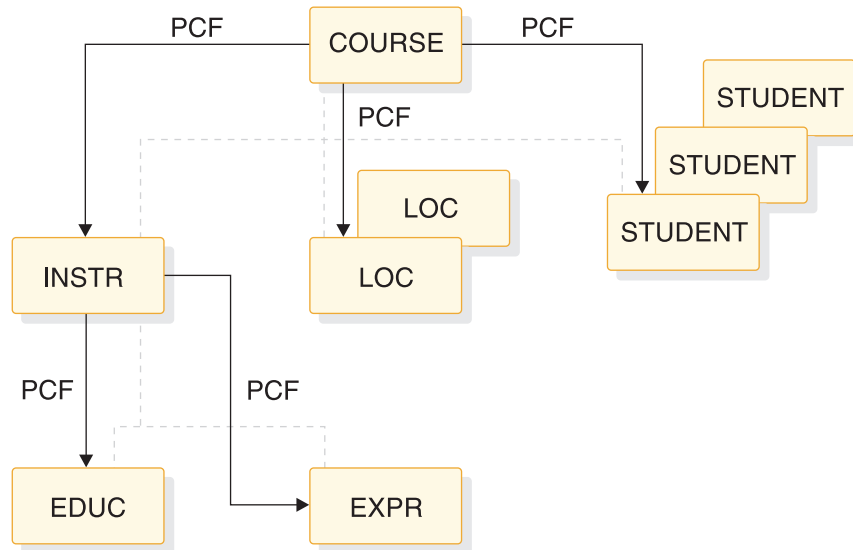
Figure 39. Physical child first pointers

With PCF pointers, the hierarchy is only partly connected. No pointers exist to connect occurrences of the same segment type under a parent. Physical twin pointers can be used to form this connection. Use PCF pointers when segments in a database record are typically processed randomly and either sequence fields are defined for the segment type, or if not defined, the insert rule is FIRST or HERE. If sequence fields are not defined and new segments are inserted at the end of existing segment occurrences, the combination of PCF and physical child last (PCL) pointers (explained next) can be a better choice.

Four bytes are needed in each parent segment for each PCF pointer.

PCF pointers are specified by coding PARENT=((name,SNGL)) in the SEGM statement in the DBD. This is the SEGM statement for the child being pointed to, not the SEGM statement for the parent. Note, however, that the pointer is stored in the parent segment.

If you are using DDL, PCF pointers are specified by coding the FOREIGN KEY clause in the CREATE TABLE statement for the database as FOREIGN KEY REFERENCES *table_name* SINGLE. This is the table for the child being pointed to, not the table for the parent. Again, note that the pointer is stored in the parent segment.

Related concepts:

"Types of pointers you can specify" on page 148

How logical relationships affect your programming (Application Programming)

Related reference:

ISRT call (Application Programming APIs)

➡️   CREATE TABLE (Application Programming APIs)

➡️   SEGM statements (System Utilities)

Physical child first and last pointers

With physical child first and last pointers (PCF and PCL), each parent segment in a database record points to both the first and last occurrence of its immediately dependent child segment types.

PCF and PCL pointers must be used together, since you cannot use PCL pointers alone. The following figure shows PCF and PCL pointers:
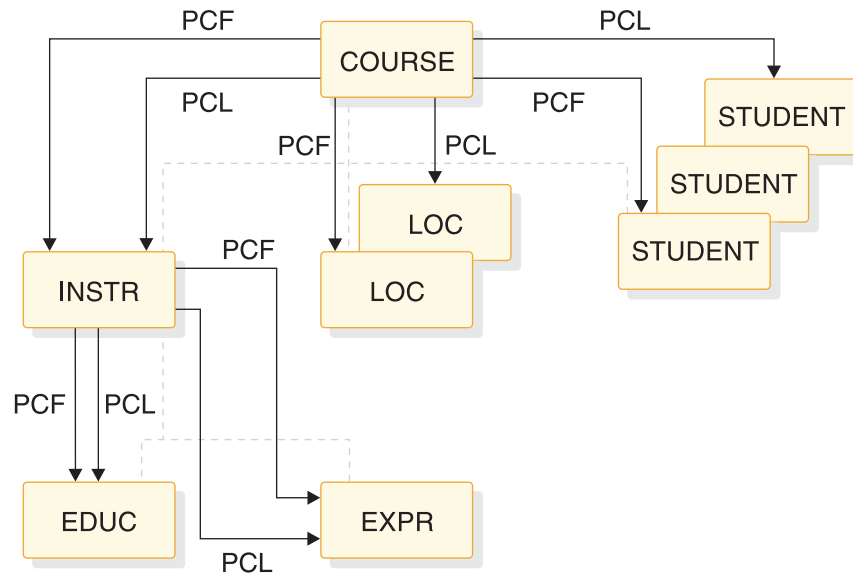

Figure 40. Physical child first and last pointers

Note that if only one physical child of a particular parent segment exists, the PCF and PCL pointers both point to the same segment. As with PCF pointers, PCF and PCL pointers leave the hierarchy only partly connected, and no pointers exist to connect occurrences of the same segment type under a parent. Physical twin pointers can be used to form this connection.

PCF and PCL pointers (as opposed to just PCF pointers) are typically used when:
- No sequence field is defined for the segment type.
- New segment occurrences of a segment type are inserted at the end of all existing segment occurrences.

On insert operations, if the ISRT rule of LAST has been specified, segments are inserted at the end of all existing segment occurrences for that segment type. When PCL pointers are used, fast access to the place where the segment will be inserted is possible. This is because there is no need to search forward through all segment occurrences stored before the last occurrence. PCL pointers also give application programs fast retrieval of the last segment in a chain of segment occurrences. Application programs can issue calls to retrieve the last segment by using an unqualified SSA with the command code L. When a PCL pointer is followed to get the last segment occurrence, any further movement in the database is forward.

A PCL pointer does not enable you to search from the last to the first occurrence of a series of dependent child segment occurrences.

Four bytes are needed in each parent segment for each PCF and PCL pointer.

PCF and PCL pointers are specified by coding the PARENT= operand in the SEGM statement in the DBD as PARENT=((name,DBLE)). This is the SEGM statement for the child being pointed to, not the SEGM statement for the parent. Note, however, that the pointers are stored in the parent segment.

If you are using DDL, PCF and PCL pointers are specified by coding the FOREIGN KEY clause in the CREATE TABLE statement for the database as FOREIGN KEY REFERENCES *table_name* DOUBLE. This is the table for the child being pointed to, not the table for the parent.

A parent segment can have SNGL specified on one immediately dependent child segment type and DBLE specified on another.

The following example DBD statement specifies PCF and PCL pointers.

```
DBD
SEGM A
SEGM B  PARENT=((name.SNGL))   (specifies PCF pointer only)
SEGM C  PARENT=((name.DBLE))   (specifies PCF and PCL pointers)
```

The following example shows the same PCF and PCL pointer specifications as specified by using DDL.

```
CREATE DATABASE ...
CREATE TABLE A ...
CREATE TABLE B (
      ...
      FOREIGN KEY REFERENCES table_name SINGLE    (specifies PCF pointer only)
    )
CREATE TABLE C (
      ...
      FOREIGN KEY REFERENCES table_name DOUBLE    (specifies PCF and PCL pointers)
    )
```

The following figure shows the result of specifying PCF and PCL pointers in a database definition.
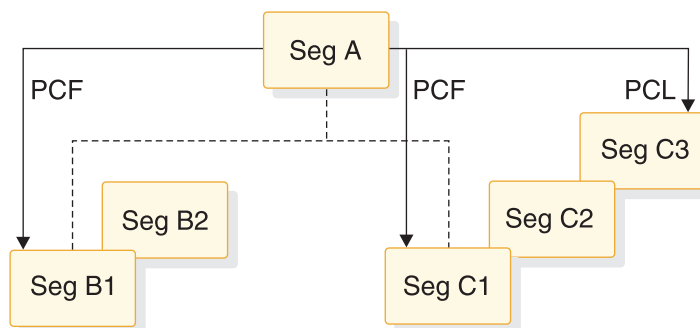
Figure 41. Specifying PCF and PCL pointers

Related concepts:

"Types of pointers you can specify" on page 148

How logical relationships affect your programming (Application Programming)

Related reference:

ISRT call (Application Programming APIs)

➡ CREATE TABLE (Application Programming APIs)

➡ SEGM statements (System Utilities)

Physical twin forward pointers

With physical twin forward (PTF) pointers, each segment occurrence of a given segment type under the same parent points forward to the next segment occurrence.

Note that, except in PHIDAM databases, PTF pointers can be specified for root segments. When this is done in an HDAM or PHDAM database, the root segment points to the next root in the database chained off the same root anchor points (RAP). If no more root segments are chained from this RAP, the PTF pointer is zero.

When PTF pointers are specified for root segments in a HIDAM database, the root segment does *not* point to the next root in the database.

If you specify PTF pointers on a root segment in a HIDAM database, the HIDAM index must be used for all sequential processing of root segments. Using only PTF pointers increases access time. You can eliminate this overhead by specifying PTF and physical twin backward (PTB) pointers.

You cannot use PTF pointers for root segments in a PHIDAM database. PHIDAM databases only support PTF pointers for dependent segments.

With PTF pointers, the hierarchy is only partly connected. No pointers exist to connect parent and child segments. Physical child pointers can be used to form this connection. PTF pointers should be used when segments in a database record are typically processed randomly, and you do not need sequential processing of database records.

Four bytes are needed for the PTF pointer in each segment occurrence of a given segment type.

If you are defining your database with DBD generation macro statements, PTF pointers are specified by coding PTR=T in the SEGM statement that defines the segment that contains the pointer.

If you are defining your database by using DDL statements, PTF pointers are specified by coding the TWIN parameter in a CREATE or ALTER TABLE statement that defines the segment that contains the pointer.

The following figure show PTF pointers:
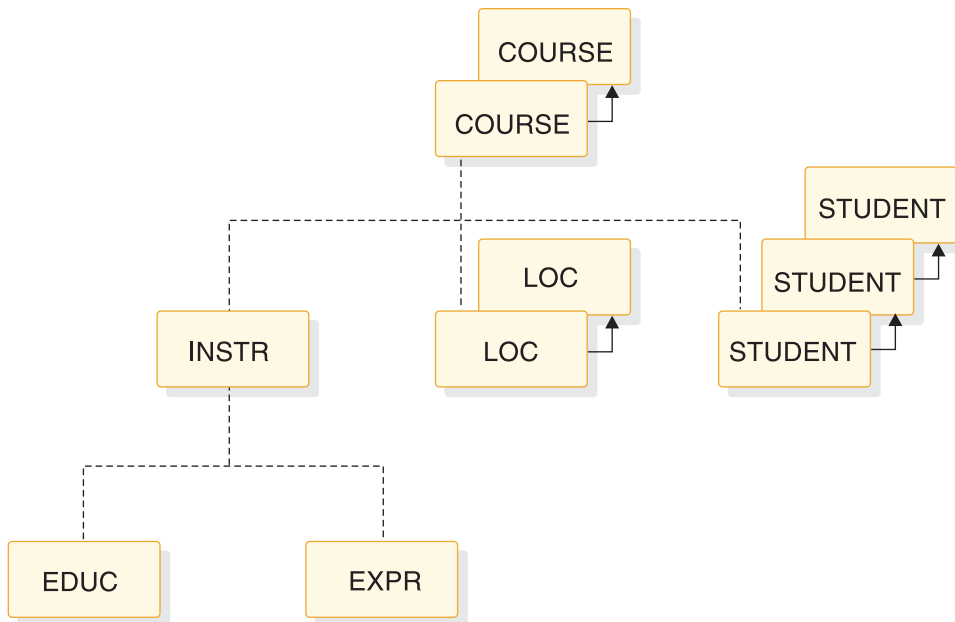
Figure 42. Physical twin forward pointers

Related concepts:

"Use of RAPs in a HIDAM database" on page 170

"Physical twin forward and backward pointers"

"General format of HD databases and use of special fields" on page 160

Physical twin forward and backward pointers

With physical twin forward and backward (PTF and PTB) pointers, each segment occurrence of a given segment type under the same parent points both forward to the next segment occurrence and backward to the previous segment occurrence.

PTF and PTB pointers must be used together, since you cannot use PTB pointers alone. The following figure illustrates how PTF and PTB pointers work.
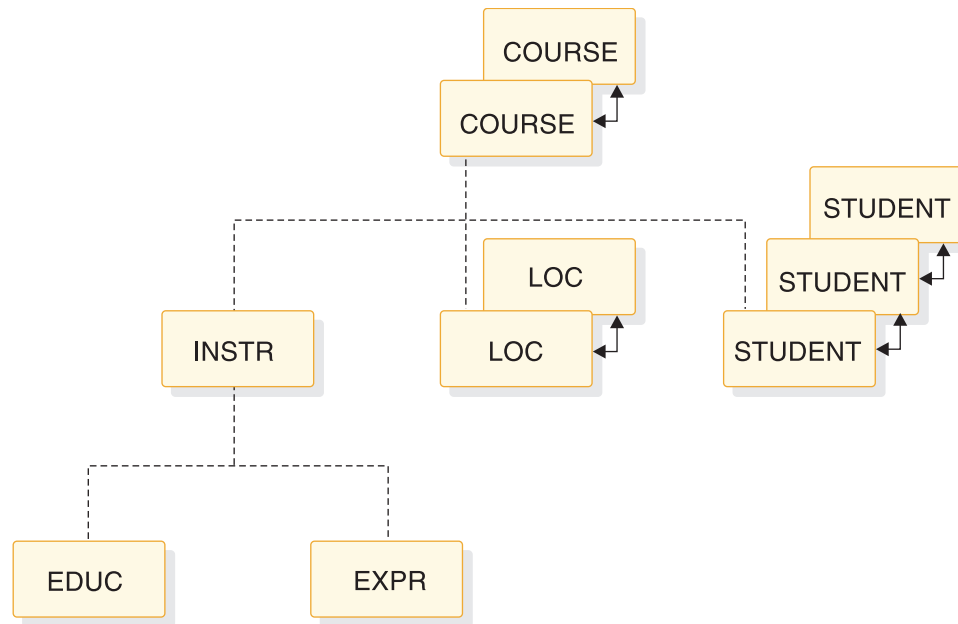
Figure 43. Physical twin forward and backward pointers

Note that PTF and PTB pointers can be specified for root segments. When this is done, the root segment points to both the next and the previous root segment in the database. As with PTF pointers, PTF and PTB pointers leave the hierarchy only partly connected. No pointers exist to connect parent and child segments. Physical child pointers (explained previously) can be used to form this connection.

PTF and PTB pointers (as opposed to just PTF pointers) should be used on the root segment of a HIDAM or a PHIDAM database when you need fast sequential processing of database records. By using PTB pointers in root segments, an application program can sequentially process database records without IMS having to refer to the HIDAM or PHIDAM index. For HIDAM databases, PTB pointers improve performance when deleting a segment in a twin chain accessed by a virtually paired logical relationship. Such twin-chain access occurs when a delete from the logical access path causes DASD space to be released.

Eight bytes are needed for the PTF and PTB pointers in each segment occurrence of a given segment type.

If you are defining your database with DBD generation macro statements, PTF and PTB pointers are specified by coding PTR=TB in the SEGM statement in the DBD.

If you are defining your database by using DDL statements, PTF and PTB pointers are specified by coding the TWINBWD parameter in a CREATE or ALTER TABLE statement.

Related concepts:

"Physical twin forward pointers" on page 155

Mixing pointers

Because pointers are specified by segment type, the various types of pointers can be mixed within a database record. However, only hierarchical or physical, but not both, can be specified for a given segment type.

The types of pointers that can be specified for a segment type are:

HF   Hierarchical forward

HF and HB
    Hierarchical forward and backward

PCF   Physical child first

PCF and PCL
    Physical child first and last

PTF   Physical twin forward

PTF and PTB
    Physical twin forward and backward

The figure below shows a database record in which pointers have been mixed. Note that, in some cases, for example, dependent segment B, many pointers exist even though only one type of pointer is or can be specified. Also note that if a segment is the last segment in a chain, its last pointer field is set to zero (the case for segment E1, for instance). One exception is noted in the rules for mixing pointers. The figure has a legend that explains what specification in the PTR= or PARENT= operand causes a particular pointer to be generated.

The rules for mixing pointers are:

- If PTR=H is specified for a segment, no PCF pointers can exist from that segment to its children. For a segment to have PCF pointers to its children, you must specify PTR=T or TB for the segment.
- If PTR=H or PTR=HB is specified for the root segment, the first child will determine if an H or HB pointer is used. All other children must be of the same type.
- If PTR=H is specified for a segment other than the root, PTR=TB and PTR=HB cannot be specified for any of its children. If PTR=HB is specified for a segment other than the root, PTR=T and PTR=H cannot be specified for any of its children.

  That is, the child of a segment that uses hierarchical pointers must contain the same number of pointers (twin or hierarchical) as the parent segment.
- If PTR=T or TB is specified for a segment whose immediate parent used PTR=H or PTR=HB, the last segment in the chain of twins does not contain a zero. Instead, it points to the first occurrence of the segment type to its right on the same level in the hierarchy of the database record. This is true even if no twin *chain* yet exists, just a single segment for which PTR=T or TB is specified (dependent segment B and E2 in the figure illustrate this rule).
- If PTR=H or HB is specified for a segment whose immediate parent used PTR=T or TB, the last segment in the chain of twins contains a zero (dependent segment C2 in the figure illustrates this rule).

The following figure shows an example of mixing pointers in a database record.

Usage of the twin forward pointer position ⟶

Segments pointed to ⟶

Notes below ⟶

Root A1

T	H
A2	B1
1	1

Root A2

PTR=H **Dependent B1**

H	
C1	
2	

Dependent B2

H	
G1	
2	

PTR=T **Dependent G1**

H	PCF
0	H1
2	5

PTR=T **Dependent C1**

H	PCF	PCF	PCL
C2	D1	E1	E2
2	3	4	4

PTR=H **Dependent C2**

H	
F1	
2	

PTR=H **Dependent F1**

H	
F2	
2	

Dependent F2

H	
B2	
2	

PTR=T **Dependent H1**

T	PCF
H2	I1
2	5

Dependent H2

T	
0	
2	

PTR=H Parent=SNGL **Dependent D1**

H	
D2	
2	

Dependent D2

H	
0	
2	

PTR=T Parent=DBLE **Dependent E1**

H	
E2	
2	

Dependent E2

H	
0	
2	

PTR=H **Dependent I1**

H	
0	
2	

Figure 44. Mixing pointers

Notes for Figure:

1. These pointers are generated when you specify PTR=H on the root segment.
2. If you specify PTR=H, usage is hierarchical (H); otherwise usage is twin (T).
3. These pointers are generated when you specify PTR=T on segment type C and PARENT=SNGL on segment type D
4. These pointers are generated when you specify PTR=T on segment type C and PARENT=DBLE on segment type E
5. These pointers are generated when you specify PTR=T on this segment type

Related concepts:

"Types of pointers you can specify" on page 148

Related tasks:

"Determining segment size" on page 560

Sequence of pointers in a segment's prefix

When a segment contains more than one type of pointer, pointers are put in the segment's prefix in a specific sequence.

The pointers are put in the segment's prefix in the following sequence:

1. HF

2. HB

Or:

1. PTF
2. PTB
3. PCF
4. PCL

General format of HD databases and use of special fields

The way in which an HD database is organized is not particularly complex, but some of the special fields in the database used for things like managing space make HD databases seem quite different from sequentially organized databases.

The databases referred to here are the HDAM or PHDAM and the HIDAM or PHIDAM databases. HIDAM and PHIDAM each have an additional database, the primary index database, for which you must allocate a data set. For HIDAM databases, the primary index database must be defined with its own set of statements. For PHIDAM databases, the primary index database does not need to be defined by its own set of statements. For both, IMS maintains the index. This topic examines the index database when dealing with the storage of HIDAM records. The following figure shows the general format of an HD database and some of the special fields used in it.
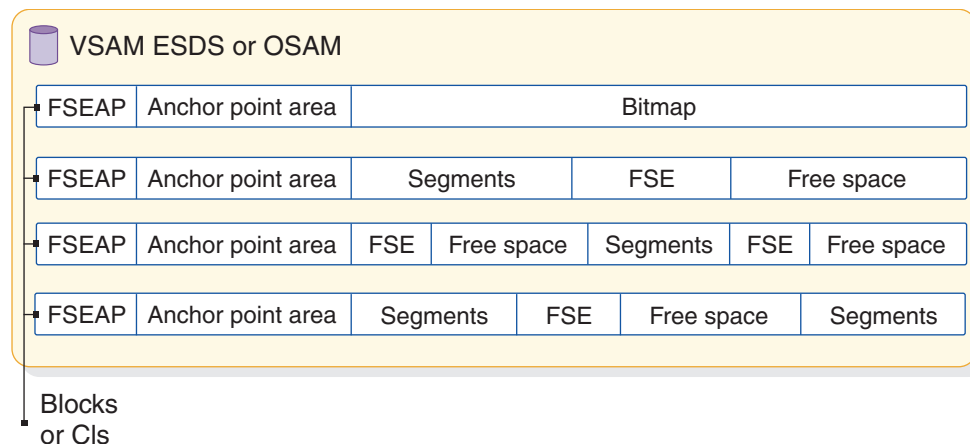
Figure 45. Format of an HD database and special fields in it

HD databases use a single data set, that is either a VSAM ESDS or an OSAM data set. The data set contains one or more CIs (VSAM ESDS) or blocks (OSAM). Database records in the data set are in unblocked format. Logical record length is the same as the block size when OSAM is used. When VSAM is used, logical record length is slightly less than CI size. (VSAM requires some extra control information in the CI.)

You can either specify logical record length yourself or accept the IMS defaults. IMS generates logical record lengths equal to a quarter, third, half, or full track block.

All segments in HD Databases begin on a halfword boundary. If a segment's total length is an odd number, the space used in an HD database will be one byte longer than the segment. The extra byte is called a "slack byte".

Note that the database in the figure above contains areas of free space. This free space could be the result of delete or replace operations done on segments in the data set. Remember, space can be reused in HD databases. Or it could be free space you set aside when loading the database. HD databases allow you to set aside free space by specifying that periodic blocks or CIs be left free or by specifying that a percentage of space in each block or CI be left free.

Examine the four fields illustrated in the above figure. Three of the fields are used to manage space in the database. The remaining one, the anchor point area, contains the addresses of root segments. The fields are:

- Bitmap
- Free space element anchor point
- Free space element
- Anchor point area

Related concepts:

"Physical twin forward pointers" on page 155

Related tasks:

"Step 5. Determine the amount of space needed for bitmaps" on page 566

Bitmaps

Bitmaps contain a string of bits. Each bit describes whether enough space is available in a particular CI or block to hold an occurrence of the longest segment defined in the data set group.

The first bit says whether the CI or block that the bitmap is in has free space. Each consecutive bit says whether the next consecutive CI or block has free space. When the bit value is one, it means the CI or block has enough space to store an occurrence of the longest segment type you have defined in the data set group. When the bit value is zero, not enough space is available.

The first bitmap in an OSAM data set is in the first block of the first extent of the data set. In VSAM data sets, the second CI is used for the bitmap and the first CI is reserved. The first bitmap in a data set contains n bits that describe space availability in the next $n-1$ consecutive CIs or blocks in the data set. After the first bitmap, another bitmap is stored at every nth CI or block to describe whether space is available in the next group of CIs or blocks in the data set.

For a HALDB partition, the first bitmap block stores the partition ID (2 bytes) and the reorganization number (2 bytes). These are stored before the FSEAP at the beginning of the block.

An example bitmap is shown in the following figure.
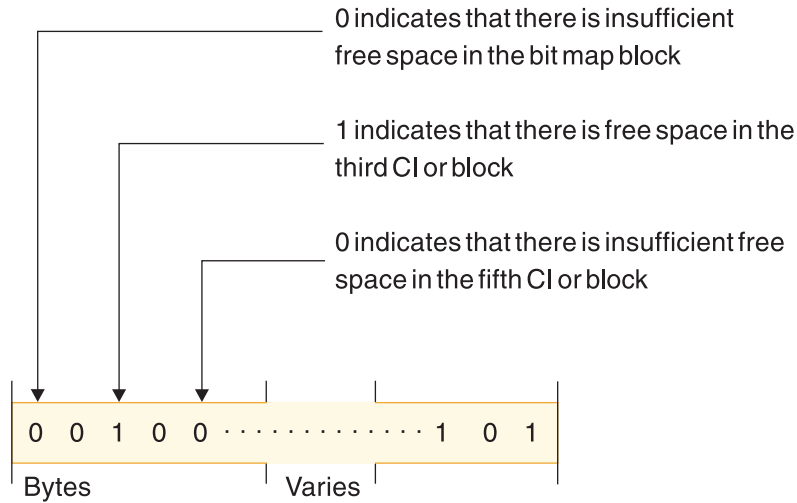
Figure 46. Bitmap for HD databases

Free space element anchor point (FSEAP)

Free space element anchor points (FSEAP) are made up of two 2-byte fields.

The first field contains the offset, in bytes, to the first free space element (FSE) in the CI or block. FSEs describe areas of free space in a block or CI. The second field identifies whether this block or CI contains a bitmap. If the block or CI does not contain a bitmap, the field is zeros. One FSEAP exists at the beginning of every CI or block in the data set. IMS automatically generates and maintains FSEAPs.

An FSEAP is shown in the following figure.
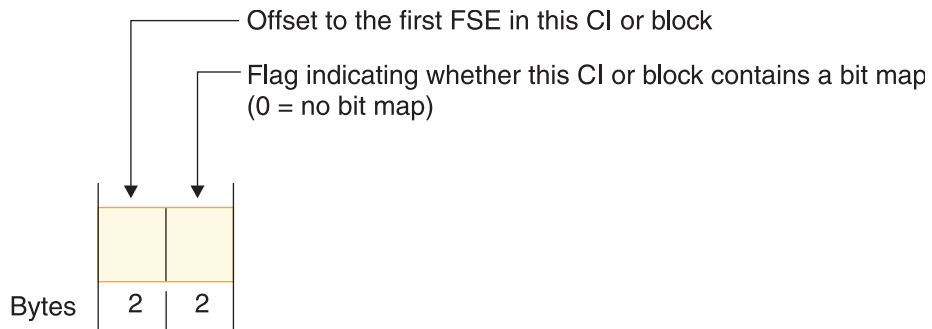
Figure 47. An FSEAP

The FSEAP in the first bitmap block in an OSAM data set has a special use. It is used to contain the DBRC usage indicator for the database. The DBRC usage indicator is used at database open time for update processing to verify usage of the correct DBRC RECON data set.

Free space element (FSE)

An FSE describes each area of free space in a CI or block that is 8 or more bytes in length.

IMS automatically generates and maintains FSEs. FSEs occupy the first 8 bytes of the area that is free space. FSEs consist of three fields:

- Free space chain pointer (CP) field. This field contains, in bytes, the offset from the beginning of this CI or block to the next FSE in the CI or block. This field is 2 bytes long. The CP field is set to zero if this is the last FSE in the block or CI.
- Available length (AL) field. This field contains, in bytes, the length of the free space identified by this FSE. The value in this field includes the length of the FSE itself. The AL field is 2 bytes long.
- Task ID (ID) field. This field contains the task ID of the program that freed the space identified by the FSE. The task ID allows a given program to free and reuse the same space during a given scheduling without contending for that space with other programs. The ID field is 4 bytes long.

An FSE is shown in the following figure.
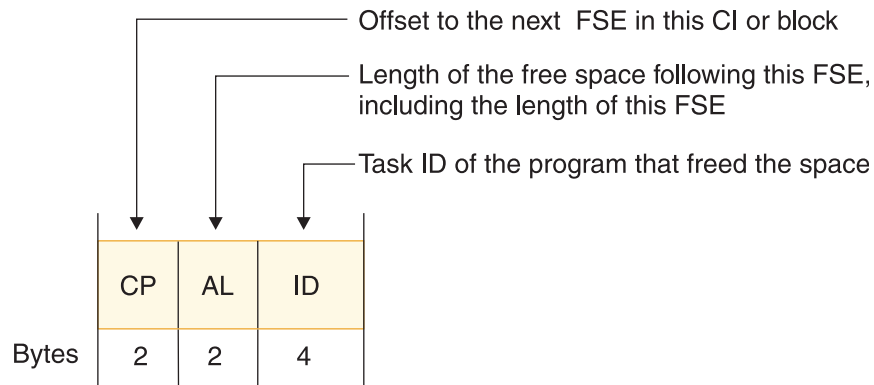
Figure 48. An FSE

Anchor point area

The anchor point area is made up of one or more 4-byte root anchor points (RAPs).

Each RAP contains the address of a root segment. For HDAM, you specify the number of RAPs you need on the RMNAME parameter in the DBD statement. For PHDAM, you specify the number of RAPs you need on the RMNAME parameter in the DBD statement, or by using the HALDB Partition Definition utility, or on the DBRC INIT.PART command. For HIDAM (but not PHIDAM), you specify whether RAPs exist by specifying PTR=T or PTR=H for a root segment type. Only one RAP per block or CI is generated. How RAPs are used in HDAM, PHDAM, and HIDAM differs.

An anchor point area in an HDAM or PHDAM database is shown in the following figure.

Anchor point area containing, in this case, two RAPs

	RAP	RAP
Bytes	4	4
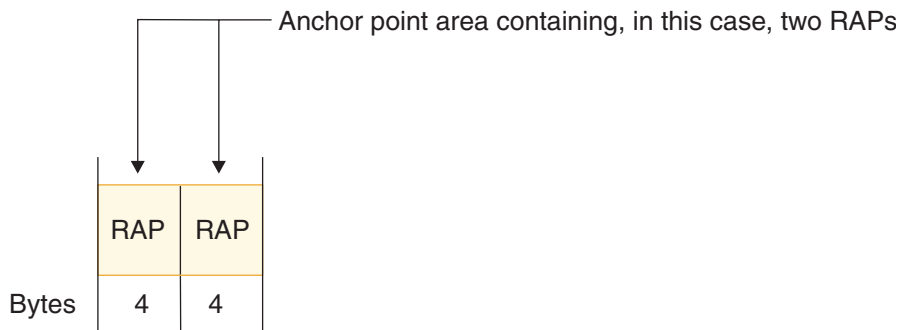
Figure 49. An HDAM or PHDAM anchor point area

Related concepts:

"How HDAM and PHDAM records are stored"

"How HIDAM and PHIDAM records are stored" on page 168

How HDAM and PHDAM records are stored

HDAM or PHDAM databases consist of two parts: a root addressable area and an overflow area.

The root addressable area contains root segments and is the primary storage area for dependent segments in a database record. The overflow area is for the storage of segments that do not fit in the root addressable area. You specify the size of the root addressable area in the relative block number (RBN) operand of the RMNAME parameter in the DBD statement. For PHDAM, you can also use the HALDB Partition Definition utility to specify the size of the root addressable area. You also specify the maximum number of bytes of a database record to be stored in the root addressable area by using the BYTES operand of the RMNAME parameter in the DBD statement. For PHDAM databases, you can use the HALDB Partition Definition utility to specify the maximum number of bytes in the root addressable area.

The following figure shows example SKILL database records.

Figure 50. Two example SKILL records in an HD database

The following figure shows how these records are stored in a HDAM or HIDAM database.

VSAM ESDS or OSAM

| RAP | RAP | SKILL3 | NAME4 | EDUC4 | EDUC5 | EDUC6 | Free space |

Root addressable area

| RAP | RAP | Free space |

| RAP | RAP | SKILL1 | NAME1 | EXPR1 | EDUC1 | NAME2 | Free space |

Overflow area

| EXPR2 | EXPR3 | EXPR4 | NAME3 | EDUC2 | EDUC3 | Free space |

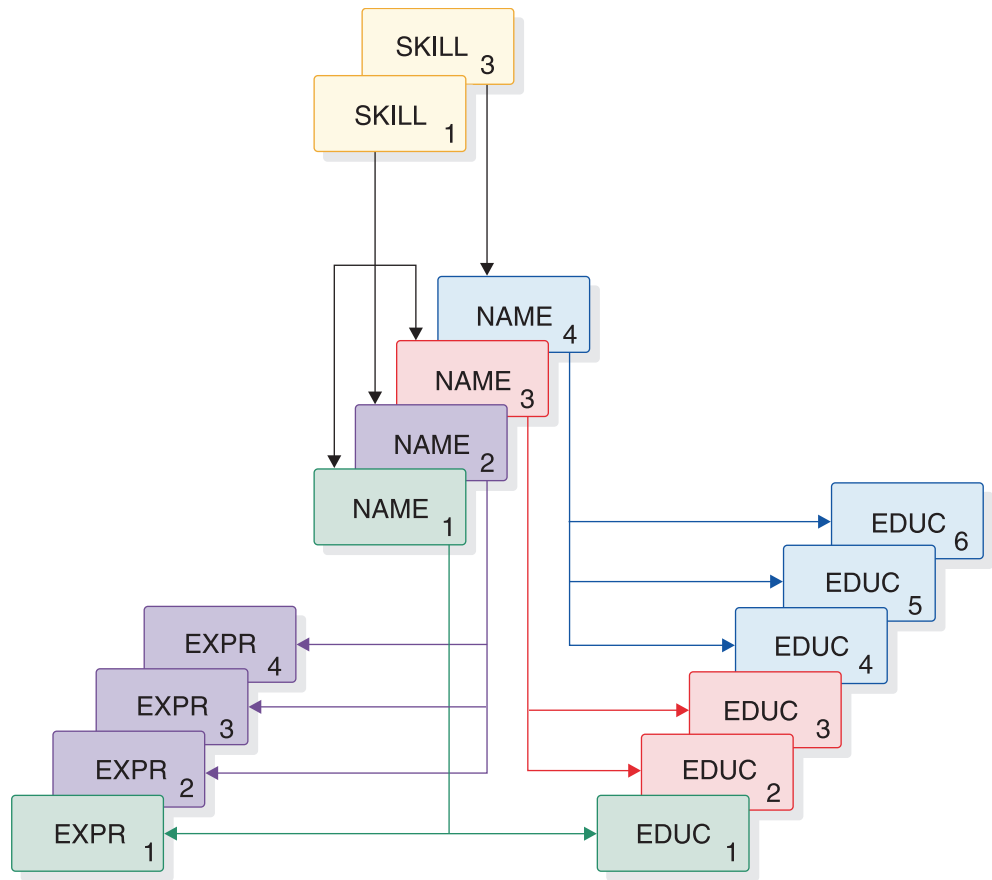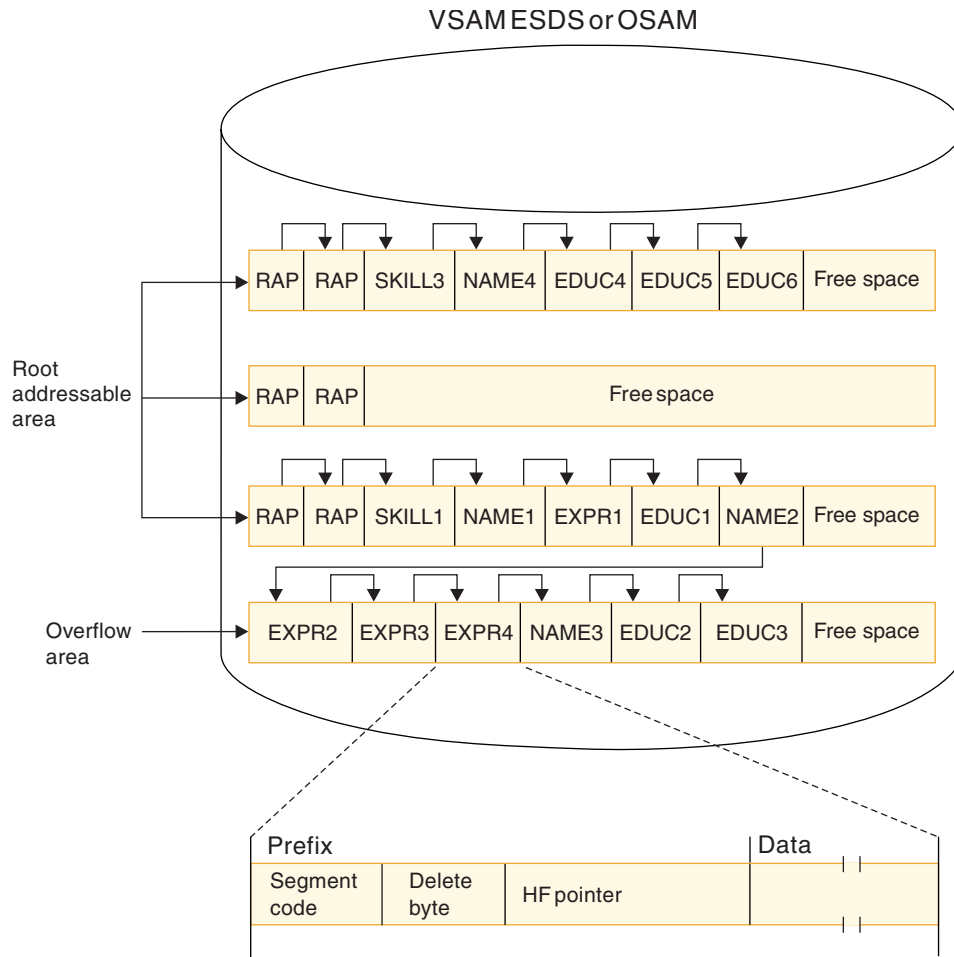| Prefix			Data	
Segment code	Delete byte	HF pointer		

Figure 51. HDAM or PHDAM database records in storage

When the database is initially loaded, the root and each dependent segment are put in the root addressable area until the next segment to be stored will cause the total space used to exceed the amount of space you specified in the BYTES operand. At this point, all remaining dependent segments in the database record are stored in the overflow area.

In an HDAM or a PHDAM database, the order in which you load database records does not matter. The user randomizing module determines where each root is stored. However, as with all types of databases, when the database is loaded, all dependents of a root must be loaded in hierarchical sequence following the root.

To store an HDAM or a PHDAM database record, the randomizing module takes the root's key and, by hashing or some other arithmetic technique, computes an RBN or CI number and a RAP number within the block or CI. The module gives these numbers to IMS, and IMS determines where in the root addressable area to store the root. The RBN or CI tells IMS in which CI or block (relative to the beginning of the data set) the RAP will be stored. The RAP number tells which RAP in the CI or block will contain the address of the root. During load, IMS stores the root and as many of its dependent segments that will fit (based on the bytes operand) in the root addressable area.

When the database is initially loaded, it puts the root and segments in the first available space in the specified CI or block, if this is possible. IMS then puts the

4-byte address of the root in the RAP of the CI or block designated by the randomizing module. RAPs only exist in the root addressable area. If space is not available in the root addressable area for a root, it is put in the overflow area. The root, however, is chained from a RAP in the root addressable area.

Related concepts:

"When not enough root storage room exists"

"Anchor point area" on page 163

"Inserting root segments into an HDAM or PHDAM database" on page 172

When not enough root storage room exists

If the CI or block specified by the randomizing module does not contain enough room to store the root, IMS uses the HD space search algorithm to find space.

When insufficient space exists in the specified CI or block to store the root, the algorithm finds the closest available space to the specified CI or block. When space is found, the address of the root is still stored in the specified RAP in the original block or CI generated by the randomizing module.

If the randomizing module generates the same relative block and RAP number for more than one root, the RAP points to a single root and all additional roots with the same relative block and RAP number are chained to each other using physical twin pointers. Roots are always chained in ascending key sequence. If non-unique keys exist, the ISRT rules of FIRST, LAST, and HERE determine the sequence in which roots are chained. All roots chained like this from a single anchor point area are called *synonyms*.

"How HDAM and PHDAM records are stored" on page 164 shows two HDAM or PHDAM database records and how they appear in storage after initial load. In this example, enough space exists in the specified block or CI to store the roots, and the unique relative block and RAP numbers for each root generated by the randomizing module. The bytes parameter specifies enough space for five segments of the database record to fit in the root addressable area. All remaining segments are put in the overflow area. When HDAM or PHDAM database records are initially loaded, dependent segments that cannot fit in the root addressable area are simply put in the first available space in the overflow area.

Note how segments in the database record are chained together. In this case, hierarchical pointers are used instead of the combination of physical child/physical twin pointers. Each segment points to the next segment in hierarchical sequence. Also note that two RAPs were specified per CI or block and each of the roots loaded is pointed to by a RAP. For simplicity, "How HDAM and PHDAM records are stored" on page 164 does not show the various space management fields.

An HDAM or PHDAM segment in storage consists of a prefix followed by user data. The first byte of the prefix is the segment code, which identifies the segment type to IMS. This number can be from 1 to 255. The segment code is assigned to the segment type by IMS in ascending sequence, starting with the root segment and continuing through all dependents in hierarchical sequence. The second byte of the prefix is the delete byte. The third field in the prefix contains the one or more addresses of segments to which this segment is pointing. In this example, hierarchical forward pointers are used. Therefore, the EXPR4 segment contains only one address, the address of the NAME3 segment.

Related concepts:

"How the HD space search algorithm works" on page 177

Related reference:

➦ ISRT call (Application Programming APIs)

How HIDAM and PHIDAM records are stored

A HIDAM database is actually composed of two databases. One database contains the database records and the other database contains the HIDAM index. HIDAM uses the index to get to a specific root segment rather than the root anchor points that HDAM and PHDAM use.

Related concepts:

How a HIDAM or PHIDAM database is loaded

Root segments in a HIDAM or PHIDAM database must have a unique key field, because an index entry exists for each root segment based on the root's key.

When initially loading a HIDAM or a PHIDAM database, you should present all root segments to the load program in ascending key sequence, with all dependents of a root following in hierarchical sequence. The figure below shows how the two Skills database records shown in Figure 50 on page 165 appear in storage after initial load. Note that HIDAM and PHIDAM, unlike HDAM and PHDAM, have no root addressable or overflow area, just a series of blocks or CIs.

Restriction: Load programs for PHIDAM databases must run in a DLI region type. Load programs for HIDAM databases do not have this restriction.

When database records are initially loaded, they are simply loaded one after another in the order in which they are presented to the load program. The space in the following figure at the end of each block or CI is free space specified when the database was loaded. In this example, 30% free space per block or CI was specified.
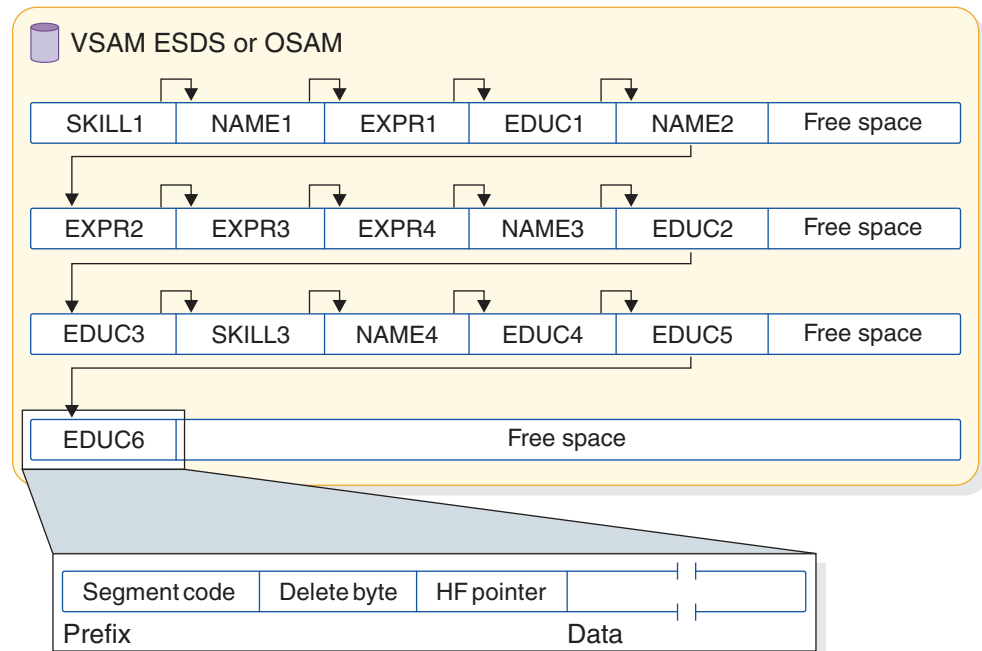
Figure 52. HIDAM database records in storage

Note how segments in a database record are chained together. In this case, hierarchical pointers were used instead of the combination of physical child/physical twin pointers. Each segment points to the next segment in hierarchical sequence. No RAPs exist in the figure above. Although HIDAM databases can have RAPs, you probably do not need to use them.

In storage, a HIDAM or PHIDAM segment consists of a prefix followed by user data. The first byte of the prefix is the segment code, which identifies the segment type to IMS. This number can be from 1 to 255. The segment code is assigned to the segment by IMS in ascending sequence, starting with the root segment and continuing through all dependents in hierarchical sequence. The second byte of the prefix is the delete byte. The third field in the prefix contains the one or more addresses of segments to which this segment is pointing. In this example, hierarchical forward pointers are used. The EDUC6 segment contains only one address, the address of the root segment of the next database record (not shown here) in the database.

Related concepts:

"Use of RAPs in a HIDAM database" on page 170

Creating an index segment

As each root is stored in a HIDAM or PHIDAM database, IMS creates an index segment for the root and stores it in the index database or data set.

The index database consists of a VSAM KSDS. The KSDS contains an index segment for each root in the database or HALDB partition. When initially loading a HIDAM or PHIDAM database, IMS will insert a root segment with a key of all X'FF's as the last root in the database or partition.

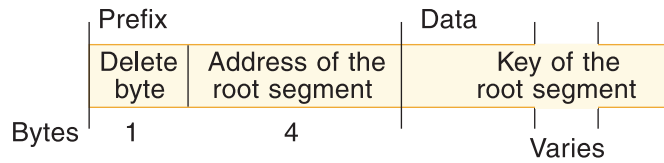The format of an index segment is shown in the following figure.

Figure 53. Format of an index segment

The prefix portion of the index segment contains the delete byte and the root's address. The data portion of the index segment contains the key field of the root being indexed. This key field identifies which root segment the index segment is for and remains the reason why root segments in a HIDAM or PHIDAM database must have unique sequence fields. Each index segment is a separate logical record. The following figure shows the index database that IMS would generate when the two database records in Figure 50 on page 165 were loaded.
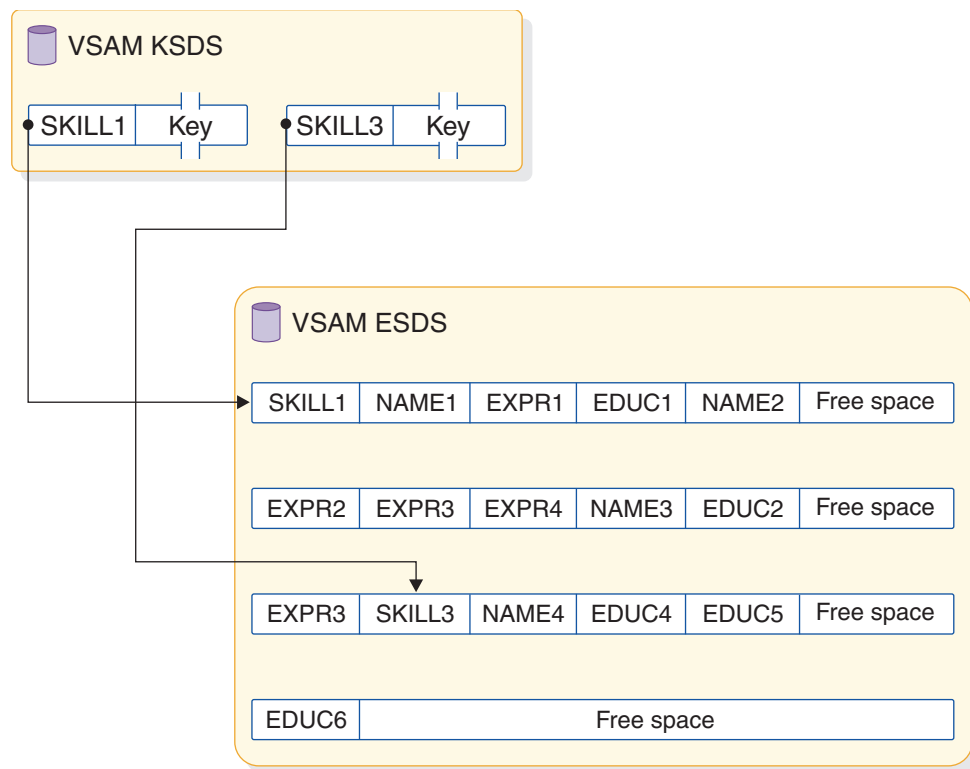
Figure 54. HIDAM or PHIDAM index databases

Use of RAPs in a HIDAM database

RAPs are used differently in HIDAM databases than they are in HDAM or PHDAM databases.

In HDAM or PHDAM, RAPs exist to point to root segments. When the randomizing module generates roots with the same relative block and RAP number (synonyms), the RAP points to one root and synonyms are chained together off that root.

In HIDAM databases, RAPs are generated only if you specify PTR=T or PTR=H for a root segment. When either of these is specified, one RAP is put at the beginning of each CI or block, and root segments within the CI or block are chained from the RAP in reverse order based on the time they were inserted. By this method, the

RAP points to the last root inserted into the block or CI, and the hierarchical or twin forward pointer in the first root inserted into the block or CI is set to zero. The hierarchical or twin forward pointer in each of the other root segments in the block points to the previous root inserted in the block.

The figure below shows what happens if you specify PTR=T or PTR=H for root segments in a HIDAM database. The figure uses the following abbreviations:

FSE   Free space element

RAP   Root anchor point

SC   Segment code

DB   Delete byte

TF   Twin forward
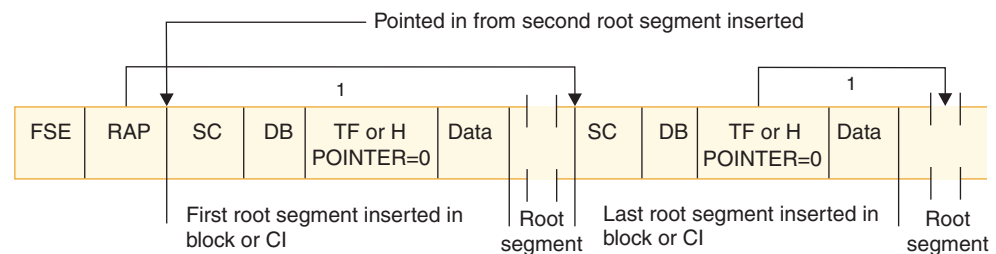
H   Hierarchical forward

Figure 55. Specifying PTR=T or PTR=H for root segments in a HIDAM database

Note that if you specify PTR=H for a HIDAM root, you get an additional hierarchical pointer to the first dependent in the hierarchy. In the preceding figure, a "1" indicates where this additional hierarchical pointer would appear.

The implication of using PTR=T or PTR=H is that the pointer from one root to the next cannot be used to process roots sequentially. Instead, the HIDAM index must be used for all sequential root processing, and this increases access time. Specify PTR=TB or PTR=HB for root segments in a HIDAM database. Then no RAP is generated, and GN calls against root segments proceed along the normal physical twin forward chain. If no pointers are specified for HIDAM root segments, the default is PTR=T.

Related concepts:

"How a HIDAM or PHIDAM database is loaded" on page 168

"Physical twin forward pointers" on page 155

Accessing segments

The way in which a segment in an HD database is accessed depends on whether the DL/I call for the segment is qualified or unqualified.

Qualified calls

When a call is issued for a root segment and the call is qualified on the root segment's key, the way in which the database record containing the segment is found depends on whether the database is HDAM, PHDAM, HIDAM, or PHIDAM.

In an HDAM or a PHDAM database, the randomizing module generates the root segment's (and therefore the database record's) location. In a HIDAM or a PHIDAM database, the HIDAM or PHIDAM index is searched until the index segment containing the root's key is found.

Once the root segment is found, if the qualified call is for a dependent segment, IMS searches for the dependent by following the pointers in each dependent segment's prefix. The exact way in which the search proceeds depends on the type of pointers you are using. The following figure shows how a dependent segment is found when PCF and PTF pointers are used.
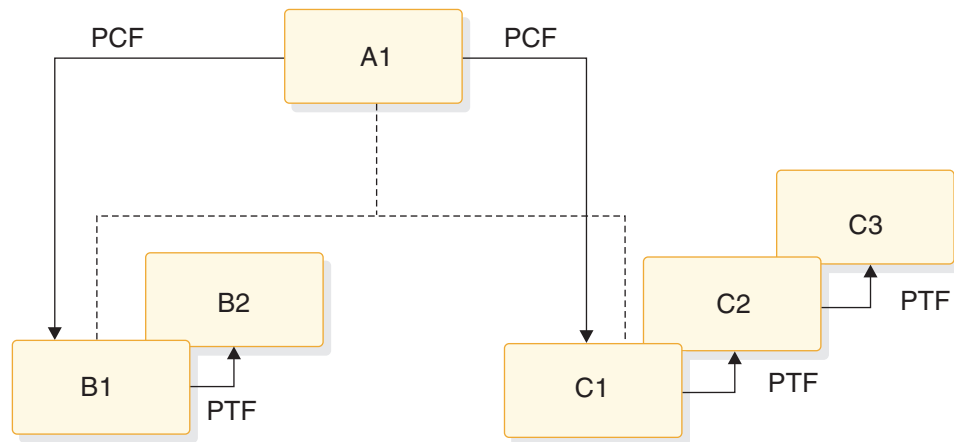

Figure 56. How dependent segments are found using PCF and PTF pointers

Unqualified calls

When an unqualified call is issued for a segment, the way in which the search proceeds depends several different factors.

The factors include:
- Whether the database is HDAM, PHDAM, HIDAM, or PHIDAM
- Whether a root or dependent segment is being accessed
- Where position in the database is currently established
- What type of pointers are being used
- Where parentage is set (if the call is a GNP)

Because of the many variables, it is not practical to generalize on how a segment is accessed.

Inserting root segments

The way in which a root segment is inserted into an HD database depends on whether the database is HDAM, PHDAM, HIDAM, or PHIDAM.

For PHDAM or PHIDAM databases, partition selection is first performed based on the key of the root segment.

Inserting root segments into an HDAM or PHDAM database

After initial load, root segments are inserted into an HDAM or PHDAM database in exactly the same way they are inserted during initial load.

Related concepts:

"How HDAM and PHDAM records are stored" on page 164

Inserting root segments into a HIDAM or PHIDAM database

Root segments are inserted into HIDAM and PHIDAM databases in ascending root sequence.

After initial load, root segments are inserted into a HIDAM or PHIDAM database as follows:

1. The HIDAM or PHIDAM index is searched for an index segment with a root key greater than the key of the root to be inserted.
2. The new index segment is inserted in ascending root sequence.
3. Once the index segment is created, the root segment is stored in the database at the location specified by the HD space search algorithm.

The following figure shows the insertion of a root segment, SKILL2, into the database first shown in Figure 54 on page 170.
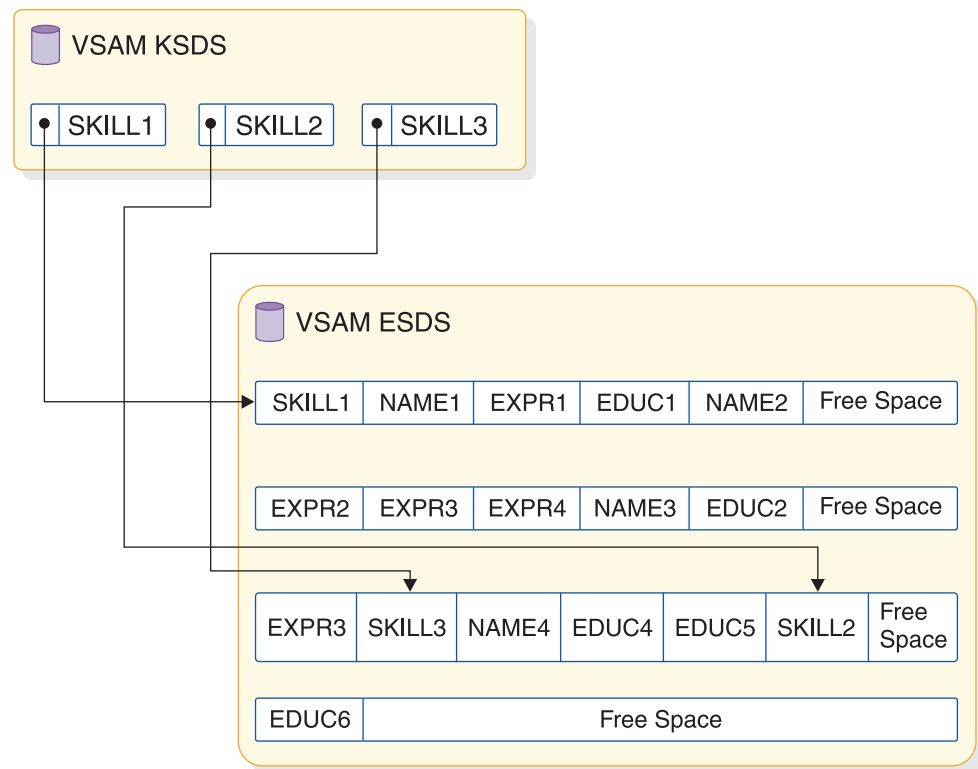
Figure 57. Inserting a root segment into a HIDAM or PHIDAM database

Related concepts:

"How the HD space search algorithm works" on page 177

Updating the space management fields when a root segment is inserted

When a root segment is inserted into an HD database, the space management fields need to be updated.

The following figure illustrates this process. The figure makes several assumptions so real values could be put in the space management fields. These assumptions are:

- The database is HDAM or PHDAM (and therefore has a root addressable area).

- VSAM is the access method; therefore there are CIs (not blocks) in the database. Because VSAM is used, each logical record has 7 bytes of control information.
- Logical records are 512 bytes long.
- One RAP exists in each CI.
- The root segment to be inserted (SKILL1) is 32 bytes long.

The "before" picture shows the CI containing the bitmap (in VSAM, the bitmap is always in the second CI in the database). The second bit in the bitmap is set to 1, which says there is free space in the next CI. In the next CI (CI #3):

- The FSEAP says there is an FSE (which describes an area of free space) 8 bytes from the beginning of this CI.
- The anchor point area (which has one RAP in this case) contains zeros because no root segments are currently stored in this CI.
- The FSE AL field says there is 497 bytes of free space available starting at the beginning of this FSE.

The SKILL1 root segment to be inserted is only 32 bytes long; therefore CI #3 has plenty of space to store SKILL1.

The "after" picture shows how the space management fields in CI #3 are updated when SKILL1 is inserted.

- The FSEAP now says there is an FSE 40 bytes from the beginning of this CI.
- The RAP points to SKILL1. The pointer value in the RAP is derived using the following formula:

  Pointer value = (CI size)*(CI number - 1) + Offset within the CI to the root segment

  In this case, the pointer value is 1032 (pointer value = 512 x 2 + 8).
- The FSE has been "moved" to the beginning of the remaining area of free space. The FSE AL field says there is 465 bytes (497 - 32) of free space available, starting at the beginning of this FSE.
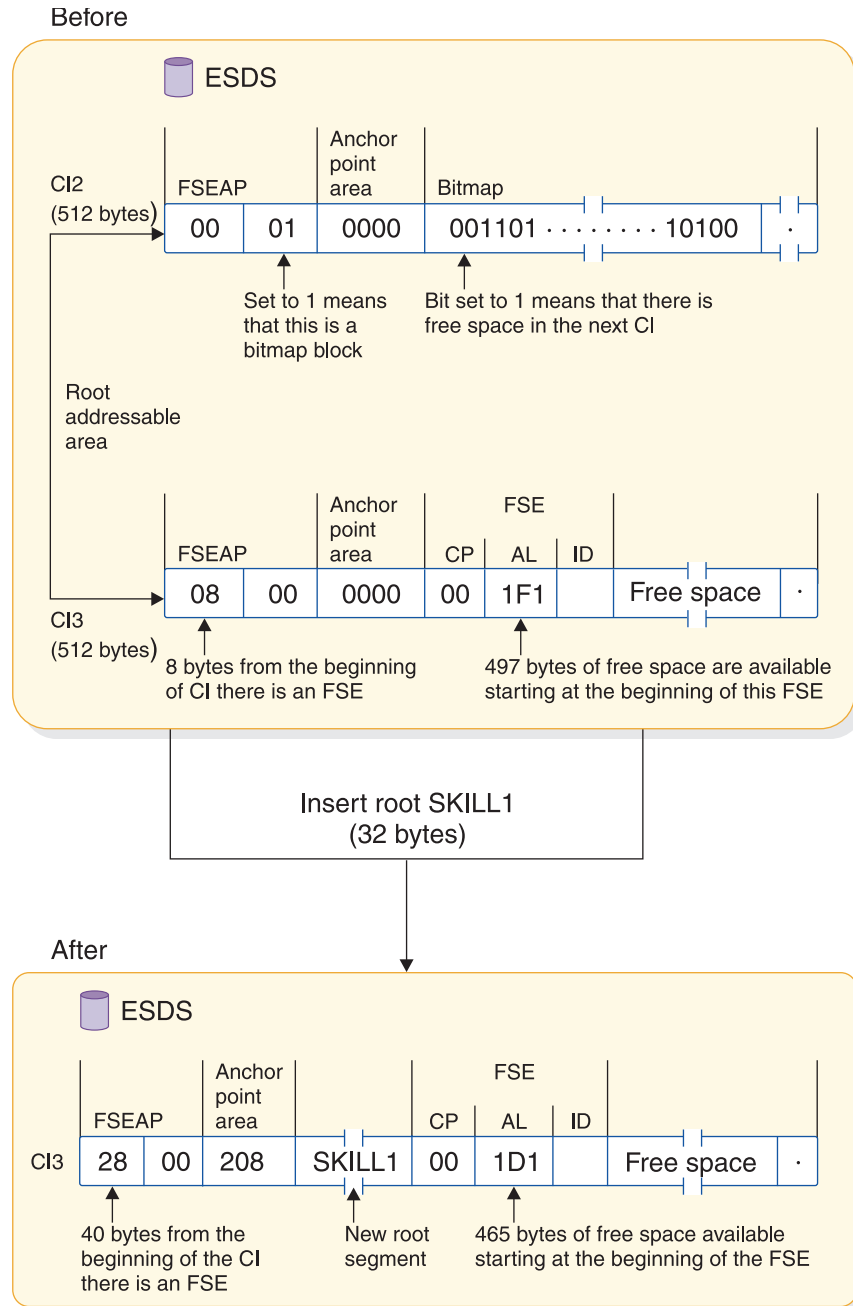
Figure 58. Updating the space management fields in an HDAM or PHDAM database

Related concepts:

"Deleting segments" on page 176

"Inserting dependent segments"

Inserting dependent segments

After initial load, dependent segments are inserted into HD databases using the HD space search algorithm.

As with the insertion of root segments into an HD database, the various space management fields in the database need to be updated.

Related concepts:

"How the HD space search algorithm works" on page 177

"Updating the space management fields when a root segment is inserted" on page 173

Deleting segments

When a segment is deleted in an HD database, it is physically removed from the database. The space it occupied can be reused when new segments are inserted.

As with the insertion of segments into an HD database, the various space management fields need to be updated.

- The bitmap needs to be updated if the block or CI from which the segment is deleted now contains enough space for a segment to be inserted. (Remember, the bitmap says whether enough space exists in the block or CI to hold a segment of the longest type defined. Thus, if the deleted segment did not free up enough space for the longest segment type defined, the bitmap is not changed.)
- The FSEAP needs to be updated to show where the first FSE in the block or CI is now located.
- When a segment is deleted, a new FSE might be created or the AL field value in the FSE that immediately precedes the deleted segment might need to be updated.
- If the deleted segment is a root segment in an HDAM or a PHDAM database, the value in its PTF pointer is put in the RAP or in the PTF pointer that pointed to it. This maintains the chain off the RAP and removes the deleted segment from the chain.

If a deleted segment is next to an already available area of space, the two areas are combined into one unless they are created by an online task that has not yet reached a sync point.

Related concepts:

"Updating the space management fields when a root segment is inserted" on page 173

Replacing segments

Replacing segments in HD databases is straightforward as long as fixed-length segments are used.

The segment data, once changed, is simply returned to its original location in storage. The key field in a segment cannot be replaced.

Provided sufficient adjacent space is available, the segment data is returned to its original location when a variable-length segment is replaced with a longer segment. If adjacent space is unavailable, space is obtained from the overflow area for the lengthened data portion of the segment. This segment is referred to as a "separated data segment." It has a 2-byte prefix consisting of a 1-byte segment code and a 1-byte delete flag, followed by the segment data. The delete byte of the separated data segment is set to X'FF', indicating that this is a separated data segment. A pointer is built immediately following the original segment to point to the separated data. Bit 4 of the delete byte of the original segment is set ON to indicate that the data for this segment is separated. The unused remaining space in the original segment is available for reuse.

How the HD space search algorithm works

The general rule for inserting a segment into an HD database is to store the segment (whether root or dependent) in the most desirable block or CI.

Related concepts:

"When not enough root storage room exists" on page 167

"Inserting dependent segments" on page 175

"Inserting root segments into a HIDAM or PHIDAM database" on page 173

Related tasks:

"Specifying free space (HDAM, PHDAM, HIDAM, and PHIDAM only)" on page 431

Root segment

The most desirable block depends on the access method.

For HDAM or PHDAM roots, the most desirable block is the one containing either the RAP or root segment that will point to the root being inserted. For HIDAM or PHIDAM roots, if the root does not have a twin backward pointer, the most desirable block is the one containing the root with the next higher key. If the root has a twin backward pointer, the most desirable block is the root with the next lower key.

Dependent segment

The most desirable block is the one containing the segment that points to the inserted segment.

If both physical child and physical twin pointers are used, the most desirable block is the one containing either the parent or the immediately-preceding twin. If hierarchical pointers are used, the most desirable block is the one containing the immediately-preceding segment in the hierarchy.

Second-most desirable block

When it is not possible to store one or more segments in the most desirable block because, for example, space is not available, the HD space search algorithm searches for the second-most desirable block or CI.

This search is done only if the block is in the buffer pool or contains free space according to the bitmap. The second-most desirable block or CI is a block or CI that was left free when the database was loaded or reorganized.

You can specify that every *n*th block or CI be left free. If you do not specify that every *n*th block or CI be left free, the HD space search algorithm does not search for the second-most desirable block or CI.

For HDAM or HIDAM databases, you can enter your free space specifications by using the FRSPC= keyword in the DATASET macro of the DBDGEN utility.

For PHDAM or PHIDAM databases, you can enter your free space specifications for each partition separately in the DBRC RECON data set by using either the HALDB Partition Definition utility or the FBFF(*value*) and FSPF(*value*) parameters of the DBRC batch commands INIT.PART or CHANGE.PART.

All search ranges defined in the HD space search algorithm, excluding steps 9 and 10, are limited to the physical extent that includes the most desirable block. When

the most desirable block is in the overflow area, the search ranges, excluding steps 9 and 10, are restricted to the overflow area.

The steps in the HD space search algorithm follow. They are arranged in the sequence in which they are performed. The first time any one of the steps in the list results in available space, the search is ended and the segment is stored.

The HD space search algorithm looks for space in the following order:
1. In the most desirable block (this block or CI is in the buffer pool).
2. In the second-most desirable block or CI.
3. In any block or CI in the buffer pool on the same cylinder.
4. In any block or CI on the same track, as determined by consulting the bitmap. (The bitmap says whether space is available for the longest segment type defined.)
5. In any block or CI on the same cylinder, as determined by consulting the bitmap.
6. In any block or CI in the buffer pool within plus or minus n cylinders. Specify n in the SCAN= keyword in the DATASET statement in the DBD.

   For HALDB databases, the value of the SCAN= keyword is always 0.
7. In any block or CI plus or minus n cylinders, as determined by consulting the bitmap.
8. In any block or CI in the buffer pool at the end of the data set.
9. In any block or CI at the end of the data set, as determined by consulting the bitmap. The data sets will be extended as far as possible before going to the next step.
10. In any block or CI in the data set where space exists, as determined by consulting the bitmap. (This step is not used when a HIDAM or PHIDAM database is loaded.)

Some of the above steps are skipped in load mode processing.

If the dependent segment being inserted is at the highest level in a secondary data set group, the place and the way in which space is found differ:
- First, if the segment has no twins, steps 1 through 8 in the HD space search algorithm are skipped.
- Second, if the segment has a twin that precedes it in the twin chain, the most desirable block is the one containing that twin.
- Third, if the segment has only twins that follow it in the twin chain, the most desirable block is the one containing the twin to which the new segment is chained.

Locking protocols

IMS uses locks to isolate the database changes made by concurrently executing programs.

Locking is accomplished by using either the Program Isolation (PI) lock manager or the IRLM. The PI lock manager provides only four locking levels and the IRLM supports eleven lock states.

The IRLM also provides support for "feedback only" and "test" locking required, and it supplies feedback on lock requests compatible with feedback supplied by the PI lock manager.

Locking to provide program isolation

For all database organizations, the basic item locked is the database record.

The database record is locked when position is first obtained in it. The item locked is the root segment, or for HDAM or PHDAM, the anchor point. Therefore, for HDAM or PHDAM, all database records chained from the anchor are locked. The processing option of the PCB determines whether or not two programs can concurrently access the same database record. If the processing option permits updates, then no other program can concurrently access the database record. The database record is locked until position is changed to a different database record or until the program reaches a commit point.

When a program updates a segment with an INSERT, DELETE, or REPLACE call, the segment, not the database record, is locked. On an INSERT or DELETE call, at least one other segment is altered and locked.

Because data is always accessed hierarchically, when a lock on a root (or anchor) is obtained, IMS determines if any programs hold locks on dependent segments. If no program holds locks on dependent segments, it is not necessary to lock dependent segments when they are accessed.

The following locking protocol allows IMS to make this determination. If a root segment is updated, the root lock is held at update level until commit. If a dependent segment is updated, it is locked at update level. When exiting the database record, the root segment is demoted to read level. When a program enters the database record and obtains the lock at either read or update level, the lock manager provides feedback indicating whether or not another program has the lock at read level. This determines if dependent segments will be locked when they are accessed. For HISAM, the primary logical record is treated as the root, and the overflow logical records are treated as dependent segments.

These lock protocols apply when the PI lock manager is used; however, if the IRLM is used, no lock is obtained when a dependent segment is updated. Instead, the root lock is held at single update level when exiting the database record. Therefore, no additional locks are required if a dependent segment is inserted, deleted, or replaced.

Related concepts:

"Deleting segments" on page 138

Locking for Q command codes

When a Q command code is issued for a root or dependent segment, a Q command code lock at share level is obtained for the segment. This lock is not released until a DEQ call with the same class is issued, or until commit time.

If a root segment is returned in hold status, the root lock obtained when entering the database record prevents another user with update capability from entering the database record. If a dependent segment is returned in hold status, a Q command code test lock is required. An indicator is turned on whenever a Q command code lock is issued for a database. This indicator is reset whenever the only application scheduled against the database ends. If the indicator is not set, then no Q command code locks are outstanding and no test lock is required to return a dependent segment in hold status.

Resource locking considerations with block level sharing

Resource locking can occur either locally in a non-sysplex environment or globally in a sysplex environment.

In a non-sysplex environment, local locks can be granted in one of three ways:

- **Immediately** because of one of the following reasons:

    IMS was able to get the required IRLM locks, and there is no other interest on this resource.

    The request is compatible with other holders or waiters.

- **Asynchronously** because the request could not get the required IRLM latches and was suspended. (This can also occur in a sysplex environment.) The lock is granted when latches become available and one of three conditions exist:

    No other holders exist.

    The request is compatible with other holders or waiters.

    The request is not compatible with the holders or waiters and was granted after their interest was released. (This could also occur in a sysplex environment.)

In a sysplex environment, global locks can be granted in one of three ways:

- **Locally by the IRLM** because of one of the following reasons:

    There is no other interest for this resource.

    This IRLM has the only interest, this request is compatible with the holders or waiters on this system, and XES already knows about the resource.

- **Synchronously on the XES CALL** because of one of the following reasons:

    XES shows no other interest for this resource.

    XES shows only SHARE interest for the hash class.

- **Asynchronously on the XES CALL** because of one of three conditions:

    Either XES shows EXCLUSIVE interest on the hash class by an IRLM, but the resource names do not match (FALSE CONTENTION by RMF™).

    Or XES shows EXCLUSIVE interest on the hash class by an IRLM and the resource names match, but the IRLM CONTENTION EXIT grants it anyway because the STATES are compatible (IRLM FALSE CONTENTION).

    Or the request is incompatible with the other HOLDERs and is granted by the CONTENTION Exit after their interest is released (IRLM REAL CONTENTION).

Data sharing impact on locking

When you use block-level data sharing, the IRLM must obtain the concurrence of the sharing system before granting global locks.

Root locks are global locks, and dependent segment locks are not. When you use block-level data sharing, locks prevent the sharing systems from concurrently updating the same buffer. The buffer is locked before making the update, and the lock is held until after the buffer is written during commit processing. No buffer locks are obtained when a buffer is read.

If a Q command code is issued on any segment, the buffer is locked. This prevents the sharing system from updating the buffer until the Q command code lock is released.

Locking in HDAM, PHDAM, HIDAM, and PHIDAM databases

If you access a HIDAM or PHIDAM root through the index, a lock is obtained on the index, using the RBA of the root segment as the resource name. Consequently, a single lock request locks both the index and the root.

When NOTWIN pointers are specified on a PHIDAM root, a lock on the next higher non-deleted root is required to provide data integrity. IMS obtains the additional lock by reading the index until a non-deleted index entry is found and then locking the RBA of the root segment as the resource name.

When you access an HDAM or a PHDAM database, the anchor of the desired root segment is locked as long as position exists on any root chained from that anchor. Therefore, if an update PCB has position on an HDAM or PHDAM root, no other user can access that anchor. When a segment has been updated and the IRLM is used, no other user can access the anchor until the user that is updating commits. If the PI lock manager is used and an uncommitted unit of work holds the anchor, locks are needed to access all root and dependent segments chained from the anchor until the user that is updating commits.

Locking for secondary indexes

When a secondary index is inserted, deleted or replaced, it is locked with a root segment lock.

When the secondary index is used to access the target of the secondary index, depending on what the index points to, it might be necessary to lock the secondary index.

Backup and recovery of HIDAM and PHIDAM primary indexes

The backup and recovery of HD primary indexes differs depending on whether the database is a HIDAM or a PHIDAM database.

You back up and recover HIDAM primary indexes in the same way as you do most other database data sets: by using image copies and database change records from the logs. Create image copies of the primary index data sets as often as you create image copies of the database data sets of the indexed HIDAM database. During recovery, after you restore the primary index from image copies, you apply the database change records from the logs by using the Database Recovery utility (DFSURDB0). If you do not create image copies of the primary index, your only recovery alternative is to rebuild the HIDAM primary index by using a separately priced index builder tool, such as the IBM IMS Index Builder for z/OS.

You do not back up or recover PHIDAM primary indexes. PHIDAM primary indexes are rebuilt after the recovery of the indexed database data sets by using the HALDB Index/ILDS Rebuild utility (DFSPREC0).

Related concepts:

Chapter 27, "Database backup and recovery," on page 589

Related reference:

Backup utilities (Database Utilities)

Recovery utilities (Database Utilities)

Partitions in PHDAM, PHIDAM, and PSINDEX databases

You can view a HALDB database, whether it is a PHDAM, PHIDAM, or a PSINDEX database, as a single database whose records are divided into manageable sections called partitions.

A partition is a subset of a HALDB database. The maximum size of a partition depends on the maximum number of data sets per partition and the maximum size of the data sets.

In many cases, each partition can be administered independently from the other partitions in the database. The division of records across partitions in a HALDB database is transparent to application programs, unless you choose to process partitions selectively.

HALDB partitions include unique features that allow IMS to manage them and that make them easier for you to use, as well. The HALDB partition structure also offers unique functions that are unavailable with non-partitioned databases, such as selective partition processing.

HALDB partition names and numbers

Each HALDB partition has a name, an ID number, a change version number, and a reorganization number. You define the partition name and IMS assigns and manages the numbers.

The partition ID numbers, change version numbers, and reorganization numbers are critical to the management of HALDB partitions by IMS and to the integrity of the data that the partitions contain.

In addition to being stored in each partition record in the RECON data set, these numbers are also stored in the HALDB master database record. IMS uses the numbers in the HALDB master database record to help manage the partitions and to access the data that the partitions contain.

If you delete a HALDB master database record, the partition numbers it contains are lost and the partitions associated with the deleted HALDB master database record can no longer be accessed. The partition numbers cannot be reproduced by redefining the HALDB master database record.

Deleting a HALDB master database record can also result in the loss of access to data in logically related HALDB databases, because the extended pointer set (EPS) of the logically related segments becomes invalid when the partition ID and reorganization number of the target segments are lost.

For these reasons, never delete a HALDB master database record from the RECON data set unless you are permanently deleting the HALDB database and all of its data, as well as all references to the HALDB database being deleted that are in any logically related or secondary index databases.

Related concepts:

"Partition definition control blocks and partition definitions in the RECON data set" on page 815

HALDB partition names
The HALDB *partition name* is a unique, 7-character alphanumeric identifier that you define and control.

Unless you delete the HALDB master database or the partition, the partition name does not change. The partition name does not necessarily correspond to the records that the partition contains.

Tip: If you want the partition names in a HALDB database to reflect the record keys that each partition contains, and if you want to keep the partition names in sequence over the life of the database, assign names to your partitions that provide room for new partitions to be added to the database without breaking the naming sequence.

For example, you could define the following partition names: ABC100, ABC200, ABC300, ABC400, ABC500, and so forth, to conform to the key range sequence of the records they contain. If partition ABC300 later becomes too large and you need to split its records by adding a new partition, you can name the new partition ABC250 without breaking the naming sequence.

You can specify a partition name instead of a master database name in many commands to restrict the command to the specified partition.

HALDB partition ID numbers

IMS assigns a *partition ID number* to each new partition when you define the partition.

IMS generates each new partition ID number by incrementing by one the last partition ID number assigned. Because you do not have control over the partition ID numbers, you cannot assume that the partition IDs in a database will stay in order.

For example, if you defined the partitions ABC100, ABC200, ABC300 in order, partition ABC100 would have partition ID number 1 and partition ABC300 would have partition ID number 3. Later, if you define a new partition ABC250, IMS assigns to it partition ID number 4.

HALDB change version numbers

IMS assigns a *change version number* to each partition and to the HALDB master database.

IMS uses the change version number to ensure that internal partition definition control blocks match the HALDB definitions that are stored in the RECON data set.

You do not have control over the change version numbers, although they are displayed in some reports that are generated by the DBRC LIST command. Every time you change the definitions of a HALDB partition, DBRC increments the change version numbers that are stored in the RECON data set for both the partition and the HALDB master database. When IMS detects that DBRC has incremented the number, IMS updates the control blocks of the HALDB database to reflect the new changes.

HALDB partition reorganization numbers

IMS assigns and maintains a *reorganization number* for each partition to ensure the integrity of data across database reorganizations.

IMS also uses the reorganization number in the HALDB self-healing pointer process after reorganizations of HALDB partitions that use either logical relationships or secondary indexes.

The reorganization number is stored in the following places in each partition:
- In the first block of the first database data set in each partition
- In the indirect list key (ILK) included in every segment in the partition
- In the extended pointer set (EPS) of each secondary index entry and each logical child segment
- In each indirect list entry (ILE) in the ILDS for each secondary index target segment and each logical parent segment

Attention: If the reorganization number of a partition becomes corrupted, future reorganizations or modifications of the partitions in the HALDB database might produce duplicate segment ILKs and data will be lost.

Reorganization numbers can become corrupt if the HALDB reorganization number verification function is not enabled and either a reorganization fails to increment the reorganization number of a partition correctly or a segment that has a low reorganization number in its EPS is moved into a partition and lowers the reorganization number of the destination partition.

A corrupt reorganization number is difficult to detect. If you do not use logical relationships or secondary indexes, a corrupt reorganization number does not cause any immediate problems. However, if you later add either logical relationships or secondary indexes to a HALDB database that has a corrupt reorganization number, you are likely to lose data.

To ensure the consistency of partition reorganization numbers, enable the HALDB reorganization number verification function. The HALDB reorganization number verification function records the reorganization number of each partition in the RECON data set and ensures that reorganization numbers are always incremented properly. When enabled, the HALDB reorganization number verification function applies to all HALDB databases that are recorded in the RECON data set.

To enable the HALDB reorganization number verification function:
- Issue either of the DBRC commands `INIT.RECON REORGV` or `CHANGE.RECON REORGV` or either of the type-1 commands `/RMINIT DBRC='RECON REORGV'` or `/RMCHANGE DBRC='RECON REORGV'`.
- Run a program that updates at least one record in each partition in each HALDB database that is registered in the RECON data set.

When you enable the HALDB reorganization number verification function, the reorganization numbers for all HALDB partitions in the RECON data set are reset to zero. Accessing a record in each partition updates the RECON data set with the current reorganization number that is stored in each partition.

Related concepts:

"Record distribution and high key partitioning" on page 813

"The HALDB self-healing pointer process" on page 691

⇨ Initializing and maintaining the RECON data sets (System Administration)

Related tasks:

"Changing the high key of a partition" on page 819

Related reference:

⇨ /RMxxxxxx commands (Commands)

⇨ DBRC commands (Commands)

HALDB partition initialization

After you define a partition and allocate its data sets, you must initialize the partition.

The initialization process makes a partition usable, but does not place any database segments in the partition. After the initialization process, a partition is empty.

To initialize HALDB partitions, you can use either the HALDB Partition Data Set Initialization utility (DFSUPNT0) or the Database Prereorganization utility (DFSURPR0).

Partition initialization writes the partition ID number and the initial reorganization number in PHDAM and PHIDAM partitions. The initial reorganization number is set to one, unless HALDB reorganization number verification is enabled, in which case the reorganization number is incremented by one from the existing reorganization number that is stored in the RECON data set.

The partition ID number and the reorganization numbers are written in the first 4 bytes of the first block of the first data set. This first block is called the *bitmap block*.

For PHDAM partitions, partition initialization writes and deletes a dummy record.

For PHIDAM partitions, partition initialization writes a high key record of all X'FF's in each partition.

For PSINDEX partitions, partition initialization writes and deletes a dummy record, which makes the high-used-RBA non-zero.

HALDB partition data sets

HALDB databases, regardless of type, can contain 1 to 1 001 partitions; however, the number of data sets per partition depends on the type of HALDB database and whether or not the integrated HALDB Online Reorganization function is used.

HALDB partitions contain the following types of data sets:

Database data sets
>   The database data sets contain the segment data for PHDAM and PHIDAM databases. Database data sets can be OSAM or VSAM entry sequenced data sets (ESDS).

Index data sets
>   Index data sets can be a primary index in a PHIDAM database or a secondary index data set in a PSINDEX partition. Index data sets are VSAM key sequenced data sets (KSDS).

Indirect list data set (ILDS)
>   ILDSs contain indirect list entries (ILE) that are used to manage pointers when logical relationships and PSINDEXes are used. ILDSs are VSAM KSDSs.

Number of data sets in a HALDB partition

The minimum and maximum number of data sets a HALDB partition can contain depends on the type of HALDB database and whether or not you use the integrated HALDB Online Reorganization function.

For the integrated HALDB Online Reorganization function, IMS creates an additional data set for each database data set and PHIDAM primary index data set

in the partition being reorganized. The additional data sets are used by the reorganization process and might or might not be active, or even present, depending on whether an online reorganization is currently in progress and, if a reorganization is not in progress, whether the inactive data sets were deleted after the last online reorganization completed.

The following table lists the minimum and maximum number of data sets a HALDB partition can contain.

Table 49. Minimum and maximum number of data sets for each HALDB partitions

HALDB type	Minimum number of data sets	Maximum number of data sets
PHDAM	Two or three: an OSAM or VSAM ESDS for the database data set, a KSDS for the ILDS, and, if the integrated HALDB Online Reorganization function is used, a second OSAM or VSAM ESDS.	Eleven or twenty one: ten OSAM or VSAM ESDSs for the database data sets, one KSDS for the ILDS, and, if the integrated HALDB Online Reorganization function is used, ten additional OSAM or VSAM ESDSs.
PHIDAM	Three or five: an OSAM or VSAM ESDS for the database data sets, a KSDS for the ILDS, a KSDS for the primary index, and, if the integrated HALDB Online Reorganization function is used, a second OSAM or VSAM ESDS and a second KSDS for the primary index.	Twelve or twenty three: ten OSAM or VSAM ESDSs for the database data sets, one KSDS for the ILDS, one KSDS for the primary index, and, if the integrated HALDB Online Reorganization function is used, ten additional OSAM or VSAM ESDSs and a second KSDS for the primary index.
PSINDEX	One: a KSDS	One: a KSDS

Indirect list data sets and HALDB partitions

Every HALDB PHDAM and PHIDAM partition that uses a secondary index or logical relationships must have an indirect list data set (ILDS) allocated to it.

The HALDB self-healing pointer process uses the ILDS to update secondary index pointers and logical relationship pointers after database reorganizations.

In a batch environment, even the partitions in a PHDAM or PHIDAM database that do not use secondary indexes or logical relationships must have an ILDS allocated.

In an online environment, IMS does not need to allocate an ILDS for partitions that do not use a secondary index or logical relationships.

Like all data sets in HALDB databases, the maximum size of an ILDS is 4 GB. Each ILE in an ILDS is 50 bytes. Consequently, an ILDS cannot support more that 85 000 000 logical parent segments or secondary index target segments in a single partition. It is very unlikely that you might reach the ILE limit, but if you do, you can split the single partition into two or more partitions.

When you convert a database to HALDB, reorganize the database data sets in a partition, or perform a recovery of the database data sets in a partition, the ILDS is updated or rebuilt to reflect the changes to the physical location of the target segments of the ILEs in the ILDS. The IMS utilities that can update or rebuild the ILDS are:
- The HD Reorganization Reload utility (DFSURGL0)

- The HALDB Index/ILDS Rebuild utility (DFSPREC0)

Both of these utilities provide options for rebuilding the ILDS by using either VSAM update mode or VSAM load mode. VSAM load mode, which adds the free space called for in the VSAM DEFINE statement that defines the ILDS, can improve the performance of both the current execution of the utility and of subsequent reorganizations and recoveries.

HALDB partition data sets and recovery

The recovery of HALDB databases is performed by recovering each partition. You recover the database data sets in each partition the same way you recover the database data sets in a non-HALDB database.

After the database data sets in the partition are recovered, you can then rebuild the primary index, if it exists, and the ILDS. HALDB primary indexes and ILDSs are not backed up or recovered.

To rebuild HALDB primary indexes and ILDSs, use the HALDB Index/ILDS Rebuild utility, which provides options for building the ILDS by using either VSAM update mode or VSAM load mode. VSAM load mode, which includes the free space called for in the VSAM KSDS DD statement that defines the ILDS, can improve the performance of both the current execution of the utility and of subsequent reorganizations and recoveries.

Related concepts:

Chapter 27, "Database backup and recovery," on page 589

HALDB partition selection

IMS must select the correct HALDB partition whenever it accesses a database record. The selection process is called *partition selection*.

Partition selection determines the partitions in which the root segments are placed and the order in which partitions are processed.

IMS performs partition selection by using either *key range partitioning*, which is based on the high root keys of the partitions, or by using a partition selection exit routine, which uses selection criteria that you define.

IMS assigns database records to partitions based on the key of the root segment.

For batch, BMP, and JBP programs, a PCB can be restricted to access one or more partitions.

Related concepts:

"HDAM, PHDAM, HIDAM, and PHIDAM databases" on page 144

Partition selection using high keys

If you use high-key partitioning, high keys define the partition boundaries and determine how the records are distributed across your partitions.

IMS performs partition selection based on the high key that is defined for each partition. The high key of a partition also defines the range of keys that the partition contains. IMS assigns a root segment to the partition with the lowest high key that is greater than or equal to the key of the root segment. For example, suppose that there are three partitions with high keys of 1000, 2000, and 3000. Root segment keys of 1001 through 2000 are in the partition with a high key of 2000.

The high keys of the partitions also define the order of the partitions within the HALDB database.

High-key partitioning is the simpler method to implement because you do not have to write an exit routine. You only need to assign a high key to each partition.

In PHIDAM and PSINDEX databases that use high-key partitioning, the records are in key sequence across the entire database, just as they are in HIDAM and non-HALDB secondary index databases. In PHIDAM or PSINDEX databases that use a partition selection exit routine, records are in key sequence within a partition, but not necessarily across partitions, which makes these databases inconsistent with HIDAM and non-HALDB secondary index databases. Application programs that require database records to be in key sequence across partitions do not work correctly when a partition selection exit routine is used.

Recommendation: When you use high-key partitioning, specify a high key value of all X'FF's for the partition that has the highest high key in the database. A high key value of all X'FF's ensures that all keys can be assigned to a partition. If the last partition (the partition with the highest key specified) has a key value other than all X'FF's, any attempt to access or insert a database record with a key higher than the high key specified results in an FM status code for the call. Application programs written for non-HALDB databases are unlikely to be able to process the FM status code.

Partition selection using a partition selection exit routine

If you need to select partitions by some criteria other than their high keys, you can use a partition selection exit routine.

IMS provides a sample HALDB Partition Selection exit routine (DFSPSE00), which assigns records to partitions based on the key of the root segment. The exit routine also determines the order in which sequential processing accesses the partitions. You can also write your own partition selection exit routine.

For a PHIDAM database, a partition selection exit routine can distribute the records in a key sequence within a partition that is out of sequence with the key sequences of the other partitions in the database. For example, a partition selection exit routine that uses the rightmost portion of a key to select the partition can conform to the characteristics of a HDAM database on data retrieval calls. Partition PARTA might include records in the following sequence: A001, B001, C001, D001. Partition PARTB might include records in this sequence: A010, B010, C010, D010. As in a HDAM database, a sequential retrieval call to find a segment with a key of C010 fails if partition PARTA is selected.

You can also use a partition selection exit routine to isolate certain database records by their characteristics. For example, if the sizes of most records in a PHDAM database are fairly uniform, except for a few records that are very large, the unusually large records can cause space usage problems within partitions. If the keys of the large records are known, an exit routine could recognize their keys and place them in a partition with different space characteristics. The partition might have many fewer records spread across the same amount of space or have its own specialized randomization routine.

The IBM IMS HALDB Conversion and Maintenance Aid for z/OS includes the IHCPSEL0 exit routine, which can perform this type of partition selection. If you

use the IHCPSEL0 exit routine, you do not need to write an exit routine. You need only to specify the part of the key that is to be used and the values for each partition.

You can find more information about the IBM IMS HALDB Conversion and Maintenance Aid for z/OS on the IBM DB2 and IMS Tools website at www.ibm.com/software/data/db2imstools.

How application programs process HALDB partitioned databases

Unless their processing is restricted, application programs process the data in partitioned HALDB databases in the same way that they process non-partitioned full-function databases, without regard to the partitions in which the data is stored.

Application programs that process data sequentially proceed across the partitions in a HALDB database in partition selection order. Application programs that process data randomly access the partitions in a HALDB database randomly as well. Application programs and the PCBs they use to access HALDB databases are not required to account for the partitions in the HALDB database that they access.

Note: The BMP application can have an unlimited number of databases and HALDB partitions without committing the changes.

HALDB selective partition processing

You can restrict BMP, JBP, and batch application programs to a single HALDB partition or a subset of HALDB partitions.

Restricting an application program to a subset of partitions allows multiple instances of the application program to process HALDB partitions in parallel, independently from the other application programs. The independent processing of partitions by application programs is similar to the independent processing of partitions by utilities.

To restrict processing to a subset of partitions, restrict the database PCB to the partition by specifying in a DFSHALDB DD statement the partition name and either the label name of the database PCB or the nth position of the database PCB.

For more information about the DFSHALDB DD statement, see *IMS Version 15 System Definition*.

Logical relationships and selective partition processing:

BMP, JBP, and batch-processing applications can selectively process a subset of one or more contiguous partitions that have logical relationships.

If a logical child is in a partition to which an application program's processing has been restricted and the logical parent is in another partition that the application does not have access to, the application can process the logical parent anyway. Because of a logical relationship, an application with restricted access can process a partition that it does not have direct access to.

Secondary indexes and selective partition processing:

You can restrict BMP, JBP, and batch-processing applications programs to a subset of one or more contiguous partitions of a HALDB partitioned secondary index (PSINDEX).

To specify selective partitions for processing, specify the name of the PSINDEX partition in the DFSHALDB statement

You can process the partitions of a PSINDEX selectively regardless of whether your application program processes your PSINDEX as a standalone database or as an alternate processing sequence for a PHDAM or PHIDAM database.

The partitions in a PSINDEX do not correspond to the partitions in the PHDAM or PHIDAM database that the PSINDEX indexes. Consequently, when you specify selective partition processing for a PSINDEX, selective partition processing applies only to the PSINDEX partition, not to the partitions of the PHDAM or PHIDAM database. The target segments can be in any partition in the indexed PHDAM or PHIDAM database.

Similarly, if you specify selective partition processing for a PHDAM or PHIDAM database, the selective partition processing does not restrict access to any of the partitions in any associated PSINDEXs.

Regardless of whether you are using selective partition processing with a PSINDEX or with an indexed PHDAM or PHIDAM database, selective partition processing does not affect the internal updating of a secondary index by IMS when a target segment is updated. For example, if an application program is restricted to a single partition of the PSINDEX and inserts a segment into the indexed PHDAM or PHIDAM database, the corresponding new index entry can be inserted in any partition of the PSINDEX.

Partition selection when processing is restricted to a single partition:

If you use high key partitioning, IMS selects partitions by using the root key that is used in the DL/I call and the high key that is defined for the partition. When access is restricted to a single partition and the root key is outside of the key range of the partition, IMS returns an FM or GE status code.

If you use a partition selection exit routine to select partitions, the routine is called when the DL/I call provides a specific root key. The exit routine selects a partition based on the specified root key. If the partition that is selected is different from the one that the application has access to, IMS returns an FM or GE status code.

When access is restricted to a single partition, the first partition is always the partition to which access is restricted, and the next partition does not exist. The exit routine is not called to select a first partition or the next partition.

Recommendation: When restricting processing to a single partition, include in the SSA only the root keys that are in the key range of the partition.

Examples of single partition processing:

The following examples illustrate the circumstances in which the FM, GE, and GB status codes are returned to an application program that is restricted to processing a single partition.

In all of the examples, the DB PCB usage is restricted to a HALDB partition that contains records with root keys 201 through 400.

GU rootkey=110

> The root key 110 is outside the range of root keys for the partition. IMS returns an FM status code.

GU rootkey=240 GN rootkey=110

> The processing moves forward from root key 240 to find a key that is equal to 110. Because 110 is lower than 240, IMS returns a GE status code.

GU rootkey=240 GN rootkey>=110

> The processing moves forward from root key 240 to find a key that is equal to or greater than 110. If a key is not found before reaching the end of the partition, IMS returns a GB status code.

GN rootkey>=110

> The processing attempts to start the search at key 110. Because the key is outside of the root key range of the partition, IMS returns an FM status code.

Examples of single partition processing of a PSINDEX:

The following examples illustrate the circumstances in which the FM, GE, and GB status codes are returned to an application program that is restricted to processing a single partition of a HALDB partitioned secondary index (PSINDEX).

In all of the examples, the DB PCB usage is restricted to a partition that contains records with secondary index keys 201 through 400. Partition 2 of the PSINDEX references multiple partitions in the indexed HALDB database.

GU xdfldkey=110

> The root key 110 is outside the range of root keys for the partition. IMS returns an FM status code.

GU xdfldkey=240 GN xdfldkey=110

> The processing moves forward from root key 240 to find a key that is equal to 110. Because 110 is lower than 240, IMS returns a GE status code.

GU xdfldkey=240 GN xdfldkey>=110

> The processing moves forward from root key 240 to find a key that is equal to or greater than 110. If the key is not found before reaching the end of the partition, IMS returns a GB status code.

GN xdfldkey>=110

> The processing attempts to start the search at key 110. Because the key is outside of the root key range of the partition, IMS returns an FM status code.

Partition selection when processing is restricted to a range of partitions:

A partition is selected by using the root key for the DL/I call and the high key that is defined for the partition.

When access is restricted to a range of consecutive partitions and the root key is outside the key range of any of the partitions, status code FM or GE is returned.

If you use a partition selection exit routine, the routine is called when the DL/I call provides a specific root key. The exit routine selects a partition based on the root

key given. If the partition selected is not one that the application has access to, status code FM or GE is returned. The exit routine is not called to select a first partition or next partition.

When access is restricted to a range of partitions, the first partition is always the partition named in the DFSHALDB statement and the next partition selected depends on the partition selection order as defined by either the partition high keys or the partition selection exit routine.

<u>Recommendation:</u> When you are restricting processing to a range of partitions, the SSA should include only the root keys that are in the key ranges of the partitions.

Examples of selectively processing a range of partitions:

For the following examples, the DB PCB usage is restricted to a range of three HALDB partitions: A, B, and C. The DFSHALDB statement specifies partition A and NUM=3.

The partitions contain the following root key ranges:
- Partition A contains the records with the root keys 201 through 400.
- Partition B contains the records with the root keys 401 through 600.
- Partition C contains the records with the root keys 601 through 800.

`GU rootkey=110`
> The root key 110 is outside of the range of root keys in the partitions. IMS returns an FM status code.

`GU rootkey=240 GN rootkey=110`
> The processing moves forward from root key 240 to find a key that is equal to 110. Because 110 is lower than 240, IMS returns a GE status code.

`GU rootkey=240 GN rootkey>=110`
> The processing moves forward from root key 240 to find a key that is equal to or greater than 110. If a key is not found before reaching the end of the partition, IMS returns a GB status code.

`GN rootkey>=110`
> The processing attempts to start the search at key 110. Because the key is outside of the root key range of the partitions, IMS returns an FM status code.

`GU rootkey=810`
> The root key 810 is outside of the range of root keys for the range of partitions. IMS returns an FM status code.

`GU rootkey=440 GN rootkey>=110`
> The processing moves forward from root key 440 to find a key that is equal to or greater than 110. If a key is not found before the end of the partition is reached, IMS returns a GB status code.

Parallel partition processing:

Using selective partition processing, different instances of your application programs can process different partitions of a database in parallel in both the batch environment and the online environment.

DBRC authorizes application programs to process individual partitions rather than the entire HALDB database. Processing partitions in parallel can reduce the time that is required for the processing.

In the batch environment, batch application programs are authorized to process individual partitions one at a time, rather than the entire HALDB database. IRLM is not required.

In the online environment, multiple dependent regions can process records in the same or in different partitions. Data sharing is not required.

If you use block-level data sharing, you can easily process different partitions in parallel with multiple subsystems. The subsystems can be online systems or batch jobs. To use block-level data sharing, you must use IRLM and you must register the databases in DBRC as allowing block level data sharing. For more information about block-level data sharing, see the DBRC information and the data sharing information in *IMS Version 15 System Administration*.

To enable multiple instances of an application program to process partitions in parallel, restrict the database PCB of each instance to the partitions by specifying in a DFSHALDB DD statement the partition name and either the label name of the database PCB or the *n*th position of the database PCB.

You might also need to make one or more of the following modifications to the input, output, or processing logic of the application program:
- Split the application program input to feed multiple instances of the application program.
- Consolidate the output of multiple instances of the application program.
- Modify the application program to respond correctly to the unavailability of the partitions it cannot access.

For more information about the DFSHALDB DD statement, see *IMS Version 15 System Definition*.

IMS utilities supported by HALDB

IMS provides several utilities developed specifically to support HALDB partitioned databases. HALDB partitioned databases also support many of the same utilities supported by other full-function database types.

Database Recovery Control (DBRC) is required for execution of any utility operating on a HALDB database. Each utility checks for the presence of DBRC. If DBRC is not present, the utility issues an error message and terminates.

Image copy utilities reject any attempt to image copy HALDB ILDSs or PHIDAM primary index data sets. Recovery utilities reject any attempt to recover HALDB ILDSs or PHIDAM primary index data sets. Both image copy and recovery utilities can run only against a particular data set of a HALDB partition.

The following table lists all of the database utilities that can be used with HALDB databases.

Table 50. Utilities that can run against HALDB databases

Utility	Description	Comment
DFSMAID0	HALDB Migration Aid	

Table 50. Utilities that can run against HALDB databases (continued)

Utility	Description	Comment
DFSPREC0	HALDB Index/ILDS Rebuild	
DFSUPNT0	HALDB Partition Data Set Initialization	
%DFSHALDB	HALDB Partition Definition	Invocation of the utility by a c-list. %DFSHALDB is the TSO invocation of module DSPXPDDU.
DFSURUL0	HISAM Reorganization Unload	
DFSURRL0	HISAM Reorganization Reload	
DFSURGU0	HD Reorganization Unload	Applies to PHDAM, PHIDAM, and PSINDEX
DFSURGL0	HD Reorganization Reload	Applies to PHDAM, PHIDAM, and PSINDEX
DFSURPR0	Prereorganization	
DFSUDMP0	Image Copy	
DFSUICP0	Online Image Copy	
DFSUDMT0	Database Image Copy 2	
DFSUCUM0	Change Accumulation	
DFSURDB0	Database Recovery	
DFSBBO00	Batch Backout	

Database I/O error management

When a database I/O error occurs, IMS copies the buffer contents of the error block/control interval (CI) to a virtual buffer. A subsequent DL/I request causes the error block/CI to be read back into the buffer pool.

The write error information and buffers are maintained across restarts, deferring recovery to a convenient time. I/O error retry is automatically performed at database close time. If the retry operation is successful, the error condition no longer exists and recovery is not needed.

When a database I/O error occurs in a sysplex environment, the local system maintains the buffer and informs all members of the data-sharing group with registered interest in the database that the CI is unavailable. Subsequent DL/I requests for that CI receive a failure return code as long as the I/O error persists.

Although you do not have to register your databases with DBRC in order for error handling to work, registration is required for HALDB databases and highly recommended for all other types of full-function databases.

The integrated HALDB Online Reorganization function can help eliminate the HALDB I/O errors on sharing systems. If an online reorganization is started on the system that owns the write error EEQE, the online reorganization function can take the local copy of the buffer and write it out to the output data sets. After the buffer is written to the output data sets, the updates in the buffer are available to all sharing systems again.

Attention: If an error occurs on a database registered with DBRC and the system stops, the database could be damaged if the system is restarted and a /DBR command is not issued prior to accessing the database. The restart causes the error buffers to be restored as they were when the system stopped. If the same block had been updated during the batch run, the batch update would be overlaid.

Chapter 16. Fast Path database types

Fast Path databases include data entry databases (DEDBs) and main storage databases (MSDBs). DEDBs provide efficient storage for and access to large volumes of data. DEDBs also provide a high level of availability to that data. MSDBs store and provide access to an installation's most frequently used data.

Both DEDBs and MSDBs use the direct method of storing data. With the direct method, the hierarchical sequence of segments is maintained by putting direct-address pointers in each segment's prefix.

Each IMS environment supports Fast Path databases as follows:
- DB/DC supports both DEDBs and MSDBs.
- DBCTL supports DEDBs, but does not support MSDBs.
- DCCTL does not support DEDBs or MSDBs.

Related concepts:

Chapter 14, "Summary of IMS database types and functions," on page 115

Data entry databases

Data entry databases (DEDBs) provide efficient storage for and access to large volumes of data. DEDBs also provide a high level of availability of that data.

Several characteristics of DEDBs also make DEDBs useful when you must gather detailed and summary information. These characteristics include:

Area format

Area data set replication

Record deactivation

Non-recovery option

A DEDB is a hierarchical database that contains up to 127 segment types: a root segment, an optional sequential dependent segment, and 0 to 126 direct dependent segments. If an optional sequential dependent segment is defined, you can define no more than 125 direct dependent segments. A DEDB structure can have as many as 15 hierarchical levels. Instances of sequential dependent segments for an area are stored in chronological order, regardless of the root on which they are dependent. Direct dependent segments are stored hierarchically, which allows for rapid retrieval.

Recommendation: Because ETO terminals cannot access terminal-related MSDBs, you should develop any new Fast Path databases as DEDBs instead of as MSDBs. You should also consider converting any of your existing non-terminal-related MSDBs with non-terminal-related keys to VSO DEDBs. You can use the MSDB-to-DEDB Conversion utility to do so.

Related concepts:

Chapter 15, "Full-function database types," on page 117

"Performance considerations overview" on page 119

"The segment" on page 15

Related reference:

➡ MSDB-to-DEDB Conversion utility (DBFUCDB0) (Database Utilities)

DEDB functions

DEDBs and MSDBs have many similar functions.

The common functions include:
- Virtual storage
- The field (FLD) call
- Fixed length segments
- MSDB or DEDB commit view

In addition, DEDBs have the following functions and support:
- Full DBRC support
- Block-level sharing of areas available to
  - DBCTL
  - LU 6.2 applications
  - DB/DC applications
- HSSP support
- DEDB utilities
- Online database maintenance
- A full hierarchical model, including support of INSERT and DELETE calls
- A randomizer search technique
- Secondary index support

DEDB areas

A DEDB can be organized into one or more data sets called areas. Areas increase the efficiency, capacity, and flexibility of DEDBs. This topic discusses DEDB areas and how to work with them.

Areas and the DEDB format

A DEDB can use multiple data sets, called areas, with each area containing the entire data structure.

The physical format of DEDBs makes the data they contain more readily available. In a hierarchical IMS database that does not use areas, the logical data structure is spread across the entire database. If multiple data sets are used, the data structure is broken up on a segment basis.

Each area in a DEDB is a VSAM data set. A DEDB record (a root and its dependent segments) does not span areas. A DEDB can be divided into as many as 2048 such areas. This organization is transparent to the application program.

The maximum size of a DEDB area is 4 GB. The maximum number of areas per database is 2048, thus the maximum size of a DEDB database is 8 796 093 020 160 bytes (8 TB).

IMS does not enforce a limit on the number of area data sets that can be open at the same time by multiple DEDB databases. However, the resources available at your installation and the consumption of those resources by both your IMS

configuration and the other z/OS® subsystems that your installation might be running, such as Db2 for z/OS, could potentially limit the number of area data sets that you can open.

For area data sets, one of the resources that could become constrained with a very large number of open data sets is storage in the extended common service area (ECSA) and the extended private storage (EPVT).

The randomizing module is used to determine which records are placed in each area. Because of the area concept, larger databases can exceed the limitation of 2^{32} bytes for a single VSAM data set. Each area can have its own space management parameters. You can choose these parameters according to the message volume, which can vary from area to area. DEDB areas can be allocated on different volume types.

Initialization, reorganization, and recovery of DEDBs are done on an area basis. Resource allocation is done at the control interval (CI) level. Multiple programs, optionally together with one online utility, can access an area concurrently within a database, as long as they are using different CIs. CI sizes can be 512 bytes, 1 K, 2 K, 4 K, and up to 28 K in 4 K increments. The media manager and Integrated Catalog Facility catalog of Data Facility Storage Management Subsystem (DFSMS) are required.

Related concepts:

"Enqueue level of segment CIs" on page 212

Opening and preopening DEDB areas

By default, IMS does not open a DEDB area until an eligible application accesses the area.

Although this prevents unneeded areas from being opened at startup, the first application that accesses a DEDB area incurs some additional processing overhead. Multiple calls to multiple areas immediately following a startup process can increase this burden significantly.

You can limit the overhead of opening areas by preopening your DEDB areas. You can also distribute this overhead between the startup process and online operation by preopening only those areas that applications use the most and by leaving all other areas closed until an application first accesses them.

You specify the preopen status of an area using the PREOPEN and NOPREO parameters of the DBRC commands INIT.DBDS or CHANGE.DBDS.

By default IMS preopens all DEDB areas that have been assigned preopen status during the startup process; however, preopening a large number of DEDB areas during the startup process can delay data processing. To avoid this delay, you can have IMS preopen DEDB areas after the startup process and asynchronously to the execution of your application programs. In this case, if IMS has not preopened a DEDB area when an application program attempts to access the area, IMS opens the DEDB area at that time. You can specify this behavior by using the FPOPN= keyword in the IMS and DBC startup procedures. Specifically, FPOPN=P causes IMS to preopen DEDB areas after startup and asynchronous to application program execution.

The FPOPN= keyword determines how IMS reopens DEDB areas for both normal restarts (/NRE) and emergency restarts (/ERE).

DEDB areas can also be opened by issuing either of the following type-2 commands with the OPTION(OPEN) keyword:

- UPDATE AREA NAME(*areaname*) START(ACCESS) OPTION(OPEN)
- UPDATE DB NAME(*dedbname*) AREA(*) START(ACCESS) OPTION(OPEN)

Note: The OPTION(OPEN) process is not logged for either the UPDATE AREA command or the UPDATE DB command. If IMS is restarted after using this option, IMS does not automatically re-open DEDB areas that were previously opened by using these UPDATE commands.

Related reference:

➦ Parameter descriptions for IMS procedures (System Definition)

Reopening DEDB areas during an emergency restart:

You can specify how IMS reopens DEDB areas during an emergency restart by using the FPOPN= keyword in the IMS procedure or DBC procedure.

The following list describes how the FPOPN= keyword affects the reopening of DEDB areas during an emergency restart:

FPOPN=N
> During the startup process, IMS opens only those areas that have preopen status. This is the default.

FPOPN=P
> After the startup process completes and asynchronous to the resumption of application processing, IMS opens only those areas that have preopen status.

FPOPN=R
> After the startup process completes and asynchronous to the resumption of application processing, IMS opens only those areas that were open prior to the abnormal termination. All DEDB areas that were closed at the time of the abnormal termination, including DEDB areas with a preopen status, will remain closed when you restart IMS.

FPOPN=D
> Suppresses the preopen process. DEDB areas that have a preopen status are not preopened and remain closed until they are first accessed by an application program or until they are manually opened with a /START AREA command.
>
> FPOPN=D overrides, but does not change, the preopen status of DEDB areas as set by the PREOPEN parameter of the DBRC commands INIT.DBDS and CHANGE.DBDS.

Related concepts:

"Emergency restart processing" on page 240

Related reference:

➦ Parameter descriptions for IMS procedures (System Definition)

➦ DBC procedure (System Definition)

➦ IMS procedure (System Definition)

Stopping DEDBs and DEDB areas

You can stop access to a DEDB or stop the scheduling of application programs against a DEDB at the database level or the area level by issuing the appropriate command.

The database-level commands include the type-1 commands /STOP DB and /DBRECOVERY DB and the type-2 command UPDATE DB STOP(ACCESS|SCHD).

The area-level commands include the type-1 commands /STOP AREA and /DBRECOVERY AREA and the type-2 command UPDATE AREA STOP(ACCESS|SCHD).

The type-1 command /STOP DB and the type-2 command UPDATE DB STOP(SCHD) have an equivalent effect in that they both stop the scheduling of new application programs against the DEDB. The commands /DBRECOVERY DB and UPDATE DB STOP(ACCESS) both stop all access to the DEDB. The area-level type-1 and type-2 commands have similar equivalencies.

Starting DEDBs and DEDB areas

You can start access to a DEDB or start the scheduling of application programs against a DEDB at the database level or the area level.

The database-level commands include the type-1 command /START DB and the type-2 command UPDATE DB START(ACCESS).

The area-level commands include the type-1 command /START AREA and the type-2 command UPDATE AREA START(ACCESS). The /START AREA command does not open areas unless you have specified them as PREOPEN or PRELOAD areas.

You can start all areas of a DEDB at once by using the AREA(*) parameter of the type-2 command UPDATE DB START(ACCESS). The AREA(*) parameter is useful if you have stopped access to a DEDB at the database-level by issuing the type-1 command /DBRECOVERY DB or the type-2 command UPDATE DB STOP(ACCESS). Note that specifying an area name, for example AREA(*area_name*), is invalid.

You can use the AREA(*) parameter with the SET(ACCESS) parameter of the type-2 command UPDATE DB START(ACCESS) to start all areas at once and to change the access type for the DEDB at the same time.

You can also open DEDB areas when you start them by specifying the OPTION(OPEN) keyword on the either of the type-2 commands UPDATE AREA START(ACCESS) or UPDATE DB START(ACCESS).

Restarting and reopening areas after an IRLM failure

The internal resource lock manager (IRLM) ensures the integrity of databases in a data sharing environment.

To avoid compromising the integrity of the data in DEDB areas when an IRLM fails, all DEDB areas under the control of the failed IRLM are stopped. After you correct the failure and reconnect IRLM to the IMS system, you must restart and reopen the DEDB areas that the IRLM controls.

You can specify how IMS restarts and reopens DEDB areas after the IRLM reconnects, by using the FPRLM= keyword in the IMS and DBC procedures.

Related concepts:

➦ Using IRLM with database-level sharing (System Administration)

➦ Recovery involving IRLM (System Administration)

↪ Restart after IMS failure (System Administration)

↪ IRLM failures (Operations and Automation)

Related reference:

↪ DBC procedure (System Definition)

↪ Parameter descriptions for IMS procedures (System Definition)

↪ IMS procedure (System Definition)

Read and write errors in DEDB areas

This topic describes how IMS handles read and write errors that occur in DEDB areas.

Read error:

When a read error is detected in an area, the application program receives an AO status code.

An Error Queue Element (EQE) is created, but not written to the second CI nor sent to the sharing system in a data sharing environment. Application programs can continue to access that area; they are prevented only from accessing the CI in error. After read errors on four different CIs, the area data set (ADS) is stopped. The read errors must be consecutive; that is, if there is an intervening write error, the read EQE count is cleared. This read error processing only applies to a multiple area data set (MADS) environment.

Write error:

When a write error is detected in an area, an EQE is created and application programs are allowed access to the area until the EQE count reaches 11.

Even though part of a database might not be available (one or more areas are stopped), the database is still logically available and transactions using that database are still scheduled. If multiple data sets make up the area, chances are that one copy of the data will always be available.

If your DEDB is nonrecoverable, write errors are handled differently, compared to recoverable DEDBs. When there is a write error in an area, an EQE is created. When there are 10 EQEs for an area, DBRC marks it "Recovery Needed" and IMS stops the area. If the area is shared, then all IMS systems in the sharing group are notified and they also stop the area. When a DEDB is marked "Recovery Needed", you must restore it, such as from an image copy. Incorporate this recovery procedure into your operational procedures.

When a write error occurs to a DEDB using MADS, an EQE is created for the ADS that had the write error. In this environment, when the maximum of 10 EQEs is reached, the ADS is stopped.

When a write error to a recoverable DEDB area using a single ADS occurs, IMS invokes the I/O toleration (IOT) processing. IMS allocates a virtual buffer in ECSA and copies the control interval in error from the Fast Path common buffer to the virtual buffer. IMS records the creation of the virtual buffer with an X'26' log record. If the database is registered with DBRC, an Extended Error Queue Element

(EEQE) is created and registered in DBRC. The EEQE identifies the control interval in error. In a data sharing environment using IRLM, all sharing partners are notified of the creation of the EEQE.

The data that is tolerated is available to the IMS system that created the EEQE. The sharing partner will get an 'AO' status when it requests that CI because the data is not available. When a request is made for a control interval that is tolerated, the data is copied from the virtual buffer to a common buffer. When an update is performed on the data, it is copied back to the virtual buffer. A standard X'5950' log record is generated for the update.

Every write error is represented by an EEQE on an area basis. The EEQEs are maintained by DBRC and logged to the IMS log as X'26' log records. There is no logical limit to the number of EEQEs that can exist for an area. There is a physical storage limitation in DBRC and ECSA for the number of EEQEs that can be maintained. This limit is installation dependent. To make sure that we do not overextend DBRC or ECSA usage, a limited number of EEQEs are allowed for a DEDB. The limit is 100. After 100 EEQEs are created for an area, the area is stopped.

During system checkpoint, /STO, and /VUN commands, IMS attempts to write back the CIs in error. If the write is successful, the EEQE is removed. If the write is unsuccessful, the EEQE remains.

Related concepts:

"Non-recovery option"

Record deactivation

If an error occurs while an application program is updating a DEDB, it is not necessary to stop the database or even the area. IMS continues to allow application programs to access that area.

IMS only prevents the application programs from accessing the control interval in error by creating an EQE for the error CI. If there are multiple copies of the area, chances are that one copy of the data will always be available. It is unlikely that the same control interval will be in error in all copies of the area. IMS automatically makes an area data set unavailable when a count of 11 errors has been reached for that data set.

Record deactivation minimizes the effect of database failure and errors to the data in these ways:

- If multiple copies of an area data set are used, and an error occurs while an application program is trying to update that area, the error does not need to be corrected immediately. Other application programs can continue to access the data in that area through other available copies of that area.
- If a copy of an area has a number of I/O errors, you can create a new copy from existing copies of the area using the DEDB Area Data Set Create utility. The copy with the errors can then be destroyed.

Non-recovery option

By specifying a VSO or non-VSO DEDB as nonrecoverable, you can improve online performance and reduce database change logging of your DEDBs.

IMS does not log any changes from a nonrecoverable DEDB, nor does it keep any updates in the DBRC RECON data set. All areas are nonrecoverable in a nonrecoverable DEDB.

Unlike full-function nonrecoverable databases, which support backout, nonrecoverable DEDBs are truly nonrecoverable and cannot REDO during restart or XRF takeover. IMS writes a single log record, X'5951', once for every area at each sync point to indicate that nonrecoverable suppression has taken place.

The X'5951' log and DMAC flags determine the integrity of an area during an emergency restart or XRF takeover. Nonrecoverable DEDB write errors can happen during restart or XRF takeover. If there are errors found in a nonrecoverable DEDB during an XRF takeover or an emergency restart, message DFS3711W is issued and the DEDB is not stopped.

Nonrecoverable DEDBs must register with DBRC. To define a DEDB as nonrecoverable, use the command `INIT.DB DBD() TYPEFP NONRECOV`. The default is RECOVABL for recoverable DEDB.

Before changing the recoverability of a DEDB, issue a `/STOP DB`, `/STO AREA`, or `/DBR DB` command. To change a recoverable DEDB to a nonrecoverable DEDB, use the DBRC command `CHANGE.DB DBD() NONRECOV`. To change nonrecoverable DEDB to a recoverable DEDB, use the command `CHANGE.DB DBD() RECOVABL`.

To restore a nonrecoverable DEDB, use the `GENJCL.RECOV RESTORE` command. The recovery utility restores the database to the last image copy taken. If the DEDB had been changed from a recoverable DEDB to a nonrecoverable DEDB, the recovery utility will apply any updates from the logs up to the point when the change was made (if no image copy was made after the change to nonrecoverable).

Related concepts:

"Write error" on page 202

"Fast Path log reduction" on page 638

Area data set replication

A data set can be copied, or replicated, up to seven times, increasing the availability of the data to application programs.

The DEDB Area Data Set Create utility (DBFUMRI0) produces one or more copies of a data set representing the area without stopping the area. All copies of an area data set must have identical CI sizes and spaces but can reside on different devices. The utility uses all the current copies to complete its new data set, proceeding to another copy if it detects an I/O error for a particular record. In this way, a clean copy is constructed from the aggregate of the available data. Current updates to the new data set take effect immediately.

The Create utility can create its new copy on a different device, as specified in its job control language (JCL). If your installation was migrating data to other storage devices, then this process could be carried out while the online system was still executing, and the data would remain current.

To ensure all copies of a DEDB remain identical, IMS updates all copies when a change is made to only one copy.

If an ADS fails open during normal open processing of a DEDB with multiple data sets (MADS), none of the copies of the ADS can be allocated, and the area is stopped. However, when open failure occurs during emergency restart, only the failed ADS is unallocated and stopped. The other copies of the ADS remain available for use.

DEDBs and data sharing

You can specify different levels of data sharing for DEDBs. The specifications you make for a DEDB apply to all the areas in the DEDB.

If you specify that a DEDB does not allow data sharing, only one IMS system can access a DEDB area at a time; however, other IMS systems can still access the other areas in the DEDB.

If you specify that a DEDB allows data sharing, multiple IMS systems can access the same DEDB area at the same time. Sharing a single DEDB area is equivalent to block-level sharing of full-function databases.

You can specify the level of data sharing that a DEDB allows by using the SHARELVL parameter in the DBRC commands INIT.DB and CHANGE.DB. If any IMS has already authorized the database, changing the SHARELVL does not modify the database record. The SHARELVL parameter applies to all areas in a DEDB.

You can share DEDB areas directly from DASD or from a coupling facility structure using the Virtual Storage Option (VSO).

Related concepts:

"Fast Path Virtual Storage Option" on page 225

"Sharing of VSO DEDB areas" on page 229

Data sharing in IMS environments (System Administration)

Related reference:

DBRC commands (Commands)

Fixed- and variable-length segments in DEDBs

DEDBs support fixed-length segments. Thus you can define fixed-length or variable-length segments for your DEDBs. This support allows you to use MSDB applications for your DEDBs.

To define fixed-length segments, specify a single value for the BYTES= parameter during DBDGEN in the SEGM macro. To define variable-length segments, specify two values for the BYTES= parameter during DBDGEN in the SEGM macro.

Application programs for fixed-length-segment DEDBs, like MSDBs, do not see the length (LL) field at the beginning of each segment. Application programs for variable-length-segment DEDBs do see the length (LL) field at the beginning of each segment, and must use it to process the segment properly.

Fixed-length-segment application programs using REPL and ISRT calls can omit the length (LL) field.

Examples of defining segments

The following examples show how to use the BYTES= parameter to define variable-length or fixed-length segments.

Defining a variable-length segment

```
ROOTSEG  SEGM  NAME=ROOTSEG1,                                              C
               PARENT=0,                                                   C
               BYTES=(390,20)
```

Defining a fixed-length segment

```
ROOTSEG  SEGM  NAME=ROOTSEG1,                                              C
              PARENT=0,                                                    C
              BYTES=(320)
```

Parts of a DEDB area

A DEDB area consists of three parts.

The parts are:
- Root addressable part
- Independent overflow part
- Sequential dependent part

The following figure shows these parts of a DEDB area.
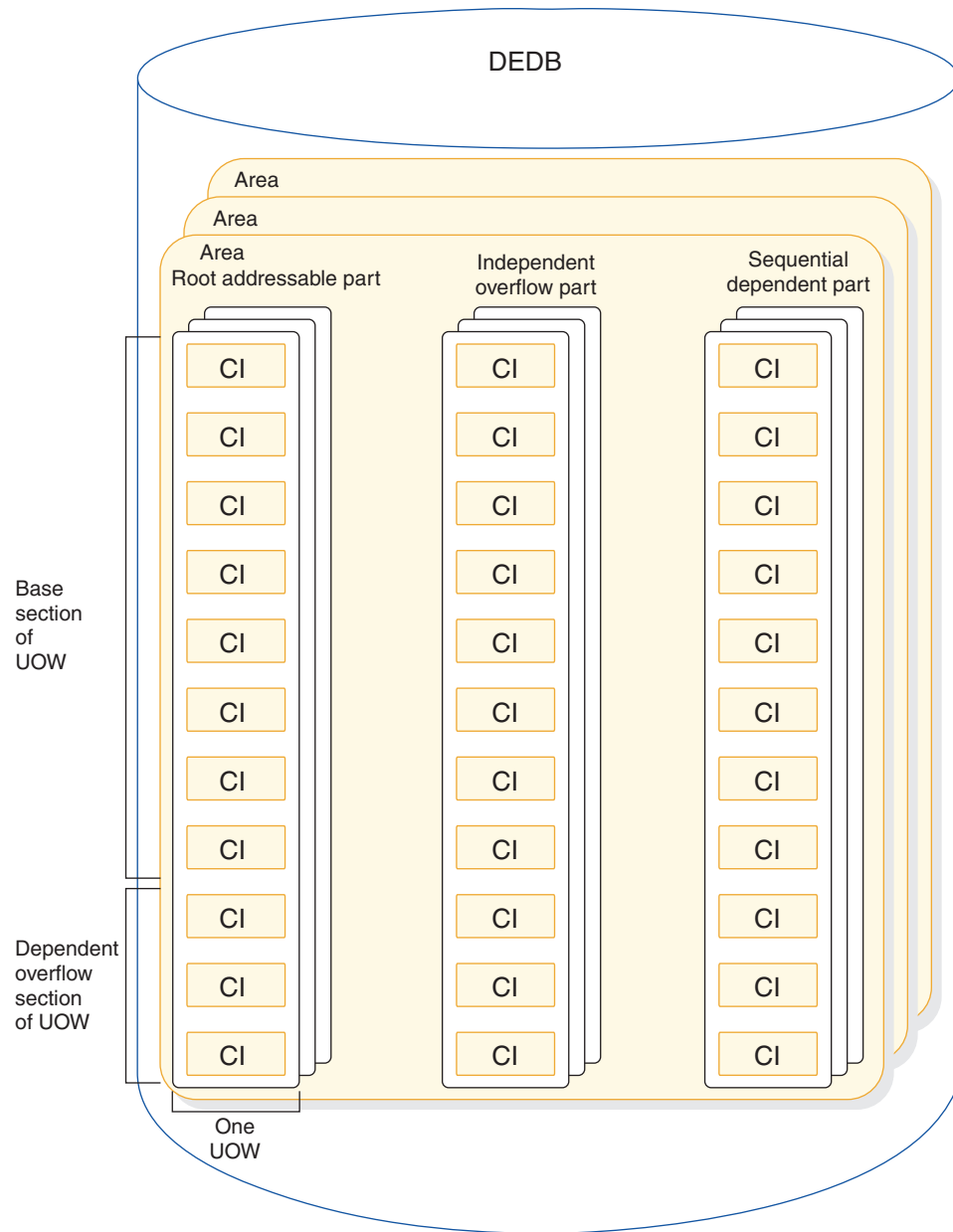
Figure 59. Parts of a DEDB area in storage

When a DEDB data set is initialized by the DEDB initialization utility (DBFUMIN0), additional CIs are created for internal use, so the DEDB area will actually contain more CIs than are shown in the preceding figure. These extra CIs are referred to as the Reorganization UOW. Although IMS does not use the extra CIs, DBFUMIN0 creates them for compatibility purposes.

Root addressable part

The root addressable part is divided into units-of-work (UOW), which are the basic elements of space allocation.

A UOW consists of a user-specified number of CIs located physically contiguous.

Each UOW in the root addressable part is further divided into a base section and an overflow section. The base section contains CIs of a UOW that are addressed by

the randomizing module, whereas the overflow section of the UOW is used as a logical extension of a CI within that UOW.

Root and direct dependent segments are stored in the base section. Both can be stored in the overflow section if the base section is full.

Independent overflow part

The independent overflow part contains empty CIs that can be used by any UOW in the area.

When a UOW gets a CI from the independent overflow part, the CI can be used only by that UOW. A CI in the independent overflow part can be considered an extension of the overflow section in the root addressable part as soon as it is allocated to a UOW. The independent overflow CI remains allocated to a specific UOW unless, after a reorganization, it is no longer required, at which time it is freed.

Sequential dependent part

The sequential dependent part holds sequential dependent segments from roots in all UOWs in the area.

Sequential dependent segments are stored in chronological order without regard to the root or UOW that contains the root. When the sequential dependent part is full, it is reused from the beginning. However, before the sequential dependent part can be reused, you must use the DEDB Sequential Dependent Delete utility (DBFUMDL0) to delete a contiguous portion or all the sequential dependent segments in that part.

CI and segment formats

The format of DEDB control intervals (CIs) and segments are shown in the following tables and figures.

This topic contains Diagnosis, Modification, and Tuning information.

The following series of diagrams show the following formats:
- CI format
- Root segment format
- Sequential dependent segment format
- Direct dependent segment format

The tables that follow each figure describe the sections of the CI and segments in the order that the sections appear in the graphic.

FSE AP	CI TYP	RAP	Segments and FSEs	CUSN	RBA	RDF	CIDF

CI prefix   CI suffix
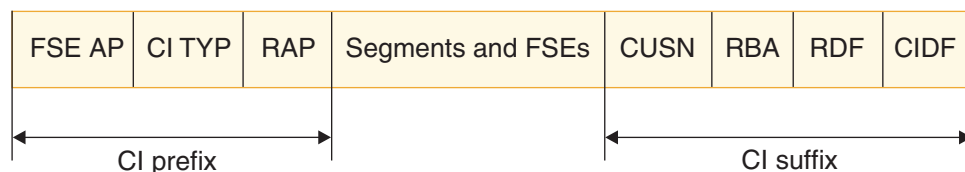
Figure 60. CI format

Table 51. CI format

CI section	Number of bytes	Explanation
FSE AP	2 bytes	Offset to the first free space element. These 2 bytes are unused if the CI is in the sequential dependent part.
CI TYP	2 bytes	Describes the use of this CI and the meaning of the next 4 bytes.
RAP	4 bytes	Root anchor point if this CI belongs to the base section of the root addressable area. All root segments randomizing to this CI are chained off this RAP in ascending key sequence. Only one RAP exists per CI. **Attention:** In the dependent and independent overflow parts, these 4 bytes are used by Fast Path control information. No RAP exists in sequential dependent CIs.
CUSN	2 bytes	CI Update Sequence Number (CUSN). A sequence number maintained in each CI. It is increased with each update of the particular CI during the synchronization process.
RBA	4 bytes	Relative byte address of this CI.
RDF	3 bytes	Record definition field (contains VSAM control information).
CIDF	4 bytes	CI definition field (contains VSAM control information).
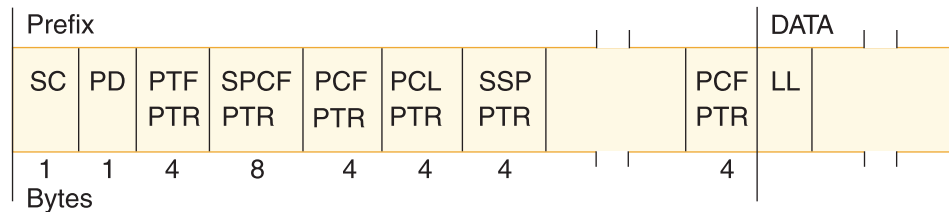
Figure 61. Root segment format (with sequential and direct dependent segments with subset pointers)

Table 52. Root segment format

Segment section	Number of bytes	Explanation
SC	1 byte	Segment code.
PD	1 byte	Prefix descriptor.
PTF	4 bytes	Physical twin forward pointer. Contains the RBA of the next root in key sequence.
SPCF	8 bytes	Sequential physical child first pointer. Contains the cycle count and RBA of the last inserted sequential dependent under this root. This pointer will not exist if the sequential dependent segment is not defined.
PCF	4 bytes	Physical child first pointer. PCF points to the first occurrence of a direct dependent segment type. There can be up to 126 PCF pointers or 125 PCF pointers if there is a sequential dependent segment. PCF pointers will not exist if direct dependent segments are not defined.
PCL	4 bytes	Physical child last pointer. PCL is an optional pointer that points to the last physical child of a segment type. This pointer will not exist if direct dependent segments are not defined.

Table 52. Root segment format (continued)

Segment section	Number of bytes	Explanation
SSP	4 bytes	Subset pointer. For each child type of the parent, up to eight optional subset pointers can exist.
LL	2 bytes	Variable length of this segment.
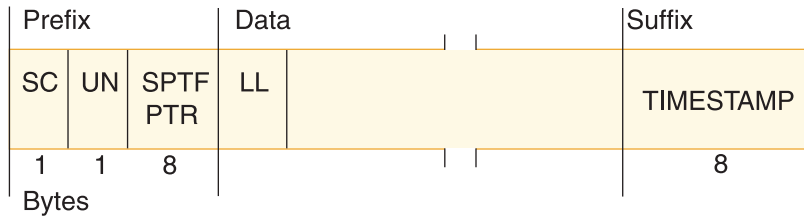
Figure 62. Sequential dependent segment format

Table 53. Sequential dependent segment format

Segment section	Number of bytes	Explanation
SC	1 byte	Segment code.
UN	1 byte	Prefix descriptor.
SPTF	8 bytes	Sequential physical twin forward pointer. Contains the cycle count and the RBA of the immediately preceding sequential dependent segment under the same root.
LL	2 bytes	Variable length of this segment.
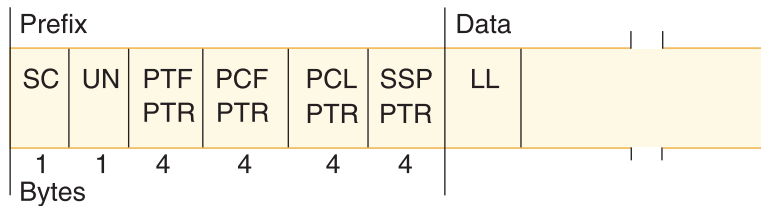
Figure 63. Direct dependent segment format

Table 54. Direct dependent segment format

Segment section	Number of bytes	Explanation
SC	1 byte	Segment code.
UN	1 byte	Unused.
PTF	4 bytes	Physical twin forward pointer. Contains the RBA of the next occurrence of this direct dependent segment type.

Table 54. Direct dependent segment format (continued)

Segment section	Number of bytes	Explanation
PCF	4 bytes	Physical child first pointer. PCF points to the first occurrence of a direct dependent segment type. In a direct dependent segment there can be up to 125 PCF pointers or 124 PCF pointers if there is a sequential dependent segment. PCF pointers will not exist if direct dependent segments are not defined.
PCL	4 bytes	Physical child last pointer. PCL is an optional pointer that points to the last physical child of a segment type. This pointer will not exist if direct dependent segments are not defined.
SSP	4 bytes	Subset pointer. For each child type of the parent, up to eight optional subset pointers can exist.
LL	2 bytes	Variable length of this segment.

Related concepts:

Root segment storage

DEDB root segments are stored as prescribed by the randomizing routine, and are chained in ascending key sequence from each anchor point.

Each CI in the base section of a UOW in an area has a single anchor point. Sequential processing using GN calls processes the roots in the following order:

1. Ascending area number
2. Ascending UOW
3. Ascending key in each anchor point chain

Each root segment contains, in ascending key sequence, a PTF pointer containing the RBA of the next root.

Related reference:

➡ Sample data entry database randomizing routines (DBFHDC40 / DBFHDC20 DBFHDC44 / DBFHDC24 DBFHDC2S) (Exit Routines)

Direct dependent segment storage

The DEDB maintains processing efficiency while supporting a hierarchical physical structure with direct dependent segment types.

A maximum of 127 segment types are supported (up to 126 direct dependent segment types, or 125 if a sequential dependent segment is present).

Direct dependent (DDEP) segment types can be efficiently retrieved hierarchically, and the user has complete online processing control over the segments. Supported processing options are insert, get, delete, and replace. With the replace function, users can alter the length of the segment. DEDB space management logic attempts to store an inserted direct dependent in the same CI that contains its root segment. If insufficient space is available in that CI, the root addressable overflow and then the independent overflow portion of the area are searched.

DDEP segments can be defined with or without a unique sequence field, and are stored in ascending key sequence.

Physical chaining of direct dependent segments consists of a physical child first (PCF) pointer in the parent for each defined dependent segment type and a physical twin forward (PTF) pointer in each dependent segment.

DEDBs allow a PCL pointer to be used. This pointer makes it possible to access the last physical child of a segment type directly from the physical parent. The INSERT rule LAST avoids the need to follow a potentially long physical child pointer chain.

Subset pointers are a means of dividing a chain of segment occurrences under the same parent into two or more groups of subsets. You can define as many as eight subset pointers for any segment type, dividing the chain into as many as nine subsets. Each subset pointer points to the start of a new subset.

Related concepts:

⬆ Processing Fast Path DEDBs with subset pointer command codes (Application Programming)

⬆ Processing Fast Path DEDBs with subset pointer options (Application Programming)

Sequential dependent segment storage

DEDB sequential dependent (SDEP) segments are stored in the sequential dependent part of an area in the order of entry.

SDEP segments chained from different roots in an area are intermixed in the sequential part of an area without regard to which roots are their parents. SDEP segments are specifically designed for fast insert capability. However, online retrieval is not as efficient because increased input operations can result.

If all SDEP dependents were chained from a single root segment, processing with Get Next in Parent calls would result in a backward sequential order. (Some applications are able to use this method.) Normally, SDEP segments are retrieved sequentially only by using the DEDB Sequential Dependent Scan utility (DBFUMSC0), described in *IMS Version 15 Database Utilities*. SDEP segments are then processed by offline jobs.

SDEP segments are used for data collection, journaling, and auditing applications.

Enqueue level of segment CIs

Allocation of CIs involves three different enqueue levels.

The enqueue levels are:
- A NO ENQ level, which is typical of any SDEP CI. SDEP segments can never be updated; therefore they can be accessed and shared by all regions at the same time.
- A SHARED level, which means that the CI can be shared between non-update transactions. A CI at the SHARED level delays requests from any update transactions.
- An EXCLUSIVE level, which prevents contenders from acquiring the same resource.

The level of enqueue at which ROOT and SDEP segment CIs are originally acquired depends on the intent of the transaction. If the intent is update, all

acquired CIs (with the exception of SDEP CIs) are held at the EXCLUSIVE level. If the intent is not update, they are held at the SHARED level, even though there is the potential for deadlock.

The level of enqueue, just described, is reexamined each time the buffer stealing facility is invoked. The buffer stealing facility examines each buffer (and CI) that is already allocated and updates its level of enqueue.

All other enqueued CIs are released and therefore can be allocated by other regions.

The following figure shows an example of a DEDB structure.
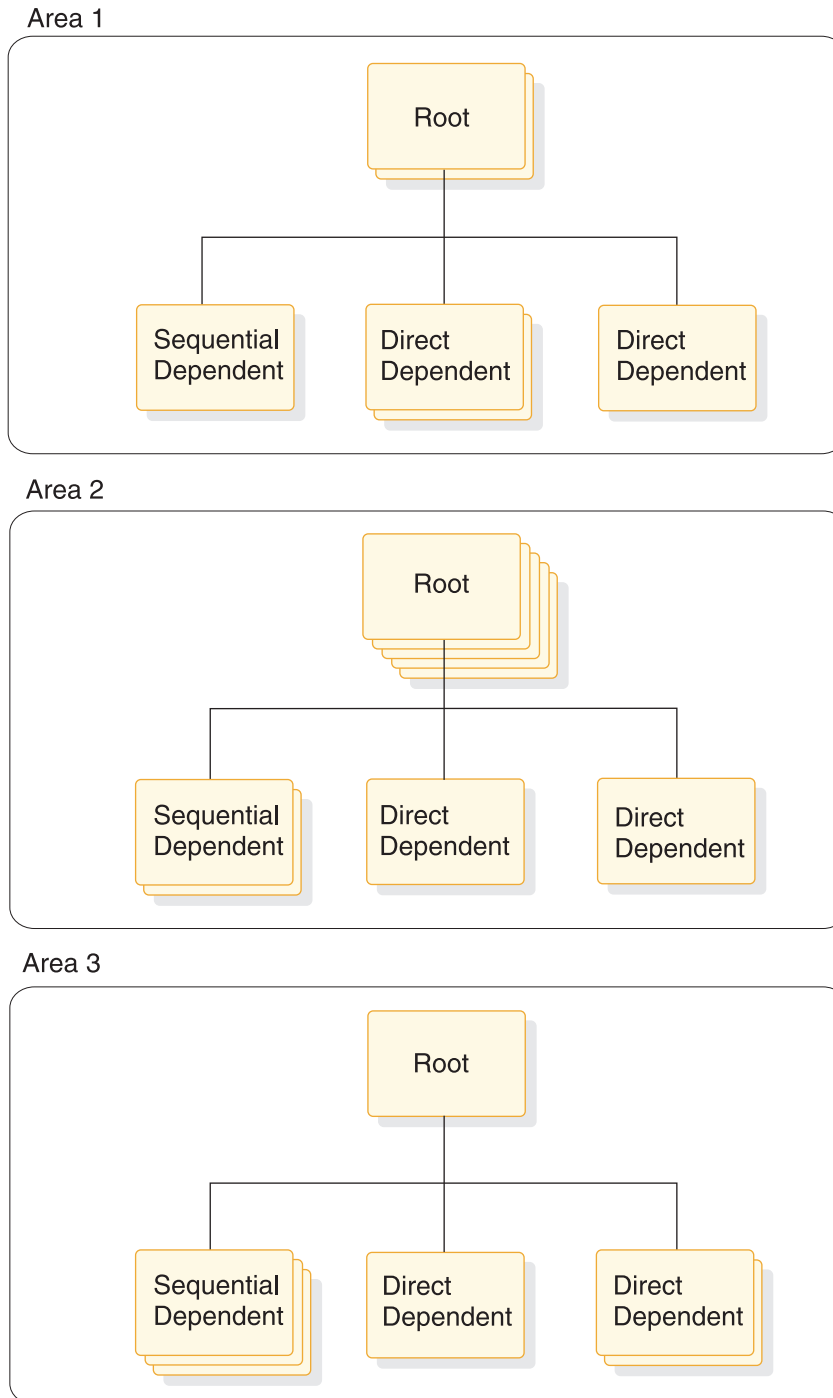
Figure 64. DEDB structure example

Related concepts:

"Fast Path buffer allocation algorithm" on page 480

"Areas and the DEDB format" on page 198

DEDB space search algorithm

The DEDB space search algorithm attempts to store the data in the minimum amount of CIs rather than scatter database record segments across a greater number of RAP and overflow CIs.

This topic contains Diagnosis, Modification, and Tuning information.

The trade-off is improved performance for future database record access versus optimum space utilization.

The general rule for inserting a segment into a DEDB is the same as it is for an HD database. The rule is to store the segment (root and direct dependents) into the most desirable block.

For root segments, the most desirable block is the RAP CI. For direct dependents, the most desirable block is the root CI. When space for storing either roots or direct dependents is not available in the most desirable block, the DEDB insert algorithm (described next) searches for additional space. Space to store a segment could exist:

- In the dependent overflow
- In an independent overflow CI currently owned by this UOW

Additional independent overflow CIs would be allocated if required.

Related reference:

➡ Sample data entry database randomizing routines (DBFHDC40 / DBFHDC20 DBFHDC44 / DBFHDC24 DBFHDC2S) (Exit Routines)

DEDB insert algorithm

The DEDB insert algorithm searches for additional space when space is not available in the most desirable block.

This topic contains Diagnosis, Modification, and Tuning information.

For root segments, if the RAP CI does not have sufficient space to hold the entire record, it contains the root and as many direct dependents as possible. Base CIs that are not randomizer targets go unused. The algorithm next searches for space in the first dependent overflow CI for this UOW. From the header of the first dependent overflow CI, a determination is made whether space exists in that CI.

If the CI pointed to by the current overflow pointer does not have enough space, the next dependent overflow CI (if one exists) is searched for space. The current overflow pointer is updated to point to this dependent overflow CI. If no more dependent overflow CIs are available, then the algorithm searches for space in the independent overflow part.

When an independent overflow CI has been selected for storing data, it can be considered a logical extension of the overflow part for the UOW that requested it.

The following figure shows how a UOW is extended into independent overflow. This UOW, defined as 10 CIs, includes 8 Base CIs and 2 dependent overflow CIs. Additional space is needed to store the database records that randomize to this UOW. Two independent overflow CIs have been acquired, extending the size of this UOW to 12 CIs. The first dependent overflow CI has a pointer to the second independent overflow CI indicating that CI is the next place to look for space.
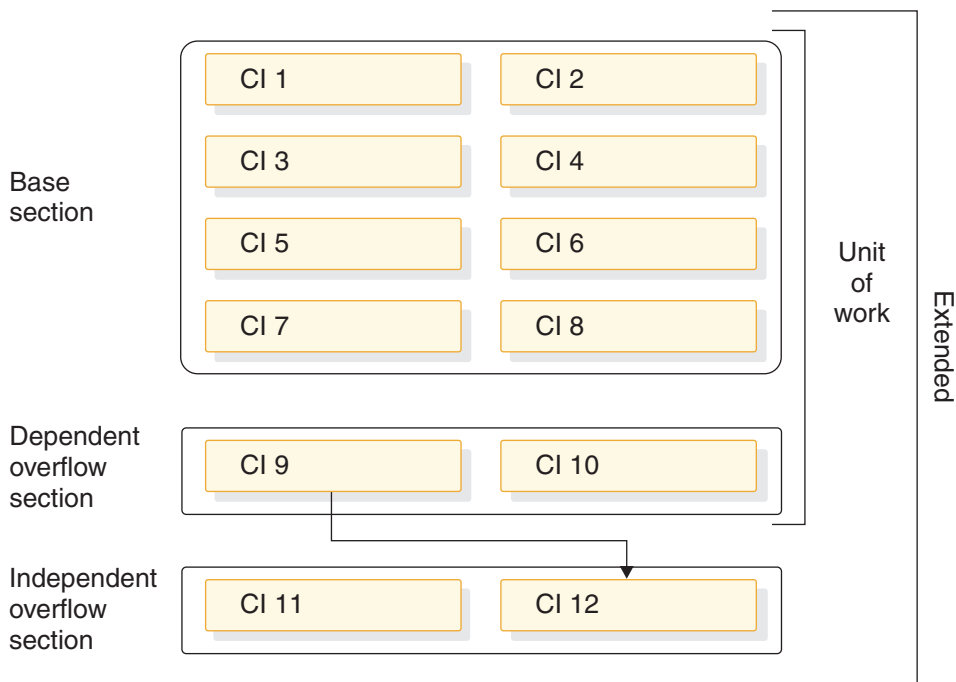
Figure 65. Extending a UOW to use independent overflow

Related reference:

"CI and segment formats" on page 208

DEDB free space algorithm

The DEDB free space algorithm is used to free dependent overflow and independent overflow CIs.

This topic contains Diagnosis, Modification, and Tuning information.

When a dependent overflow CI becomes entirely empty, it becomes the CI pointed to by the current overflow pointer in the first dependent overflow CI, indicating that this is the first overflow CI to use for overflow space if the most desirable block is full. An independent overflow CI is owned by the UOW to which it was allocated until every segment stored in it has been removed. When the last segment in an independent overflow CI is deleted, the empty CI is made available for reuse. When the last segment in a dependent overflow CI is deleted, it can be reused as described at the beginning of this topic.

A dependent overflow or an independent overflow CI can be freed by reorganization or by segment deletion.

Reorganization

During online reorganization, the segments within a UOW are read in GN order and written to the reorganization utility private buffer set. This process inserts segments into the reorganization utility private buffer set, eliminating embedded free space.

If all the segments do not fit into the reorganization utility private buffer set (RAP CI plus dependent overflow CIs), then new independent overflow CIs are allocated

as needed. When the data in the reorganization utility private buffer set is copied back to the correct location, then the following events occur:

- The newly acquired independent overflow CIs are retained.
- The old segments are deleted.
- Previously allocated independent overflow CIs are freed.

Segment deletion

A segment is deleted either by an application DLET call or because a segment is REPLaced with a different length.

Segment REPLace can cause a segment to move. Full Function handles segment length increases differently from DEDBs. In Full Function, an increased segment length that does not fit into the available free space is split, and the data is inserted away from the prefix. For DEDBs, if the replaced segment is changed, it is first deleted and then reinserted. The insertion process follows the normal space allocation rules.

The REPL call can cause a dependent overflow or an independent overflow CI to be freed if the last segment is deleted from the CI.

Managing unusable space with IMS tools

Space in a DEDB should be closely monitored to avoid out-of-space conditions for an area.

Products such as the IMS High Performance (HP) Pointer Checker, which includes the Hierarchical Database (HD) Tuning Aid and Space Monitor tools, can identify the different percentages of free space in the RAP, dependent overflow, and independent overflow CIs. If a large amount of space exists in the RAP CIs or dependent overflow CIs, and independent overflow has a high use percentage, a reorganization can allow the data to be stored in the root addressable part, freeing up independent overflow CIs for use by other UOWs. The IMS HP Pointer Checker and the tools it includes can help you determine if the data distribution is reasonable.

Related concepts:

"Tuning Fast Path systems" on page 712

DL/I calls against a DEDB

DEDB processing uses the same call interface as DL/I processing. Therefore, any DL/I call or calling sequence executed against a DEDB has the same logical result as if executed against an HDAM or PHDAM database.

This topic contains Diagnosis, Modification, and Tuning information.

The SSA rules for DEDBs have the following restrictions:

- You cannot use the Q command code with DEDBs.
- IMS ignores command codes used with sequential dependent segments.
- If you use the D command code in a call to a DEDB, the P processing option need not be specified in the PCB for the program. The P processing option has a different meaning for DEDBs than for DL/I databases.

Related concepts:

Processing DEDBs (IMS and CICS with DBCTL) (Application Programming)

Mixed mode processing

IMS application programs can run as message processing programs (MPPs), batch message processing programs (BMPs), and Fast Path programs (IFPs).

IFPs can access full function databases. Similarly, MPPs and BMPs can access DEDBs and MSDBs.

Because of differences in sync point processing, there are differences in the way database updates are committed. IFPs that request full function resources, or MPPs (or BMPs) that request DEDB (or MSDB) resources operate in "mixed mode".

Related concepts:

"Fast Path synchronization points" on page 241

Main storage databases (MSDBs)

The MSDB structure consists of fixed-length root segments only, although the root segment length can vary between MSDBs.

The maximum length of any segment is 32,000 bytes with a maximum key length of 240 bytes. Additional prefix data extends the maximum total record size to 32,258 bytes.

The following options are *not* available for MSDBs:
* Multiple data set groups
* Logical relationships
* Secondary indexing
* Variable-length segments
* Field-level sensitivity

The MSDB family of databases consists of four types:
* Terminal-related *fixed* database
* Terminal-related *dynamic* database
* Non-terminal-related database *with* terminal keys
* Non-terminal-related database *without* terminal keys

Recommendation: Use DEDBs instead of MSDBs when you develop new Fast Path databases. Terminal-related MSDBs and non-terminal-related MSDBs with terminal-related keys are no longer supported. Although non-terminal-related MSDBs with non-terminal-related-keys are still supported, you should consider converting any existing MSDBs to DEDBs. You can use the MSDB-to-DEDB Conversion utility.

An MSDB is defined in the DBD in the same way as any other IMS database, by coding `ACCESS=MSDB` in the DBD statement. The REL keyword in the DATASET statement selects one of the four MSDB types.

Both dynamic and fixed terminal-related MSDBs have the following characteristics:
* The record can be updated only through processing of messages issued from the LTERM that owns the record. However, the record can be read using messages from any LTERM.
* The name of the LTERM that owns a segment is the key of the segment. An LTERM cannot own more than one segment in any one MSDB.

- The key does not reside in the stored segment.
- Each segment in a fixed terminal-related MSDB is assigned to and owned by a different LTERM.

Terminal-related MSDBs cannot be accessed by ETO terminals.

Non-terminal-related MSDBs have the following characteristics:
- No ownership of segments exists.
- No insert or delete calls are allowed.
- The key of segments can be an LTERM name or a field in the segment. As with a terminal-related MSDB, if the key is an LTERM name, it does not reside in the segment. If the key is not an LTERM name, it resides in the sequence field of the segment. If the key resides in the segment, the segments must be loaded in key sequence because, when a qualified SSA is issued on the key field, a binary search is initiated.

Related concepts:

Chapter 15, "Full-function database types," on page 117

"Performance considerations overview" on page 119

"The segment" on page 15

When to use an MSDB

MSDBs store and provide access to an installation's most frequently used data. The data in an MSDB is stored in segments, and each segment available to one or all terminals.

MSDBs provide a high degree of parallelism and are suitable for applications in the banking industry (such as general ledger). To provide fast access and allow frequent update to this data, MSDBs reside in virtual storage during execution.

One use for a terminal-related fixed MSDB is in an application in which each segment contains data associated with a logical terminal. In this type of application, the application program can read the data (possibly for general reporting purposes) but cannot update it.

Non-terminal-related MSDBs (without terminal-related keys) are typically used in applications in which a large number of people need to update data at a high transaction rate. An example of this is a real-time inventory control application, in which reduction of inventory is noted from many cash registers.

MSDBs storage

The MSDB Maintenance utility (DBFDBMA0) creates the MSDBINIT sequential data set in physical ascending sequence.

During a cold start, or by operator request during a normal warm start, the sequential data set MSDBINIT is read and the MSDBs are created in virtual storage. See the following figure.
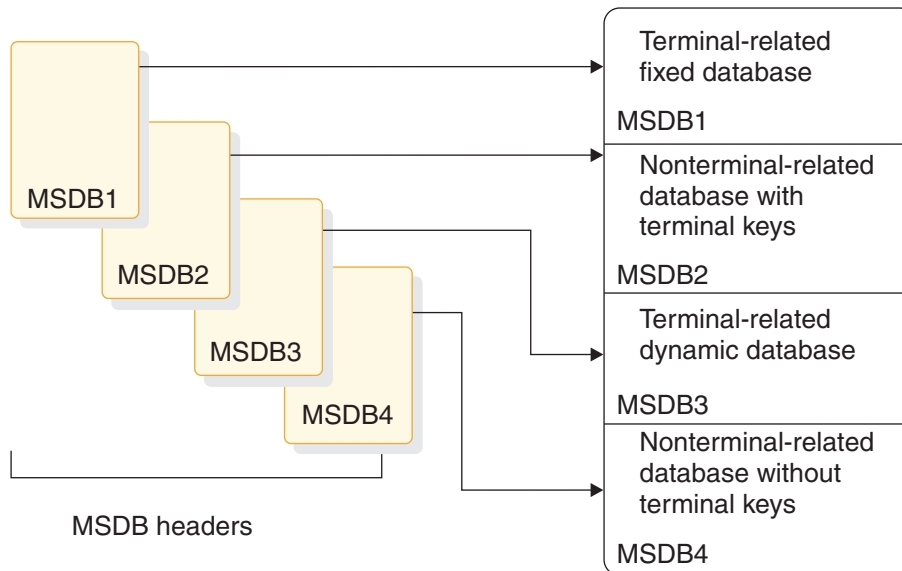
Figure 66. MSDB pointers

During a warm start, the control program uses the current checkpoint data set for initialization. The MSDB Maintenance utility can also modify the contents of an old MSDBINIT data set. For warm start, the master terminal operator can request use of the IMS.MSDBINIT, rather than a checkpoint data set.

The following figure shows the MSDBINIT record format. The table following the figure explains the record parts.

	LL	00	DBDname	Count	Type	KL	KEY	MSDB Segment
Bytes	2	2	8	4	1	1	Varies	Varies(MAX 32,000)

Figure 67. MSDBINIT record format

Table 55. MSDBINIT record format

Record part	Bytes	Explanation
LL	2	Record length (32,258 maximum)
X'00'	2	Always hexadecimal zeros
DBDname	8	DBD name
Count	4	Segment count
Type	1	MSDB type: • X'11' non-related • X'31' non-related with terminal keys • X'33' fixed related • X'37' dynamic related
KL	1	Key length (240 maximum)
Key	varies	Key or terminal name
MSDB segment	varies	MSDB segment (32,000 maximum)

Related tasks:

"Loading an MSDB" on page 584

MSDB record storage

MSDB records contain no pointers except the forward chain pointer (FCP) connecting free segment records in the terminal-related dynamic database.

This topic contains Diagnosis, Modification, and Tuning information.

The following figure shows a high-level view of how MSDBs are arranged in priority sequence.
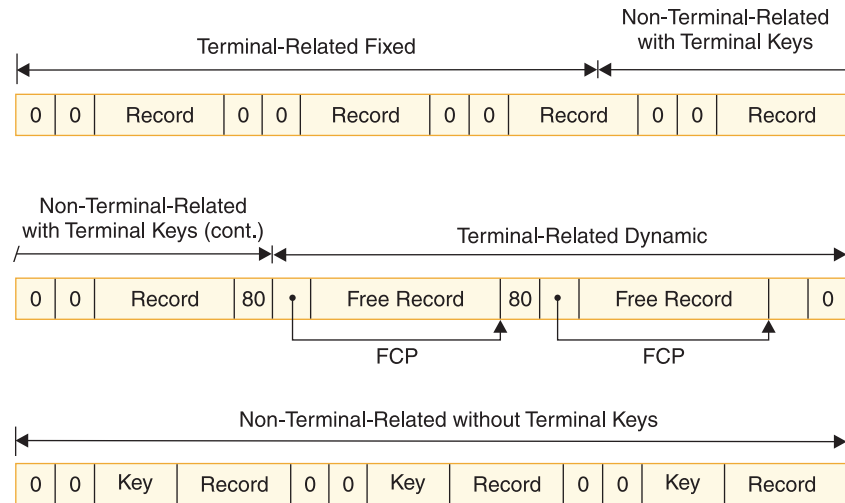
Figure 68. Sequence of the four MSDB organizations

Saving MSDBs for restart

At system checkpoint, a copy of all MSDBs is written alternately to one of the MSDB checkpoint data sets—MSDBCP1 or MSDBCP2.

During restart, the MSDBs are reloaded from the most recent copy on MSDBCP1 or MSDBCP2. During an emergency restart, the log is used to update the MSDB. During a normal restart, the operator can reload from MSDBINIT using the MSDBLOAD parameter on the restart command.

On a cold start (including /ERE CHKPT 0), MSDBs are loaded from the MSDBINIT data set.

DL/I calls against an MSDB

All DL/I database calls, except those that specify "within parent", are valid with MSDBs.

Because an MSDB is a root-only database, a "within parent" call is meaningless. Additionally, the DL/I call, FLD, exists that is applicable to all MSDBs. The FLD call allows an application program to check and modify a single field in an MSDB segment.

Rules for using an SSA

MSDB processing imposes the following restrictions on the use of an SSA (segment search argument).

No boolean operator

No command code

Even with the preceding restrictions, the result of a call to the database with no SSA, an unqualified SSA, or a qualified SSA remains the same as a call to the full-function database. For example, a retrieval call without an SSA returns the first record of the MSDB or the full-function database, depending on the environment in which you are working. The following list shows the type of compare or search technique used for a qualified SSA.

Type of Compare

- Sequence field: logical
- Non-sequence arithmetic field: arithmetic
- Non-sequence non-arithmetic: logical

Type of Search

- Sequence field: binary if operator is = or >=, otherwise sequential
- Non-sequence arithmetic field: sequential
- Non-sequence non-arithmetic: sequential

Insertion and deletion of segments

The terminal-related dynamic MSDB database accepts ISRT and DLET calls, and the other MSDB databases do not.

Actual physical insertion and deletion of segments do not occur in the dynamic database. Rather, a segment is assigned to an LTERM from a pool of free segments by an ISRT call. The DLET call releases the segment back to the free segment pool.

The figure in "Combination of binary and direct access methods" shows a layout of the four MSDBs and the control blocks and tables necessary to access them. The Extended Communications Node Table (ECNT) is located by a pointer from the Extended System Contents Directory (ESCD), which in turn is located by a pointer from the System Contents Directory (SCD). The ESCD contains first and last header pointers to the MSDB header queue. Each of the MSDB headers contains a pointer to the start of its respective database area.

Combination of binary and direct access methods

A combination access technique works against the MSDB on a DL/I call. The access technique combines a binary search and the direct access method.

A binary search of the ECNT table attempts to match the table LTERM names to the LTERM name of the requesting terminal. When a match occurs, the application program accesses the segment of the desired database using a direct pointer in the ECNT table. Access to the non-terminal-related database segments without terminal keys is accomplished by a binary search technique only, without using the ECNT.

The following figure shows the ECNT and MSDB storage layout.
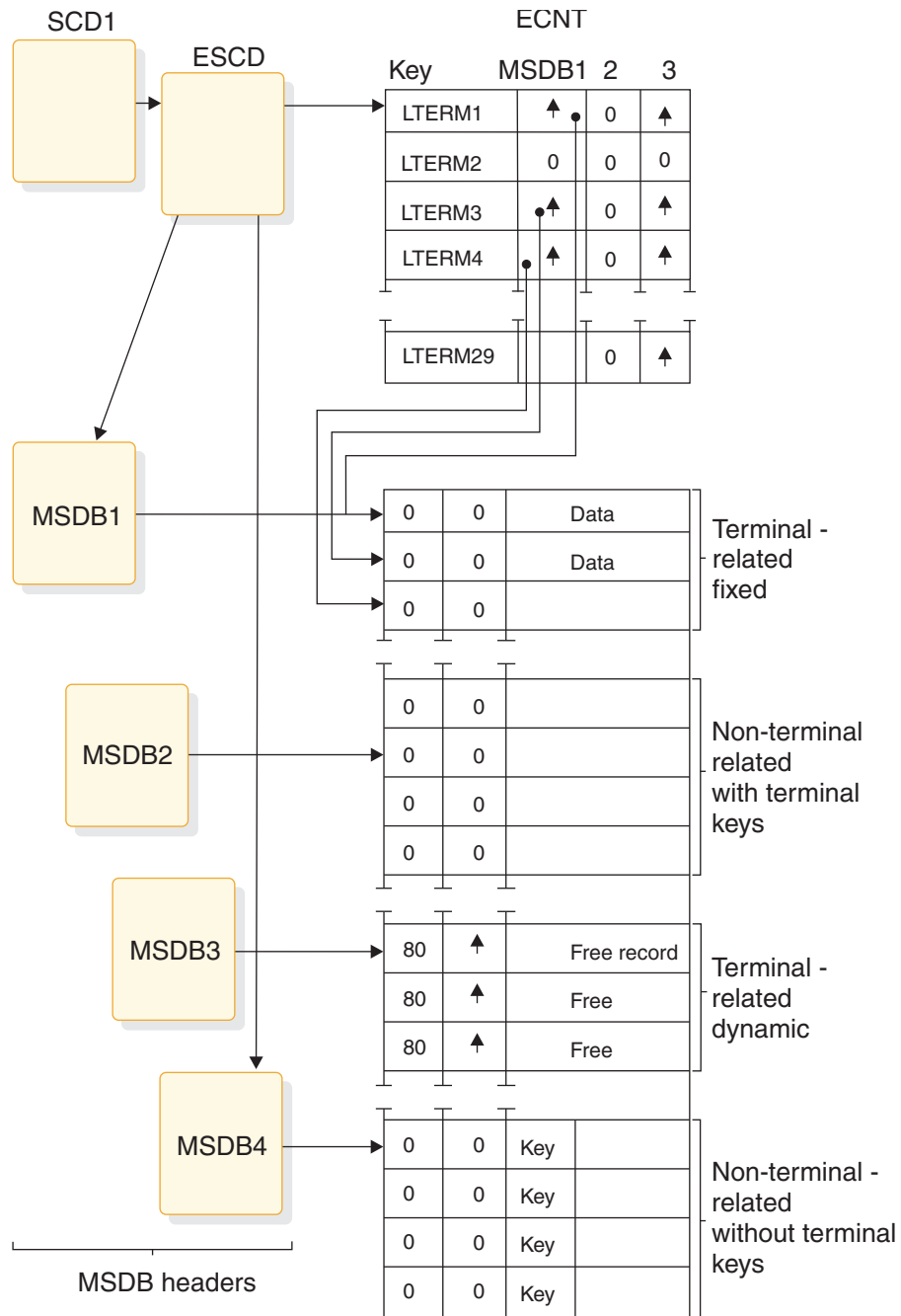
Figure 69. ECNT and MSDB storage layout

Position in an MSDB

Issuing a DL/I call causes a position pointer to fix on the current segment. The meaning of "next segment" depends on the key of the MSDB.

The current segment in a non-terminal-related database without LTERM keys is the physical segment against which a call was issued. The next segment is the following physically adjacent segment after the current segment. The other three databases, using LTERM names as keys, have a current pointer fixed on a position in the ECNT table. Each entry in the table represents one LTERM name and

segment pointers to every MSDB with which LTERM works. A zero entry indicates no association between an LTERM and an MSDB segment. If nonzero, the next segment is the next entry in the table. The zero entries are skipped until a nonzero entry is found.

The field call

The DL/I FLD call, available to MSDBs and DEDB, allows for the operation on a field, rather than on an entire segment.

Additionally, the DL/I FLD call allows conditional operation on a field.

Modification is done with the CHANGE form of the FLD call. The value of a field can be tested with the VERIFY form of the FLD call. These forms of the call allow an application program to test a field value before applying the change. If a VERIFY fails, all CHANGE requests in the same FLD call are denied.

If failures occur during processing of the FLD call, IMS reprocesses the call only if the failures are caused by deadlocks or verification errors. For other types of failures, abend U0819 is issued.

Related reference:

➡ FLD call (Application Programming APIs)

Related information:

➡ 0819 (Messages and Codes)

Call sequence results

The same call sequence against MSDBs and other IMS databases might bring different results.

For parallel access to MSDB data, updates to MSDB records take place during sync point processing. Changes are not reflected in those records until the sync point is completed. For example, the sequence of calls GHU (Get-Hold-Unique), REPL (Replace), and GU (Get-Unique) for the same database record results in the same information in the I/O area for the GU call as that returned for the GHU.

The postponement of an updated database record to the point of commitment is also true of FLD/CHANGE calls, and affects FLD/VERIFY calls. You should watch for multiple FLD/VERIFY and FLD/CHANGE calls on the same field of the same segment. Such sequences can decrease performance because reprocessing results.

For terminal-related dynamic MSDBs, the following examples of call sequences do *not* have the same results as with other IMS databases or DEDBs:

- A GHU following an ISRT receives a 'segment not found' status code.
- An ISRT after a DLET receives a 'segment already exists' status code.
- No more than one ISRT or DLET is allowed for each MSDB in processing a transaction.

The preceding differences become more critical when transactions update or refer to both full function DL/I and MSDB data. Updates to full function DL/I databases and DEDBs are immediately available while MSDB changes are not. For example, if you issue a GHU and a REPL for a segment in an MSDB, then you issue another get call for the same segment in the same commit interval, the segment IMS returns to you is the "old" value, not the updated one.

If processing is not single mode, this difference can increase. In the case of multiple mode processing, the sync point processing is not invoked for every transaction. Your solution might be to ask for single mode processing when MSDB data is to be updated.

Another consideration for MSDB processing is that terminal-related MSDB segments can be updated only by transactions originating from the owners of the segment, the LTERMs. Programs that are non-transaction-driven BMPs can only update MSDBs that are declared as non-terminal-related.

Fast Path Virtual Storage Option

The Fast Path Virtual Storage Option (VSO) allows you to map data into virtual storage or a coupling facility structure.

You can map one or more DEDB areas into virtual storage or a coupling facility structure by defining the DEDB areas as VSO areas.

For high-end performance applications with DEDBs, defining your DEDB areas as VSO allows you to realize the following performance improvements:

- Reduced read I/O

  After an IMS and VSAM control interval (CI) has been brought into virtual storage, all subsequent I/O read requests read the data from virtual storage rather than from DASD.

- Decreased locking contention

  For VSO DEDBs, locks are released after both of the following:

  - Logging is complete for the second phase of an application synchronization (commit) point
  - The data has been moved into virtual storage

  For non-VSO DEDBs, locks are held at the VSAM CI-level and are released only after the updated data has been written to DASD.

- Fewer writes to the area data set

  Updated data buffers are not immediately written to DASD; instead they are kept in the data space and written to DASD at system checkpoint or when a threshold is reached.

In all other respects, VSO DEDBs are the same as non-VSO DEDBs. Therefore, VSO DEDB areas are available for IMS DBCTL and LU 6.2 applications, as well as other IMS DB or IMS TM applications. Use the DBRC commands `INIT.DBDS` and `CHANGE.DBDS` to define VSO DEDB areas.

The virtual storage for VSO DEDB areas is housed differently depending on the share level assigned to the area. VSO DEDB areas with share levels of 0 and 1 are loaded into a z/OS data space. VSO DEDB areas with share levels of 2 and 3 are loaded into a coupling facility cache structure.

Coupling facility cache structures are defined by the system administrator and can accommodate either a single DEDB area or multiple DEDB areas. Cache structures that support multiple DEDB areas are called *multi-area structures*. For more information on multi-area structures, see *IMS Version 15 System Administration*.

Recommendation: Terminal-related MSDBs and non-terminal-related MSDBs with terminal-related keys are not supported. Non-terminal-related MSDBs without

terminal-related keys are still supported. Therefore, you should consider converting all your existing MSDBs to VSO DEDBs or non-VSO DEDBs.

Related concepts:

"DEDBs and data sharing" on page 205

Restrictions for using VSO DEDB areas

VSO DEDB areas have a number of restrictions to their use.

The restrictions include:

- VSO DEDB areas must be registered with DBRC.
- For local VSO DEDB areas, z/OS data spaces impose a 2 GB (2 147 483 648 bytes) maximum size limit, even though the maximum size of a single or multiple area data set on DASD is 4 GB. If the local VSO DEDB area is more than 2 GB, the area fails to open.

  When Local VSO DEDB areas are opened for preloading, IMS checks to make sure that the area can fit into the data space. If the area cannot fit, the open fails.

  The actual size available in a z/OS data space for a local VSO DEDB area is the maximum size (2 GB) minus amounts used by z/OS (from 0 to 4 KB) and IMS Fast Path (approximately 100 KB).

  To see the size, usage, and other statistics for a VSO DEDB area, enter the `/DISPLAY FPV` command.

- For shared VSO DEDB areas, z/OS coupling facility cache structures do not place a limit on the maximum size of an area data set. The maximum size of a coupling facility cache structure is 2 GB. If a shared VSO DEDB area is larger than 2 GB, IMS loads a maximum of 2 GB of CIs into the coupling facility cache structure.

  IMS does not check to see if a shared VSO DEDB area can fit into a coupling facility cache structure, regardless of whether the shared VSO DEDB area is preloaded or loaded only when requested.

- The DEDB Area Data Set Compare utility (DBFUMMH0) does not support VSO DEDB areas.

Related concepts:

"Accessing a data space" on page 234

Related reference:

➡️ /DISPLAY FPV command (Commands)

Defining a VSO DEDB area

All of the Virtual Storage Option (VSO) information for a DEDB is recorded in the RECON data set.

Use the following parameters of the DBRC `INIT.DBDS` and `CHANGE.DBDS` commands to define your VSO DEDB Areas:

VSO   Defines the area as a VSO area.

When a CI is read for the first time, it will be copied into a z/OS data space or a coupling facility structure. Data is read into a common buffer and is then copied into the data space or structure. Subsequent access to the data retrieves it from the data space or structure rather than from DASD.

CIs that are not read are not copied into the data space or structure.

All updates to the data are copied back to the data space or structure and any locks held are released. Updated CIs are periodically written back to DASD.

NOVSO

Defines the area as a non-VSO area. This is the default.

You can use NOVSO to define a DEDB as non-VSO or to turn off the VSO option for a given area. If the area is in virtual storage when it is redefined as NOVSO, the area must be stopped (/STOP AREA or /DBR AREA) or removed from virtual storage (/VUNLOAD) for the change to take effect.

PRELOAD

For VSO areas, this preloads the area into the data space or coupling facility structure when the VSO area is opened. This keyword implies the PREOPEN keyword, thus if PRELOAD is specified, then PREOPEN does not have to be specified.

The root addressable portion and the independent overflow portion of an area are loaded into the data space or coupling facility structure at control region initialization or during /START AREA processing. Data is then read from the data space or coupling facility structure to a common buffer. Updates are copied back to the data space or coupling facility structure and any locks are released. Updated CIs are periodically written back to DASD.

NOPREL

Defines the area as load-on-demand. For VSO DEDBs areas, as CIs are read from the data set, they are copied to the data space or coupling facility structure. This is the default.

To define an area with NOPREL gives you the ability to deactivate the preload processing. The area is not preloaded into the data space or coupling facility structure the next time that it is opened.

If you specify NOPREL, and you want the area to be preopened, you must separately specify PREOPEN for the area.

CFSTR1

Defines the name of the cache structure in the primary coupling facility. Cache structure names must follow z/OS coupling facility naming conventions. CFSTR1 uses the name of the DEDB area as its default. This parameter is valid only for VSO DEDB areas that are defined with SHARELVL(2|3).

CFSTR2

Defines the secondary coupling facility cache structure name when you use IMS-managed duplexing of structures. The cache structure name must follow z/OS coupling facility naming conventions. CFSTR2 does not provide a default name. This parameter is valid only for VSO areas of DEDBs that are defined with SHARELVL(2|3) and that are single-area structures. This parameter cannot be used with multi-area structures, which use system-managed duplexing.

MAS Defines a VSO DEDB area as using a multi-area structure as opposed to a single-area structure.

NOMAS

Defines a VSO DEDB area as using a single-area cache structure as opposed to a multi-area structure. NOMAS is the default.

LKASID
NOLKASID

Specifies whether buffer lookaside is to be performed on read requests for this area.

For VSO DEDB areas that use a single-area structure, you can specify LKASID or NOLKASID in either the RECON data set or, optionally, in the DFSVSMxx PROCLIB member. Specifications for LKASID or NOLKASID made in the RECON data set override any specifications for LKASID or NOLKASID in the DFSVSMxx PROCLIB member.

For VSO DEDB areas that use a multi-area structure, LKASID or NOLKASID must be specified by using the DFSVSMxx PROCLIB member. Specifications in the RECON data set are ignored.

FULLSEG
NOFULLSG

Mutually exclusive, optional keywords that specify whether the full segment image is logged in the X'5950' log record when the segment is updated by a Replace (REPL) call. These keywords are valid only for Fast Path DEDBs.

FULLSEG indicates that the full segment image is to be logged.

NOFULLSG indicates that only the updated portion of a segment is to be logged.

If neither of these keywords is specified, the default value set in the database record for the DEDB is used.

Related concepts:

"Coupling facility structure naming convention" on page 232

"Defining a private buffer pool using the DFSVSMxx IMS.PROCLIB member" on page 233

"Overview of dynamic database buffer pools" on page 701

Sysplex data-sharing concepts and terminology (System Administration)

Related tasks:

"Registering a cache structure name with DBRC" on page 233

Related reference:

INIT.DBDS command (Commands)

CHANGE.DBDS command (Commands)

DBRC commands (Commands)

DFSVSMxx member of the IMS PROCLIB data set (System Definition)

VSO DEDB areas and the PREOPEN and NOPREO keywords

The PREOPEN and NOPREO keywords of DBRC's INIT.DBDS and CHANGE.DBDS commands apply to both VSO DEDB areas and non-VSO DEDB areas.

When a NOPREO area is also defined as shared VSO with a share level of 2 or 3, you can open the area with the /START AREA command. This connects the area to the VSO structures.

You can use the DBRC commands to define your VSO DEDB areas at any time; it is not necessary that IMS be active. The keywords specified on these DBRC commands go into effect at two different points in Fast Path processing:

- Control region startup

  After the initial checkpoint following control region initialization, DBRC provides a list of areas with any of the VSO options (VSO, NOVSO, PRELOAD, and NOPREL) or either of the PREOPEN or NOPREO options. The options are then maintained by IMS Fast Path.

- Command processing

  When you use a /START AREA command, DBRC provides the VSO options or PREOPEN|NOPREO options for the area. If the area needs to be preopened or preloaded, it is done at this time.

  When you use a /STOP AREA command, any necessary VSO processing is performed.

  Related Reading: See *IMS Version 15 Commands, Volume 2: IMS Commands N-V* for details on /START and /STOP command processing.

Sharing of VSO DEDB areas

Sharing of VSO DEDB areas allows multiple IMS systems to concurrently read and update the same VSO DEDB area. The three main participants are the coupling facility hardware, the coupling facility policy software, and the XES and z/OS services.

The coupling facility hardware provides high-performance, random-access shared storage in which IMS systems can share data in a sysplex environment. The shared storage area in the coupling facility is divided into sections, called *structures*. For VSO DEDB data, the structure type used is called a *cache structure*, as opposed to a list structure or a lock structure. The cache structure is designed for high-performance read reference reuse and deferred write of modified data. The coupling facility and structures are defined in a common z/OS data set, the *couple data set* (COUPLExx).

The coupling facility policy software and its cache structure services provide interfaces and services to z/OS that allow sharing of VSO DEDB data in shared storage. Shared storage controls VSO DEDB reads and writes:

- A read of a VSO CI brings the CI into the coupling facility from DASD.
- A write of an updated VSO CI copies the CI to the coupling facility from main storage, and marks it as changed.
- Changed CI data is periodically written back to DASD.

The XES and z/OS services provide a way of manipulating the data within the cache structures. They provide high performance, data integrity, and data consistency for multiple IMS systems sharing data.

Related concepts:

"DEDBs and data sharing" on page 205

The coupling facility and shared storage

In the coupling facility shared storage, a cache structure can represent one or multiple VSO DEDB areas; however, any given VSO DEDB area can be represented by only one cache structure.

Cache structures are not persistent. That is, they are deleted after the last IMS system disconnects from the coupling facility.

Duplexing structures

Duplexed structures are duplicate structures for the same area.

Duplexing allows you to have dual structure support for your VSO DEDB areas, which helps to ensure the availability and recoverability of your data.

Structure duplexing can be either IMS-managed or system-managed. With IMS-managed duplexing, you must define both the primary and the secondary structures in DBRC and in the z/OS coupling facility resource management (CFRM) policy. When you use system-managed duplexing, you have to define only the primary structure. The duplexing operation is transparent to you, except that you need to request duplex mode in your CFRM policy and allocate additional resources for a secondary structure instance.

VSO multi-area structures require the use of system-managed duplexing.

Related tasks:

➡ z/OS structure duplexing for CQS (System Administration)

Automatic altering of structure size

z/OS can automatically expand or contract the size of a VSO structure in the coupling facility if it needs storage space.

Enabling this function for preloaded VSO DEDBs can prevent wasted space; however, you must be careful with this function when VSO DEDBs are loaded on demand.

Recommendation: If you preload your shared VSO DEDB areas, do not use the automatic alter function. The automatic alter function might reclaim any unchanged data in the cache structure.

To ensure correct sizing and that the automatic alter function is disabled for your preloaded shared VSO DEDB areas, specify the following parameters in the area's CFRM policy:

`INITSIZE(x)`
> Specifies the initial amount of space to be allocated for the structure in the coupling facility. You can estimate this size using the IBM System z® Coupling Facility Structure Sizer Tool (CFSizer). CFSizer is available for you to use at the following website: www.ibm.com/servers/eserver/zseries/cfsizer/, or search for "CFSizer" at the IBM website: www.ibm.com.

`SIZE(y)`
> Specifies the maximum amount of space to be allocated for the structure in the coupling facility.

`ALLOWAUTOALT(NO)`
> Specifies whether to allow system-initiated alters (automatic alter) for this structure. For preloaded shared VSO DEDB areas, you should specify ALLOWAUTOALT(NO).

If the size specified by INITSIZE is too small, IMS will alter the size of the structure to the needed value as calculated by IXLCSP, up to the size specified in the SIZE parameter.

Related Reading: For information about the CFRM parameters that enable automatic altering of structures, see *z/OS MVS™ Setting Up a Sysplex*

Related concepts:

➡️ Data sharing in IMS environments (System Administration)

System-managed rebuild

You can reconfigure a coupling facility while keeping all VSO structures online by copying the structures to another coupling facility. There is no change to the VSO definition.

Related concepts:

➡️ Sysplex data-sharing concepts and terminology (System Administration)

Private buffer pools

IMS provides special private buffer pools for Shared VSO areas. Each pool can be associated with an area, a DBD, or a specific group of areas.

These private buffer pools are only used for Shared VSO data. Using these private buffer pools, the customer can request buffer lookaside for the data. The keywords LKASID or NOLKASID, when specified on the DBRC commands INIT.DBDS or CHANGE.DBDS, indicate whether to use this lookaside capability or not.

Authorizing connections to DEDB VSO structures

Manage access to shared DEDB VSO structures by defining security profiles that grant access to the cache structure to only authorized IMS systems.

If you use RACF, the RACF security administrator defines the security profiles in the FACILITY class.

Prior to connecting to a DEDB VSO structure, an IMS system issues a RACROUTE REQUEST=AUTH call that uses the job name of the IMS control region as the user ID. To access the structure, the user ID must have at least UPDATE authority in the security profile.

The name of a security profile must use the format VSOSTR.*structure_name*, where *structure_name* is the name of the VSO structure that is to be protected. This structure name must match the structure name defined in the RECON data sets for the structure.

The following example shows the RACF commands to that both define a RACF security profile for a VSO structure and grant update authority to an IMS system. The name of the protected VSO structure is DB21AR1@STRUCT@1. The IMS system receiving update authority is VSOB06A1.

```
ADDUSER VSOB06A1
     RDEFINE FACILITY (VSOSTR.DB21AR1@STRUCT@1) UACC(NONE)
     PERMIT VSOSTR.DB21AR1@STRUCT@1 CLASS(FACILITY) ID(VSOB06A1)
     ACCESS(UPDATE)
     SETROPTS CLASSACT(FACILITY)
```

Related tasks:

"Registering a cache structure name with DBRC" on page 233

Defining a VSO DEDB cache structure name

The system programmer defines all coupling facility structures, including VSO cache structures, in the CFRM policy definition.

In this policy definition, VSO structures are defined as cache structures, as opposed to list structures (used by shared queues) or lock structures (used by IRLM).

Coupling facility structure naming convention

A coupling facility structure name is 16 characters long, padded on the right with blanks if necessary.

The name can contain any of the following, but must begin with an uppercase, alphabetic character:

Uppercase alphabetic characters

Numeric characters

Special characters ($, @, and #)

Underscore (_)

IBM names begin with:

'SYS'

Letters 'A' through 'I' (uppercase)

An IBM component prefix

Related concepts:

"Defining a VSO DEDB area" on page 226

Examples of defining coupling facility structures

The following JCL shows how to define two structures in separate coupling facilities.

```
//UPDATE  EXEC PGM=IXCL2FDA
//SYSPRINT DD SYSOUT=A
//*
//*  THE FOLLOWING SYSIN WILL UPDATE THE POLICY DATA IN THE COUPLE
//*  DATASET FOR CFRM (COUPLING FACILITY RESOURCE MANAGEMENT)
//*
//SYSIN  DD *
 UPDATE DSN(IMS.DSHR.PRIME.FUNC) VOLSER(DSHR03)

  DEFINE POLICY(POLICY1)

    DEFINE CF(FACIL01)
          ND(123456)
          SIDE(0)
          ID(01)
          DUMPSPACE(2000)

    DEFINE CF(FACIL02)
          ND(123456)
          SIDE(1)
          ID(02)
          DUMPSPACE(2000)

    DEFINE STR(LIST01)
          SIZE(1000)
          PREFLIST(FACIL01,FACIL02)
          EXCLLIST(CACHE01)

    DEFINE STR(CACHE01)
          SIZE(1000)
          PREFLIST(FACIL02,FACIL01)
          EXCLLIST(LIST01)
/*
```

In the example, the programmer defined one list structure (LIST01) and one cache structure (CACHE01).

Attention: When defining a cache structure to DBRC, ensure that the name is identical to the name used in the CFRM policy.

Related tasks:

"Registering a cache structure name with DBRC"

Registering a cache structure name with DBRC

When you define DEDB areas to DBRC, use the same structure names defined in the CFRM policy to specify the structures that each DEDB area will use.

The DEDB area definitions and the corresponding structure names are then stored in the RECON data set. The structure names are entered in either the CFSTR1 or CFSTR2 parameter of the INIT.DBDS or CHANGE.DBDS command.

Restriction: The CFSTR2 parameter is not supported by multi-area structures. If you specify both CFSTR2 and MAS in INIT.DBDS, or use CHANGE.DBDS to apply CFSTR2 to DEDB area already defined by MAS, IMS will reject the DBRC command with either a DSP0141I or DSP0144I error message.

The following example of the INIT.DBDS command registers structure name `TSTDEDBAR1`.

Related concepts:

"Defining a VSO DEDB area" on page 226

"Examples of defining coupling facility structures" on page 232

Related tasks:

"Authorizing connections to DEDB VSO structures" on page 231

Defining a private buffer pool using the DFSVSMxx IMS.PROCLIB member

You define a private buffer pool by specifying the DEDB statement in the DFSVSMxx member of the IMS.PROCLIB data set.

For example, the following two statements define two private buffer pools:

```
DEDB=(POOL1,512,400,50,800,LKASID)
DEDB=(POOL2,8196,100,20,400,NOLKASID)
```

The first statement defines a pool that has a buffer size of 512, with an initial allocation of 400 buffers, increasing by 50, as needed, to a maximum of 800. This pool is used as a local cache, and buffer lookaside is performed for areas that share this pool.

The second statement defines a pool that has a buffer size of 8K, with an initial allocation of 100 buffers, increasing by 20, as needed, to a maximum of 400. This pool is used in the same fashion as the common buffer pool. There will be no lookaside performed.

If the customer does not define a private buffer pool, the default parameter values are described by the following statement:

```
DEDB=(poolname,XXX,64,16,512)
```

In the above statement:
- XXX is the CI size of the area to be opened.
- The initial buffer allocation is 64.
- The secondary allocation is 16.

- The maximum number of buffers for the pool is 512.
- The LKASID option is specified if it is specified in DBRC for the area.

Related concepts:

"Defining a VSO DEDB area" on page 226

Related reference:

↪ Defining Fast Path DEDB buffer pools for single-area structures (System Definition)

Defining a private buffer pool for a multi-area structure

You can define private buffer pools for multi-area structure using the DEDBMAS= keyword in the DFSVSMxx PROCLIB member.

Except for two additional parameters, *cisize* and *strname*, the parameters of the DEDBMAS keyword are the same as those of the DEDB= keyword in the DFSVSMxx PROCLIB member.

The *cisize* parameter specifies the size of the control interval of the area. The *strname* parameter specifies the name of the primary coupling facility structure. The structure must be defined in a coupling facility resource management (CFRM) administrative policy.

Related reference:

↪ Defining Fast Path DEDB buffer pools for multi-area structures (System Definition)

Acquiring and accessing data spaces for VSO DEDB areas

IMS allocates data spaces to accommodate VSO DEDB areas.

When a VSO DEDB area CI is preloaded or read for the first time, it is copied into a data space (or a coupling facility structure). Subsequent access to the data retrieves it from the data space rather than from DASD.

Acquiring a data space

IMS acquires data spaces for VSO areas when the VSO areas first open, but not before.

The maximum size of any VSO area data space is two gigabytes. Data spaces for preloaded VSO areas use the z/OS DREF (disabled reference) option. Data spaces for non-preloaded VSO areas do not use the DREF option.

DREF data spaces use central storage, but no auxiliary storage. Data spaces without the DREF option use central storage and auxiliary storage, if auxiliary storage is available.

IMS acquires additional data spaces for VSO areas, both with DREF and without, as needed.

Accessing a data space

IMS assigns areas to data spaces using a "first fit" algorithm.

The entire root addressable portion of an area (including independent overflow) resides in the data space. The sequential dependent portion does not reside in the data space.

The amount of space needed for an area in a data space is *(CI size) × (number of CIs per UOW) × ((number of UOWs in root addressable portion) + (number of UOWs in independent overflow portion))* rounded to the next 4 KB.

Expressed in terms of the parameters of the `DBDGEN AREA` statement, this formula is *(SIZE parameter value) × (UOW parameter value) × (ROOT parameter value)* rounded to the next 4 KB.

The actual amount of space in a data space available for an area (or areas) is two gigabytes (524,288 blocks, 4 KB each) minus an amount reserved by z/OS (from 0 to 4 KB) minus an amount used by IMS Fast Path (approximately 100 KB). You can use the `/DISPLAY FPVIRTUAL` command to determine the actual storage usage of a particular area.

During IMS control region initialization, IMS calls DBRC to request a list of all the areas that are defined as VSO. This list includes the PREOPEN or PRELOAD status of each VSO area. If VSO areas exist, IMS acquires the appropriate data spaces. Then IMS opens all areas defined with PREOPEN and opens and loads areas defined with PRELOAD. During a normal or emergency restart, the opening and loading of areas might occur after control region initialization, if you have changed the specifications of the FPOPN parameter in the IMS procedure.

Related concepts:

"Restrictions for using VSO DEDB areas" on page 226

Resource control and locking

Using VSO can reduce the number and duration of DEDB resource locking contentions by managing DEDB resource requests on a segment level and holding locks only until updated segments are returned to the data space.

Segment-level resource control and locking applies only to Get and Replace calls.

Without VSO, the VSAM CI (physical block) is the smallest available resource for DEDB resource request management and locking. If there is an update to any part of the CI, the lock is held until the whole CI is rewritten to DASD. No other requester is allowed access to any part of the CI until the first requester's lock is released.

With VSO, the database segment is the smallest available resource for DEDB resource request management and locking. Segment-level locking is available only for the root segment of a DEDB with a root-only structure, and when that root segment is a fixed-length segment. If processing options `R` or `G` are specified in the calling PCB, IMS can manage and control DEDB resource requests and serialize change at the segment level; for other processing options, IMS maintains VSAM CI locks. Segment locks are held only until the segment updates are applied to the CI in the data space. Other requesters for different segments in the same CI are allowed concurrent access.

A VSO DEDB resource request for a segment causes the entire CI to be copied into a common buffer. VSO manages the segment request at a level of control consistent with the request and its access intent. VSO also manages access to the CI that contains the segment but at the share level in all cases. A different user's subsequent request for a segment in the same CI accesses the image of the CI already in the buffer.

Updates to the data are applied directly to the CI in the buffer at the time of the update. Segment-level resource control and serialization provide integrity among multiple requesters. After an updated segment is committed and applied to the copy of the CI in the data space, other requesters are allowed access to the updated segment from the copy of the CI in the buffer.

If after a segment change the requester's updates are not committed for any reason, VSO copies the unchanged image of the segment from the data space to the CI in the buffer. VSO does not allow other requesters to access the segment until VSO completes the process of removing the uncommitted and canceled updates. Locking at the segment level is not supported for shared VSO areas. Only CI locking is supported.

When a compression routine is defined on the root segment of a DEDB with a root-only structure, and when that root segment is a fixed-length segment, its length becomes variable after being compressed. Replacing a compressed segment then requires a delete and an insert. In this case, segment level control and locking is not available.

Preopen areas and VSO areas in a data sharing environment

A VSO can be registered with different share levels.

A VSO can be registered with any of the following share levels:

SHARELVL(0)
Exclusive access: in a data sharing environment, any SHARELVL(0) area with the PREOPEN option (including VSO PREOPEN and VSO PRELOAD) is opened by the first IMS system to complete its control region initialization. IMS will not attempt to preopen the area for any other IMS.

SHARELVL(1)
One updater, many readers: in a data sharing environment, a SHARELVL(1) area with the PREOPEN option is preopened by all sharing IMS systems. The first IMS system to complete its control region initialization has update authorization; all others have read authorization.

If the SHARELVL(1) area is a VSO area, it is allocated to a data space by any IMS that opens the area. If the area is defined as VSO PREOPEN or VSO PRELOAD, it is allocated to a data space by all sharing IMS systems.

If the area is defined as VSO NOPREO NOPREL, it is allocated to a data space by all IMS systems, as each opens the area. The first IMS to access the area has update authorization; all others have read authorization.

SHARELVL(2)
Block-level sharing: a SHARELVL(2) area with at least one coupling facility structure name (CFSTR1) defined is shared at the block or control interval (CI) level within the scope of a single IRLM. Multiple IMS systems can be authorized for update or read processing if they are using the same IRLM.

SHARELVL(3)
Block-level sharing: a SHARELVL(3) area with at least one coupling facility structure name (CFSTR1) defined is shared at the block or control interval (CI) level within the scope of multiple IRLMs. Multiple IMS systems can be authorized for nonexclusive access.

Attention: Be careful when registering a VSO area as SHARELVL(1). Those systems that receive read-only authorization never see the updates made by the read/write system because all reads come from the data space (not from DASD, where updates are eventually written).

Input and output processing with VSO

The way IMS uses buffers, data spaces, and DASD in response to read and update requests when the Virtual Storage Option (VSO) is used might be different than when VSO is not used.

The following topics provide more information.

Input processing

When an application program issues a read request to a VSO area, IMS checks to see if the data is in the data space.

If the data is in the data space, it is copied from the data space into a common buffer and passed back to the application. If the data is not in the data space, IMS reads the CI from the area data set on DASD into a common buffer, copies the data into the data space, and passes the data back to the application.

For SHARELVL(2|3) VSO areas, Fast Path uses private buffer pools. Buffer lookaside is an option for these buffer pools. When a read request is issued against a SHARELVL(2|3) VSO area using a lookaside pool, a check is made to see if the requested data is in the pool. If the data is in the pool, a validity check to XES is made. If the data is valid, it is passed back to the application from the local buffer. If the data is not found in the local buffer pool or XES indicates that the data in the pool is not valid, the data is read from the coupling facility structure and passed to the application. When the buffer pool specifies the no-lookaside option, every request for data goes to the coupling facility.

For those areas that are defined as load-on-demand (using the VSO and NOPREL options), the first access to the CI is from DASD. The data is copied to the data space and then subsequent reads for this CI retrieve the data from the data space rather than from DASD. For those areas that are defined using the VSO and PRELOAD options, all access to CIs comes from the data space.

Whether the data comes from DASD or from the data space is transparent to the processing done by application programs.

Output processing

During phase 1 of synchronization point processing, VSO data is treated the same as non-VSO data. The use of VSO is transparent to logging.

During phase 2 of the synchronization point processing, VSO and non-VSO data are treated differently. For VSO data, the updated data is copied to the data space, the lock is released and the buffer is returned to the available queue. The relative byte address (RBA) of the updated CI is maintained in a bitmap. If the RBA is already in the bitmap from a previous update, only one copy of the RBA is kept. At interval timer, the updated CIs are written to DASD. This batching of updates reduces the amount of output processing for CIs that are frequently updated. While the updates are being written to DASD, they are still available for application programs to read or update because copies of the data are made within the data space just before it is written.

For SHARELVL(2|3) VSO areas, the output thread process is used to write updated CIs to the coupling facility structures. When the write is complete, the lock is released. XES maintains the updated status of the data in the directory entry for the CI.

The PRELOAD option

The loading of one area takes place asynchronously with the loading of any others. The loading of an area is (or can be) concurrent with an application program's accesses to that area.

If the CI requested by the application program has been loaded into the data space, it is retrieved from the data space. If the requested CI has not yet been loaded into the data space, it is obtained from DASD and UOW locking is used to maintain data integrity.

The preload process for SHARELVL(2|3) VSO areas is similar to that of SHARELVL(0|1). Multiple preloads can be run concurrently, and also concurrent with application processing. The locking, however, is different. SHARELVL(2|3) Areas that are loaded into coupling facility structures use CI locking instead of UOW locking. The load process into the coupling facility is done one CI at a time.

If a read error occurs during preloading, an error message flags the error, but the preload process continues. If a subsequent application program call accesses a CI that was not loaded into the data space due to a read error, the CI request goes out to DASD. If the read error occurs again, the application program receives an "A0" status code, just as with non-VSO applications. If instead the access to DASD is successful this time, the CI is loaded into the data space.

Related concepts:

"Read errors"

I/O error processing

Using VSO increases the availability of data when write errors occur.

When a CI for a VSO area has been put into a data space, the CI is available from that data space as long as IMS is active, even if a write error occurs when an update to the CI is being written to DASD.

Write errors:

When a write error occurs, IMS creates an error queue element (EQE) for the CI in error.

For VSO areas, all read requests are satisfied by reading the data from the data space. Therefore, as long as the area continues to reside in the data space, the CI that had the write error continues to be available. When the area is removed from the data space, the CI is no longer available and any request for the CI receives an "AO" status code.

Read errors:

For VSO areas, the first access to a CI causes it to be read from DASD and copied into the data space. From then on, all read requests are satisfied from the data space. If there is a read error from the data space, z/OS abends.

For VSO areas that have been defined with the PRELOAD option, the data is preloaded into the data space; therefore, all read requests are satisfied from the data space.

To provide for additional availability, SHARELVL(2|3) VSO areas support multiple structures per area. If a read error occurs from one of the structures, the read is attempted from the second structure. If there is only one structure defined and a read error occurs, an AO status code is returned to the application.

There is a maximum of three read errors allowed from a structure. When the maximum is reached and there is only one structure defined, the area is stopped and the structure is disconnected.

When the maximum is reached and there are two structures defined, the structure in error is disconnected. The one remaining structure is used. If a write error to a structure occurs, the CI in error is deleted from the structure and written to DASD. The delete of the CI is done from the sharing partners. If none of the sharers can delete the CI from the structure, an EQE is generated and the CI is deactivated. A maximum of three write errors are allowed to a structure. If there are two structures defined and one of them reaches the maximum allowed, it is disconnected.

Related concepts:

"The PRELOAD option" on page 238

Castout thresholds for CIs in VSO areas

The castout thresholds for CIs in VSO areas are timers. When the timer expires, a CI is cast out of the VSO area.

The castout threshold for CIs in VSO areas is a timer after which a CI is castout of VSO areas. You can configure the thresholds for VSO areas with different number of CIs by specifying the following parameters in the DFSDFxxx member of the IMS PROCLIB data set under the <SECTION=FASTPATH> section:

VSO1THLD=
Specifies the castout threshold time in seconds for a VSO area with less than or equal to 800 control intervals (CIs). The valid range is 1 - 300. The default value is 300.

VSO2THLD=
Specifies the castout threshold time in seconds for a VSO area with 801 - 3500 CIs. The valid range is 1 - 300. The default value is 240.

VSO3THLD=
Specifies the castout threshold time in seconds for a VSO area with greater than 3500 CIs. The valid range is 1 - 300. The default value is 180.

Related reference:

FASTPATH section of the DFSDFxxx member (System Definition)

Checkpoint processing

During a system checkpoint, all of the VSO area updates that are in the data space are written to DASD. All of the updated CIs in the CF structures are also written to DASD.

Only CIs that have been updated are written. Also, all updates that are in progress are allowed to complete before checkpoint processing continues.

VSO options across IMS restart

For all types of IMS restart except XRF takeover (cold start, warm start, emergency restart, COLDBASE, COLDCOMM and COLDSYS emergency restart), the VSO options in effect after restart are those defined to DBRC.

In the case of the XRF takeover, the VSO options in effect after the takeover are the same as those in effect for the active IMS prior to the failure that caused the XRF takeover.

Emergency restart processing

Recovery of VSO areas across IMS or z/OS failures is similar to recovery of non-VSO areas. IMS examines the log records, from a previous system checkpoint to the end of the log, to determine if there are any committed updates that were not written to DASD before the failure.

If any such committed updates are found, IMS will REDO them (apply the update to the CI and write the updated CI to DASD). Because VSO updates are batched together during normal processing, VSO areas are likely to require more REDO processing than non-VSO areas.

During emergency restart log processing, IMS tracks VSO area updates differently depending on the share level of the VSO area. For share levels 0 and 1, IMS uses data spaces to track VSO area updates. For share levels 2 and 3, IMS uses a buffer in memory to track VSO area updates.

IMS also obtains a single non-DREF data space which it releases at the end of restart. If restart log processing is unable to get the data space or main storage resources it needs to perform VSO REDO processing, the area is stopped and marked as "recovery needed".

By default, at the end of emergency restart, IMS opens areas defined with the PREOPEN or PRELOAD options. IMS then loads areas with the PRELOAD option into a data space or coupling facility structure. You can alter this behavior by using the FPOPN keyword of the IMS procedure to have IMS restore all VSO DEDB areas to their open or closed state at the time of the failure.

VSO areas without the PREOPEN or PRELOAD options are assigned to a data space during the first access following emergency restart.

After an emergency restart, the VSO options and PREOPEN|NOPREO options in effect for an area are those that are defined to DBRC, which may not match those in effect at the time of the failure. For example, a non-shared VSO area removed from virtual storage by the /VUNLOAD command before the failure, is restored to the data space after the emergency restart. For shared VSO areas, the area remains unloaded until the next /STA AREA command is issued for it.

Related concepts:

"VSO options with XRF"

VSO options with XRF

During the tracking and takeover phases on the alternate IMS, log records are processed in the same manner as during active IMS emergency restart (from a previous active system checkpoint to the end of the log).

The alternate IMS uses the log records to determine which areas have committed updates that were not written to DASD before the failure of the active IMS. If any such committed updates are found, the alternate will REDO them, following the same process as for active IMS emergency restart.

During tracking, the alternate uses data spaces to track VSO area updates: in addition to the data space resources used for VSO areas, the alternate obtains a single non-DREF data space which it releases at the end of takeover. If XRF tracking or takeover is unable to get the data space or main storage resources it needs to perform VSO REDO processing, the area is stopped and marked "recovery needed".

Following an XRF takeover, areas that were open or in the data space remain open or in the data space. The VSO options and PREOPEN|NOPREO options that were in effect for the active IMS before the takeover remain in effect on the alternate (the new active) after the takeover. Note that these options may not match those defined to DBRC. For example, a VSO area removed from virtual storage by the /VUNLOAD command before the takeover is *not* restored to the data space after the takeover.

VSO areas defined with the preload option are preloaded at the end of the XRF takeover. In most cases, dependent regions can access the area before preloading begins, but until preloading completes, some area read requests may have to be retrieved from DASD.

Related concepts:

"Emergency restart processing" on page 240

Fast Path synchronization points

MSDBs and DEDBs are not updated during application program processing, but the updates are kept in buffers until a sync point. Output messages are not sent until the message response is logged.

The Fast Path sync point is defined as the next GU call for a message-driven program, or a SYNC or CHKP call for a BMP using Fast Path facilities.

Sync point processing occurs in two phases.

Related concepts:

"Mixed mode processing" on page 218

Phase 1 - build log record

DEDB updates and verified MSDB records are written in system log records. All DEDB updates for the current sync point are chained together as a series of log records. Resource contentions, deadlocks, out-of-space conditions, and MSDB verify failures are discovered here.

Phase 2 - write record to system log

Database and message records are written to the IMS system log. After logging, MSDB records are updated, the DEDB updates begin, and messages are sent to the terminals.

DEDB updates are applied with a type of asynchronous processing called an output thread. Until the DEDB changes are made, any program that tries to access unwritten segments is put in a wait state.

If, during application processing, a Fast Path program issues a call to a database other than MSDB or DEDB, or to an alternate PCB, the processing is serialized with full function events. This can affect the performance of the Fast Path program. In the case of a BMP or MPP making a call to a Fast Path database, the Fast Path resources are held, and the throughput for Fast Path programs needing these resources can be affected.

Managing I/O errors and long wait times

When a database write I/O error occurs in single area data sets (ADS), IMS copies the buffer contents of the error control interval (CI) to a virtual buffer. A subsequent DL/I request causes the error CI to be read back into the buffer pool.

The write error information and buffers are maintained across restarts, allowing recovery to be deferred to a convenient time. I/O error retry is automatically performed at database close time and at system checkpoint. If the retry is successful, the error condition no longer exists and recovery is not needed.

When a database read I/O error occurs, IMS creates a non-permanent EQE associated with the area data set (ADS) recording the RBA of the error. If there are other ADSs available, IMS retries the read using a different ADS. If there is only a single ADS, or if the read fails on all ADSs, the application program receives an 'AO' status code. The presence of the EQE prevents subsequent access to the same CI in this ADS. Any attempt to access the CI receives an immediate I/O error indication.

For MADS, the I/O is attempted against a different ADS. Up to three distinct I/O errors can be recorded per ADS. On the fourth error, IMS internally stops the ADS. If this is the only ADS for the area, the area is stopped.

EQEs are temporary and do not persist across IMS restarts or the opening and closing of an area. EQEs are not recorded in DBRC or in the DMAC on DASD. A write error eliminates the read EQEs and resets the counter.

The process to create the read EQE also reads the DMAC (second CI) from DASD. If the DMAC read fails, which it might if the failure is device level, IMS internally stops the ADS. ADS stop processing involves a physical close of the area, which involves a DMAC (second CI) write. If this process fails, and the ADS being closed is the only ADS for the area, the area is stopped and flagged in DBRC as 'recovery needed'.

Multiple Area Data Sets I/O Timing (MADSIOT) helps you avoid the excessively long wait times (also known as a *long busy*) that can occur while a RAMAC disk array performs internal recovery processing.

Restriction: MADSIOT applies only to multiple area data sets (MADS). For single area data sets (ADS), IMS treats the long busy condition as a permanent I/O error handled by the Fast Path I/O toleration function. The MADSIOT function works only on a system that supports the long busy state.

To invoke MADSIOT, you must define the MADSIOT keyword on the DFSVSMxx PROCLIB member. The /STA MADSIOT and /DIS AREA MADSIOT commands serve to start and monitor the MADSIOT function.

Additionally, MADSIOT requires the use of a Coupling Facility (CFLEVEL=1 or later) list structure in a sysplex environment. MADSIOT uses this Coupling Facility

to store information required for DB recovery. You must use the CFRM policy to define the list structure name, size, attributes, and location.

The following table shows the required CFRM list structure storage sizes when CFLEVEL=12 and the number of changed CIs is 1 000, 5 000, 20 000, and 30 000. The storage sizes differ at different CFLEVELs.

Table 56. Required CFRM list structure storage sizes for CFLEVEL=12

Altered number of CIs (entrynum)	Required storage size (listheadernum=50)
1 000	1 792 KB
5 000	3 584 KB
20 000	11 008 KB
30 000	15 616 KB

You can estimate CFRM list structure storage sizes tailored to your installation using an online tool: the IBM System z Coupling Facility Structure Sizer Tool (CFSizer). CFSizer is available for you to use at the following website: www.ibm.com/servers/eserver/zseries/cfsizer/, or search for "CFSizer" at the IBM website: www.ibm.com.

Related tasks:

⇨ Defining the CFRM policy (System Administration)

Related reference:

⇨ /START MADSIOT command (Commands)

⇨ /DISPLAY AREA command (Commands)

⇨ DFSVSMxx member of the IMS PROCLIB data set (System Definition)

Registering Fast Path databases in DBRC

Although Fast Path databases are not required to be registered in DBRC in order for the error handling to work, registration is highly recommended. In some situations, registration is required.

If an error occurs on a database that is not registered and the system stops, the database could be damaged if the system is restarted and a /DBR command is not issued before accessing the database. The restart causes the error buffers to be restored as they were when the system stopped.

In the following situations, Fast Path databases must be registered to DBRC:
- Virtual Storage Option (VSO)
- Preopening DEDB areas
- Preloading a VSO DEDB area control interval (CI)
- High speed sequential processing (HSSP) image copy (IC)
- Multiple area data set (MADS)
- Full segment logging
- Database quiesce
- Data sharing
- Shared Virtual Storage Option (SVSO) areas that reside on a z/OS coupling facility structure

Chapter 17. Creating logical relationships

Logical relationships resolve conflicts in the way application programs need to view segments in the database.

With logical relationships, application programs can access:
- Segment types in an order other than the one defined by the hierarchy
- A data structure that contains segments from more than one physical database.

An alternative to using logical relationships to resolve the different needs of applications is to create separate databases or carry duplicate data in a single database. However, in both cases this creates duplicate data. Avoid duplicate data because:
- Extra maintenance is required when duplicate data exists because both sets of data must be kept up to date. In addition, updates must be done simultaneously to maintain data consistency.
- Extra space is required on DASD to hold duplicate data.

By establishing a path between two segment types, logical relationships eliminate the need to store duplicate data.

To establish a logical relationship, three segment types are always defined:

A physical parent

A logical parent

A logical child

The following database types support logical relationships:
- HISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM

Two databases, one for orders that a customer has placed and one for items that can be ordered, are called ORDER and ITEM. The ORDER database contains information about customers, orders, and delivery. The ITEM database contains information about inventory.

If an application program needs data from both databases, this can be done by defining a logical relationship between the two databases. As shown in the following figure, a path can be established between the ORDER and ITEM databases using a segment type, called a logical child segment, that points into the ITEM database. The following figure shows a simple implementation of a logical relationship. In this case, ORDER is the physical parent of ORDITEM. ORDITEM is the physical child of ORDER and the logical child of ITEM.

In a logical relationship, there is a logical parent segment type and it is the segment type pointed to by the logical child. In this example, ITEM is the logical parent of ORDITEM. ORDITEM establishes the path or connection between the two segment types. If an application program now enters the ORDER database, it

can access data in the ITEM database by following the pointer in the logical child segment from the ORDER to the ITEM database.

Figure 70. A simple logical relationship

The physical parent and logical parent are the two segment types between which the path is established. The logical child is the segment type that establishes the path. The path established by the logical child is created using pointers.

Related concepts:

"Adding logical relationships" on page 514

Secondary indexes versus logical relationships

Both secondary indexes and logical relationships provide logical data structures, which are hierarchical data structures that are different from the data structure represented by the physical DBD. The decision about which type to use is based primarily on how application programs need to process the data. Fast Path supports only secondary indexes.

When to use a secondary index

In analyzing application requirements, if more than one candidate exists for the sequence field of a segment, use a secondary index. Choose one sequence field to be defined in the physical DBD. Then set up a secondary index to allow processing of the same segment in another sequence. For the example shown in the following figure, access the customer segment that follows in both customer number (CUSTNO) and customer name (CUSTNAME) sequence. To do this, define CUSTNO as the sequence field in the physical DBD and then define a secondary index that processes CUSTOMER segments in CUSTNAME sequence.

Figure 71. Fields in the CUSTOMER segment

When to use a logical relationship

If you have applications such as a bill-of-materials using a recursive structure, use a logical relationship. A recursive structure exists when there is a many-to-many association between two segments in the same physical hierarchy. For example, in the segments shown in the following figure, the assembly "car" is composed of many parts, one of which is an engine. However, the engine is itself an assembly composed of many parts.
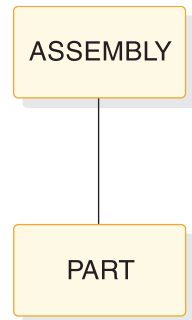
Figure 72. Assembly and parts as examples to demonstrate segments logical relationship

Finally, you can have application requirements that result in a segment that appears to have two parents. In the example shown in the following figure, the customer database keeps track of orders (CUSTORDN). Each order can have one or more line items (ORDLINE), with each line item specifying one product (PROD) and model (MODEL). In the product database, many outstanding line item requests can exist for a given model. This type of relationship is called a many-to-many relationship and is handled in IMS through a logical relationship.
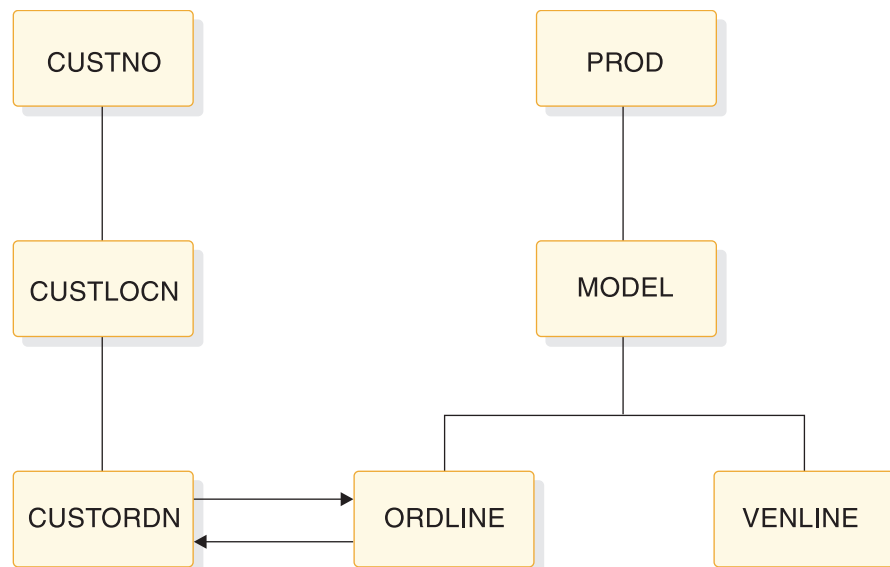
Figure 73. Example of a segment that appears to have two parents

Related concepts:
"Recursive structures: same database logical relationships" on page 268

Logical relationship types

Three types of logical relationships are discussed in this topic.

The three types of logical relationships are:

Unidirectional logical relationships
Bidirectional physically paired logical relationships
Bidirectional virtually paired logical relationships

Unidirectional logical relationships

A unidirectional relationship links two segment types, a logical child and its logical parent, in one direction. A one-way path is established using a pointer in the logical child. The following figure shows a unidirectional relationship that has been established between the ORDER and ITEM databases. A unidirectional relationship can be established between two segment types in the same or different databases. Typically, however, a unidirectional relationship is created between two segment types in different databases. In the figure, the logical relationship can be used to cross from the ORDER to the ITEM database. It cannot be used to cross from the ITEM to the ORDER database, because the ITEM segment does not point to the ORDER database.
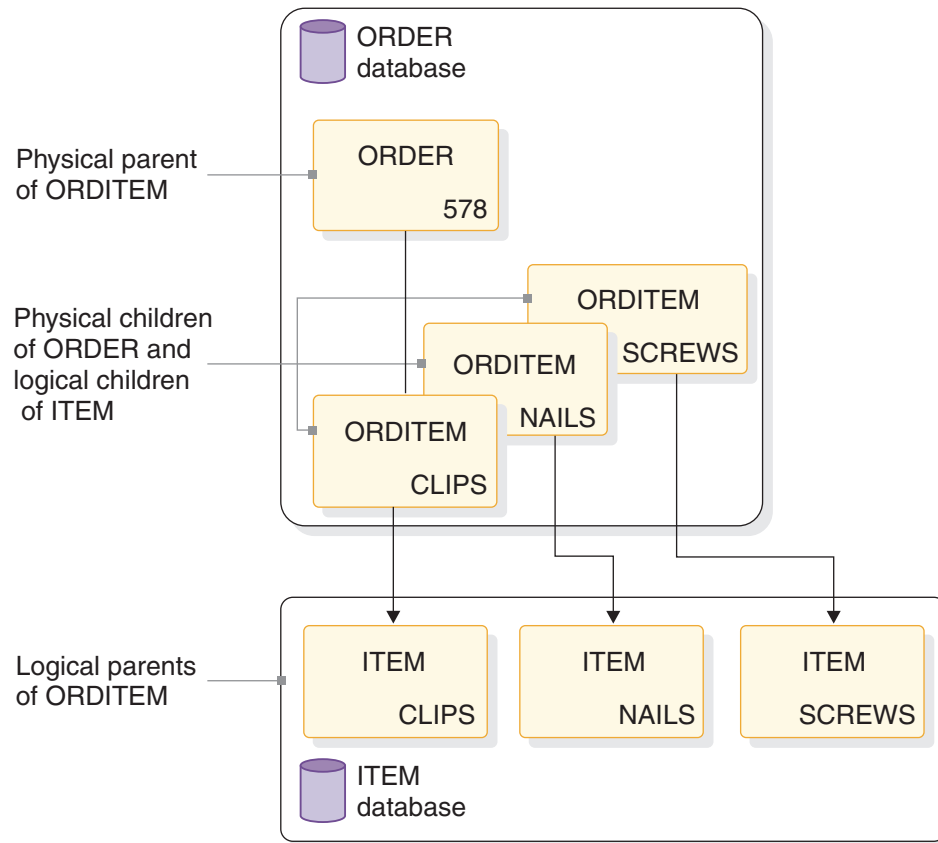
Figure 74. Unidirectional logical relationship

It is possible to establish two unidirectional relationships, as shown in the following figure. Then either physical database can be entered and the logical child in either can be used to cross to the other physical database. However, IMS treats

each unidirectional relationship as a one-way path. It does not maintain data on both paths. If data in one database is inserted, deleted, or replaced, the corresponding data in the other database is not updated. If, for example, DL/I replaces ORDITEM-SCREWS under ORDER-578, ITEMORD-578 under ITEM-SCREWS is not replaced. This maintenance problem does not exist in both bidirectional physically paired-logical and bidirectional virtually paired-logical relationships. Both relationship types are discussed next. IMS allows either physical database to be entered and updated and automatically updates the corresponding data in the other database.
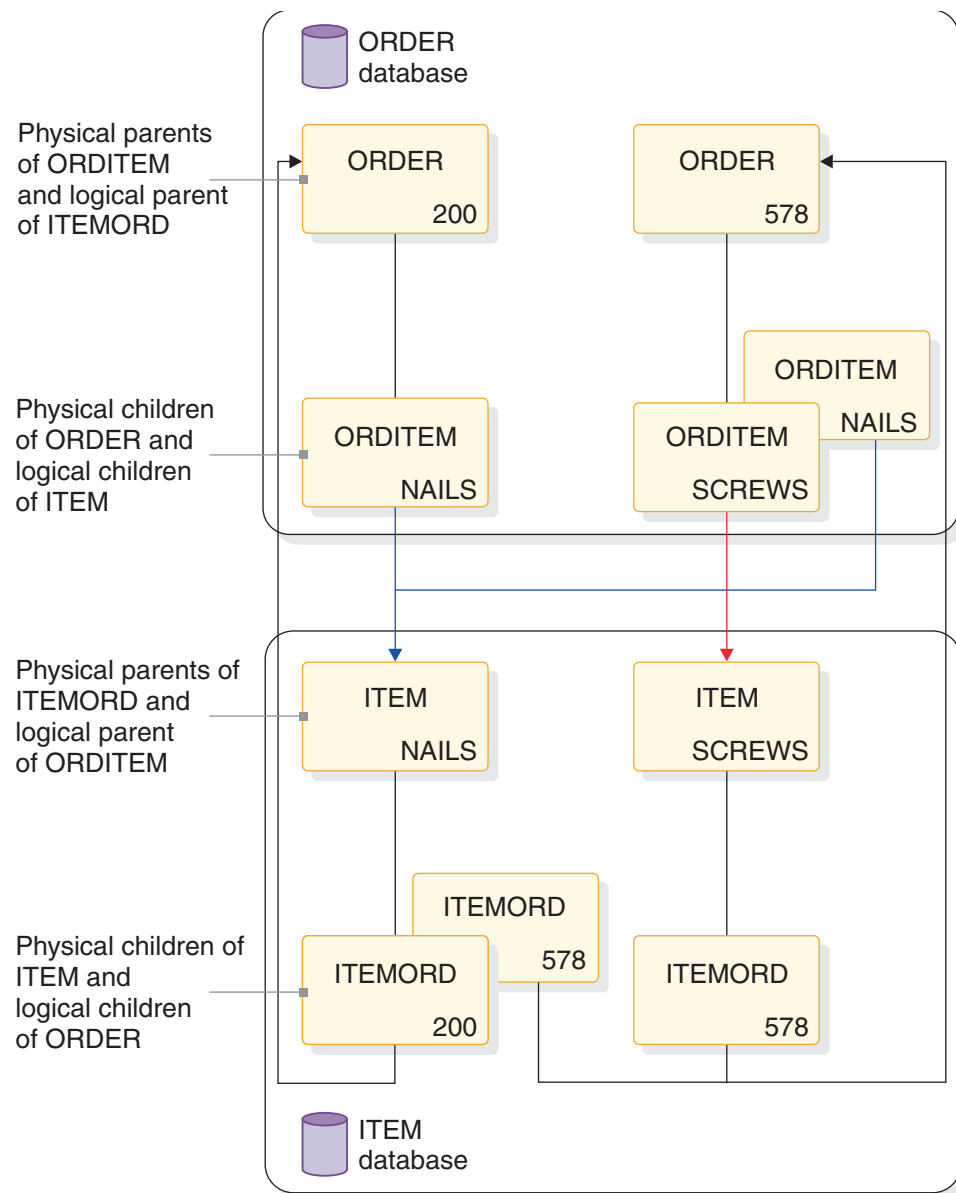
Figure 75. Two unidirectional logical relationships

Bidirectional physically paired logical relationship

A bidirectional physically paired relationship links two segment types, a logical child and its logical parent, in two directions. A two-way path is established using pointers in the logical child segments. The following figure shows a bidirectional

physically paired logical relationship that has been established between the ORDER and ITEM databases.
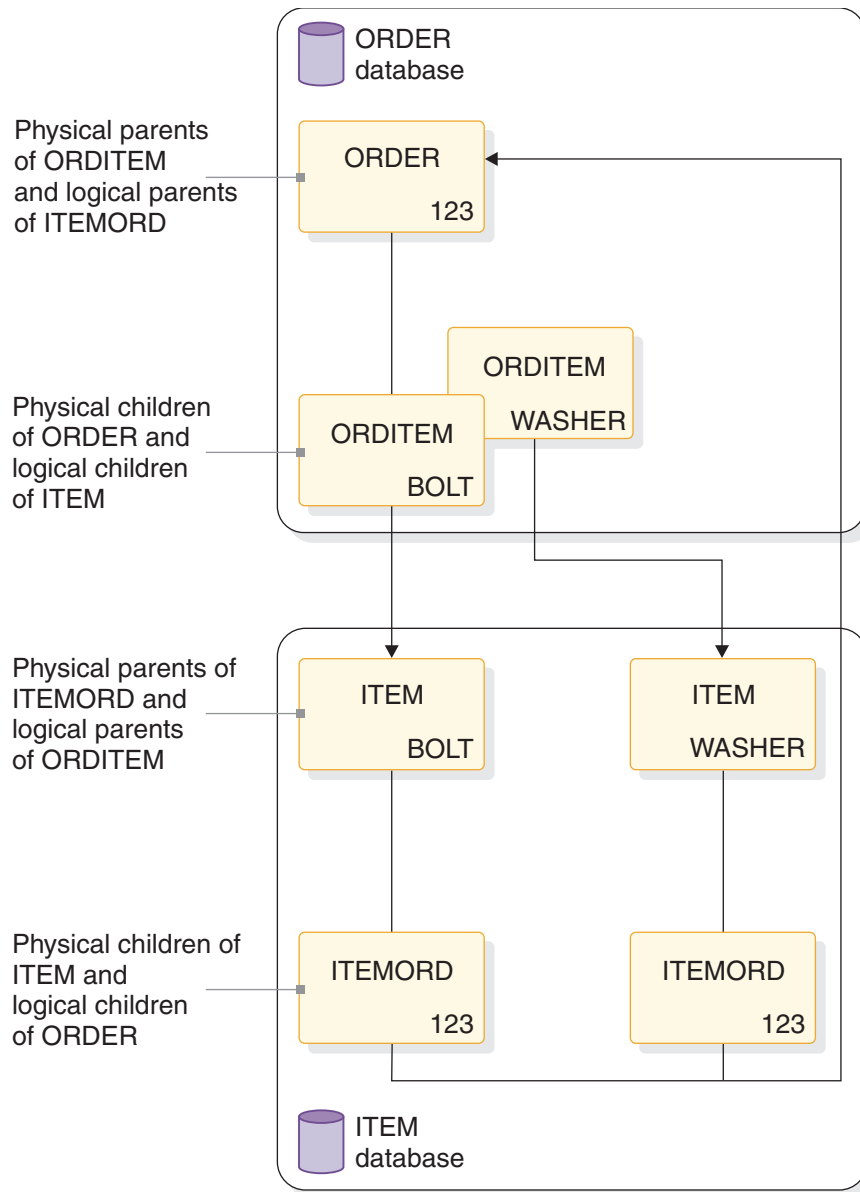
Figure 76. Bidirectional physically paired logical relationship

Like the other types of logical relationships, a physically paired relationship can be established between two segment types in the same or different databases. The relationship shown in the figure above allows either the ORDER or the ITEM database to be entered. When either database is entered, a path exists using the logical child to cross from one database to the other.

In a physically paired relationship, a logical child is stored in both databases. However, if the logical child has dependents, they are only stored in one database. For example, IMS maintains data in both paths in physically paired relationships. In the figure above, if ORDER 123 is deleted from the ORDER database, IMS deletes from the ITEM database all ITEMORD segments that point to the ORDER 123 segment. If data is changed in a logical child segment, IMS changes the data in

its paired logical child segment. Or if a logical child segment is inserted into one database, IMS inserts a paired logical child segment into the other database.

With physical pairing, the logical child is duplicate data, so there is some increase in storage requirements. In addition, there is some extra maintenance required because IMS maintains data on two paths. In the next type of logical relationship examined, this extra space and maintenance do not exist; however, IMS still allows you to enter either database. IMS also performs the maintenance for you.

Bidirectional virtually paired logical relationship

A bidirectional virtually paired relationship is like a bidirectional physically paired relationship in that:
* It links two segment types, a logical child and its logical parent, in two directions, establishing a two-way path.
* It can be established between two segment types in the same or different databases.

The figure below shows a bidirectional virtually paired relationship between the ORDER and ITEM databases. Note that although there is a two-way path, a logical child segment exists only in the ORDER database. Going from the ORDER to the ITEM database, IMS uses the pointer in the logical child segment. Going from the ITEM to the ORDER database, IMS uses the pointer in the logical parent, as well as the pointer in the logical child segment.
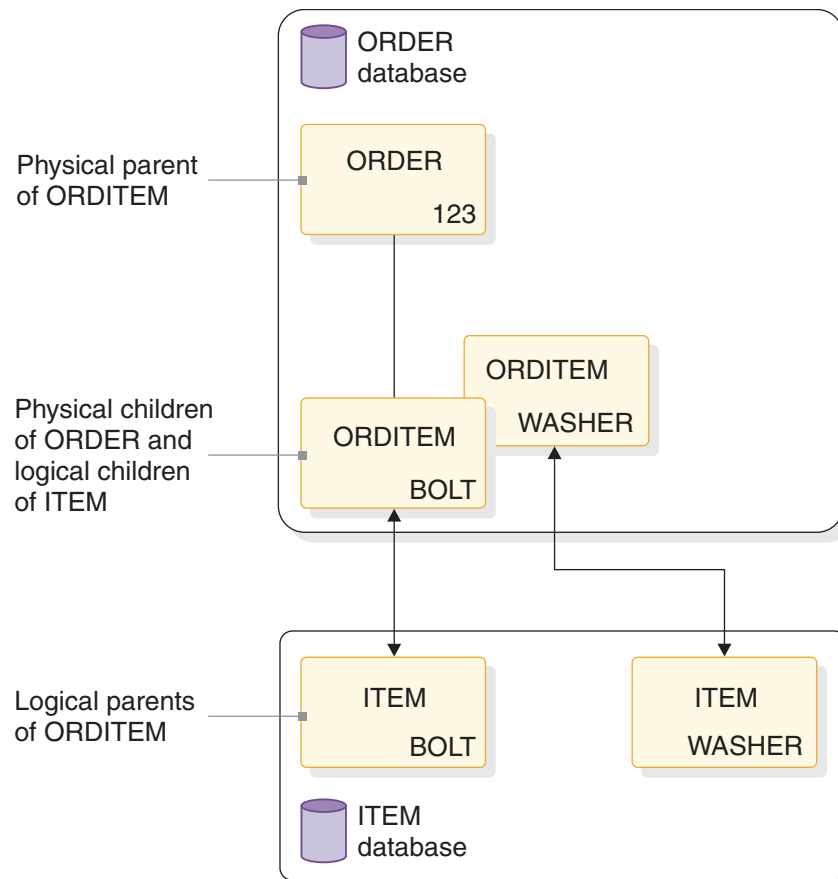
Figure 77. Bidirectionally virtually paired logical relationship

To define a virtually paired relationship, two logical child segment types are defined in the physical databases involved in the logical relationship. Only one logical child is actually placed in storage. The logical child defined and put in storage is called the *real logical child*. The logical child defined but not put in storage is called the *virtual logical child*.

IMS maintains data in both paths in a virtually paired relationship. However, because there is only one logical child segment, maintenance is simpler than it is in a physically paired relationship. When, for instance, a new ORDER segment is inserted, only one logical child segment has to be inserted. For a replace, the data only has to be changed in one segment. For a delete, the logical child segment is deleted from both paths.

Note the trade-off between physical and virtual pairing. With virtual pairing, there is no duplicate logical child and maintenance of paired logical children. However, virtual pairing requires the use and maintenance of additional pointers, called logical twin pointers.

Related reference:

Logical relationship pointer types

In all logical relationships the logical child establishes a path between two segment types. The path is established by use of pointers.

For HALDB databases, consider the following:

- Logical relationships are not allowed between HALDB databases and non-HALDB databases.
- Direct pointers and indirect pointers are used.
- Unidirectional relationships and bidirectional, physically paired relationships are supported for HALDB databases.
- Physical parent pointers are always present in PHDAM and PHIDAM segments.

Logical parent pointer

The pointer from the logical child to its logical parent is called a logical parent (LP) pointer. This pointer must be a symbolic pointer when it is pointing into a HISAM database. It can be either a *direct* or a *symbolic* pointer when it is pointing into an HDAM or a HIDAM database. PHDAM or PHIDAM databases require direct pointers.

A direct pointer consists of the direct address of the segment being pointed to, and it can only be used to point into a database where a segment, once stored, is not moved. This means the logical parent segment must be in an HD (HDAM, PHDAM, HIDAM, or PHIDAM) database, since the logical child points to the logical parent segment. The logical child segment, which contains the pointer, can be in a HISAM or an HD database except in the case of HALDB. In the HALDB case, the logical child segment must be in an HD (PHDAM or PHIDAM) database. A direct LP pointer is stored in the logical child's prefix, along with any other pointers, and is four bytes long. The following figure shows the use of a direct LP pointer. In a HISAM database, pointers are not required between segments because they are stored physically adjacent to each other in hierarchical sequence. Therefore, the only time direct pointers will exist in a HISAM database is when

there is a logical relationship using direct pointers pointing into an HD database.
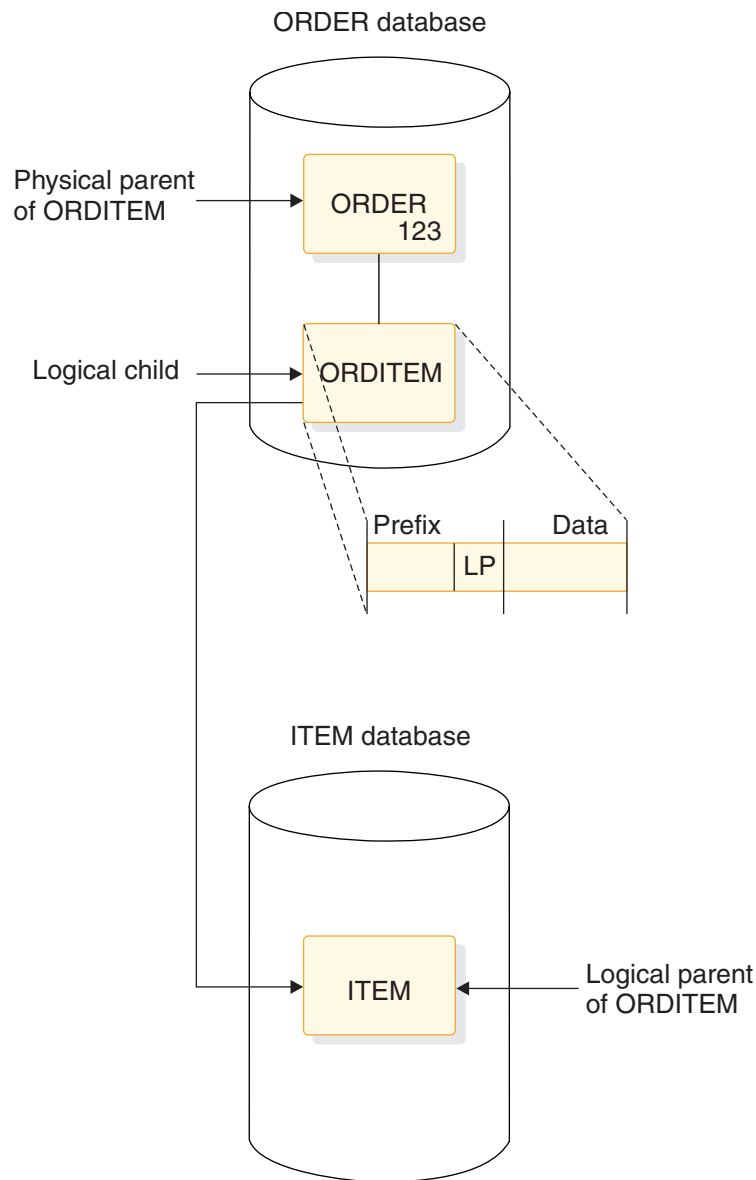
Figure 78. Direct logical parent (LP) pointer

In the preceding figure, the direct LP pointer points from the logical child ORDITEM to the logical parent ITEM. Because it is direct, the LP pointer can only point to an HD database. However, the LP pointer can "exist" in a HISAM or an HD database. The LP pointer is in the prefix of the logical child and consists of the 4-byte direct address of the logical parent.

A symbolic LP pointer, which consists of the logical parent's concatenated key (LPCK), can be used to point into a HISAM or HD database. The following figure illustrates how to use a symbolic LP pointer. The logical child ORDITEM points to the ITEM segment for BOLT. BOLT is therefore stored in ORDITEM in the LPCK. A symbolic LP pointer is stored in the first part of the data portion in the logical child segment.

Note: The LPCK part of the logical child segment is considered non-replaceable and is not checked to see whether the I/O area is changed. When the LPCK is virtual, checking for a change in the I/O area causes a performance problem. Changing the LPCK in the I/O area does not cause the REPL call to fail. However, the LPCK is not changed in the logical child segment.

With symbolic pointers, if the database the logical parent is in is HISAM or HIDAM, IMS uses the symbolic pointer to access the index to find the correct logical parent segment. If the database containing the logical parent is HDAM, the symbolic pointer must be changed by the randomizing module into a block and RAP address to find the logical parent segment. IMS accesses a logical parent faster when direct pointing is used.

Although the figures show the LP pointer in a unidirectional relationship, it works exactly the same way in all three types of logical relationships.

The following figure shows an example of a symbolic logical parent pointer.

ORDER database

Physical parent of ORDITEM

ORDER 123

Logical child

ORDITEM

Prefix	Data	
	LPCK	

Index or randomizing module

ITEM database

ITEM
BOLT

Logical parent of ORDITEM
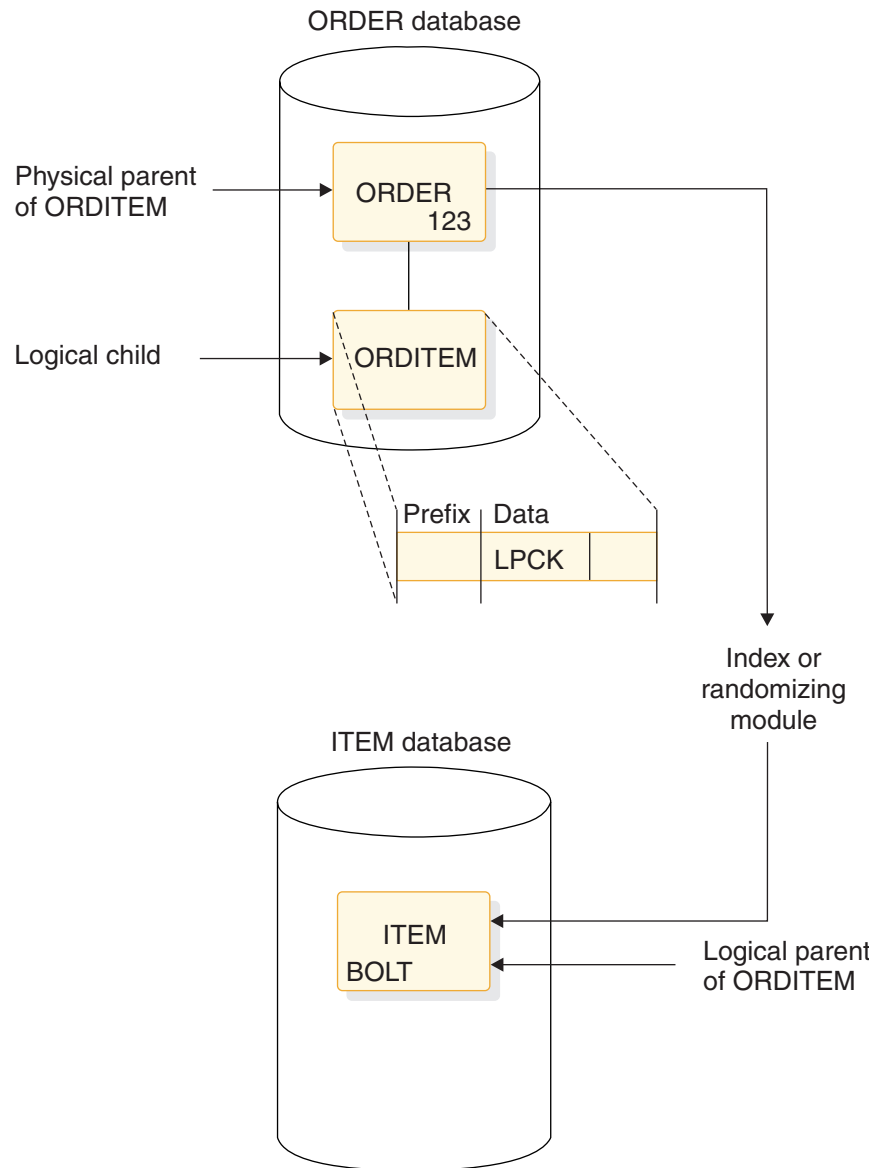
Figure 79. Symbolic logical parent (LP) pointer

In the preceding figure, the *symbolic* LP pointer points from the logical child ORDITEM to the logical parent ITEM. With symbolic pointing, the ORDER and ITEM databases can be either HISAM or HD. The LPCK, which is in the first part of the data portion of the logical child, functions as a pointer from the logical child to the logical parent, and is the pointer used in the logical child.

Logical child pointer

Logical child pointers are only used in logical relationships with virtual pairing. When virtual pairing is used, there is only one logical child on DASD, called the real logical child. This logical child has an LP pointer. The LP pointer can be symbolic or direct. In the ORDER and ITEM databases you have seen, the LP pointer allows you to go from the database containing the logical child to the database containing the logical parent. To enter either database and cross to the other with virtual pairing, you use a logical child pointer in the logical parent. Two types of logical child pointers can be used:

- Logical child first (LCF) pointers, or
- The *combination* of logical child first (LCF) and logical child last (LCL) pointers

The LCF pointer points from a logical parent to the first occurrence of each of its logical child types. The LCL pointer points to the last occurrence of the logical child segment type for which it is specified. A LCL pointer can only be specified in conjunction with a LCF pointer. The following figure shows the use of the LCF pointer. These pointers allow you to cross from the ITEM database to the logical child ORDITEM in the ORDER database. However, although you are able to cross databases using the logical child pointer, you have only gone from ITEM to the logical child ORDITEM. To go to the ORDER segment, use the physical parent pointer.

LCF and LCL pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. This means the logical child segment, the segment being pointed to, must be in an HD database. The logical parent can be in a HISAM or HD database. If the logical parent is in a HISAM database, the logical child segment must point to it using a symbolic pointer. LCF and LCL pointers are stored in the logical parent's prefix, along with any other pointers. The following figure shows an LCF pointer.
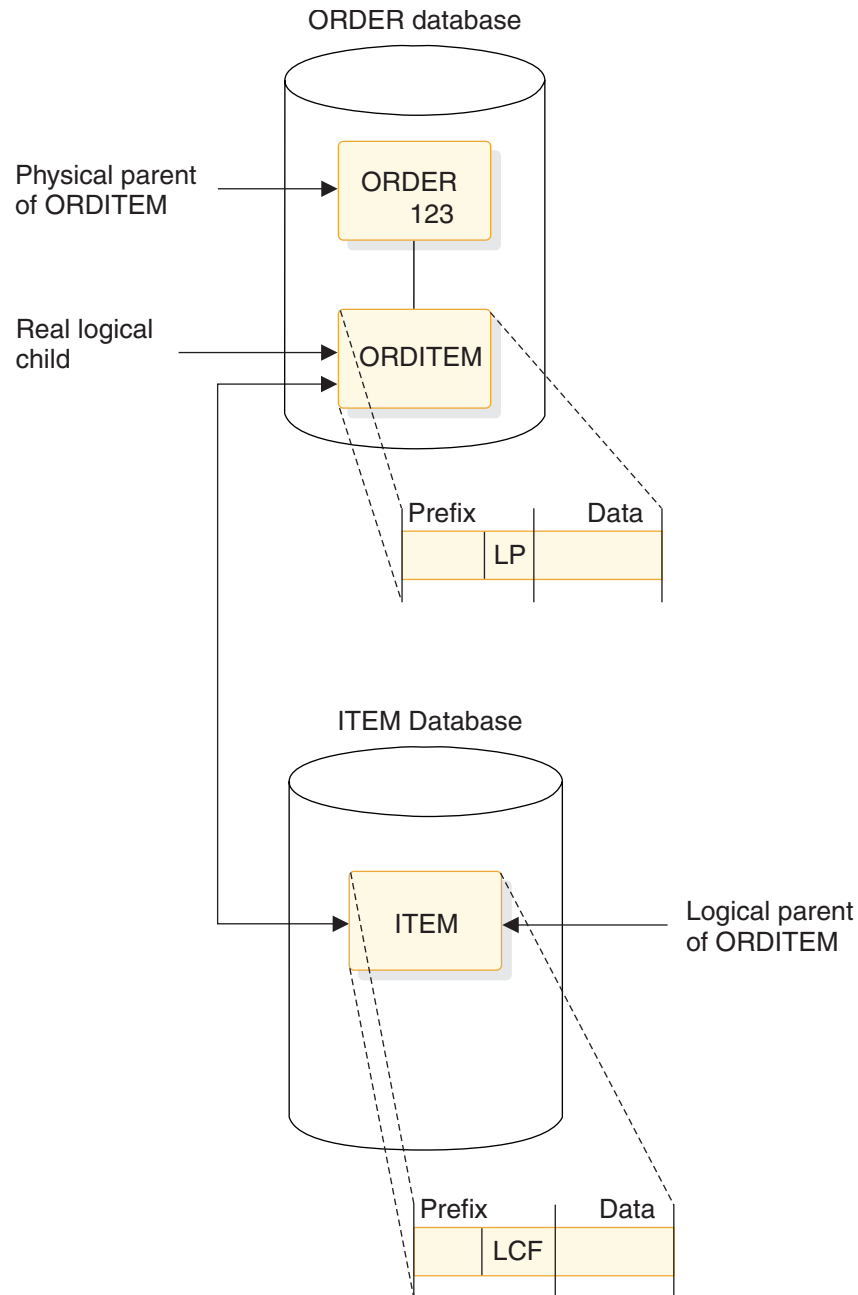
Figure 80. Logical child first (LCF) pointer (used in virtual pairing only)

In the preceding figure, the LCF pointer points from the logical parent ITEM to the logical child ORDITEM. Because it is a direct pointer, it can only point to an HD database, although, it can exist in a HISAM or an HD database. The LCF pointer is in the prefix of the logical parent and consists of the 4-byte RBA of the logical child.

Physical parent pointer

Physical parent (PP) pointers point from a segment to its physical parent. They are generated automatically by IMS for all HD databases involved in logical relationships. PP pointers are put in the prefix of all logical child and logical parent segments. They are also put in the prefix of all segments on which a logical child

or logical parent segment is dependent in its physical database. This creates a path from a logical child or its logical parent back up to the root segment on which it is dependent. Because all segments on which a logical child or logical parent is dependent are chained together with PP pointers to a root, access to these segments is possible in reverse of the usual order.

In the preceding figure, you saw that you could cross from the ITEM to the ORDER database when virtual pairing was used, and this was done using logical child pointers. However, the logical child pointer only got you from ITEM to the logical child ORDITEM. The following figure shows how to get to ORDER. The PP pointer in ORDITEM points to its physical parent ORDER. If ORDER and ITEM are in an HD database but are not root segments, they (and all other segments in the path of the root) would also contain PP pointers to their physical parents.

PP pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. PP pointers are stored in a logical child or logical parent's prefix, along with any other pointers.
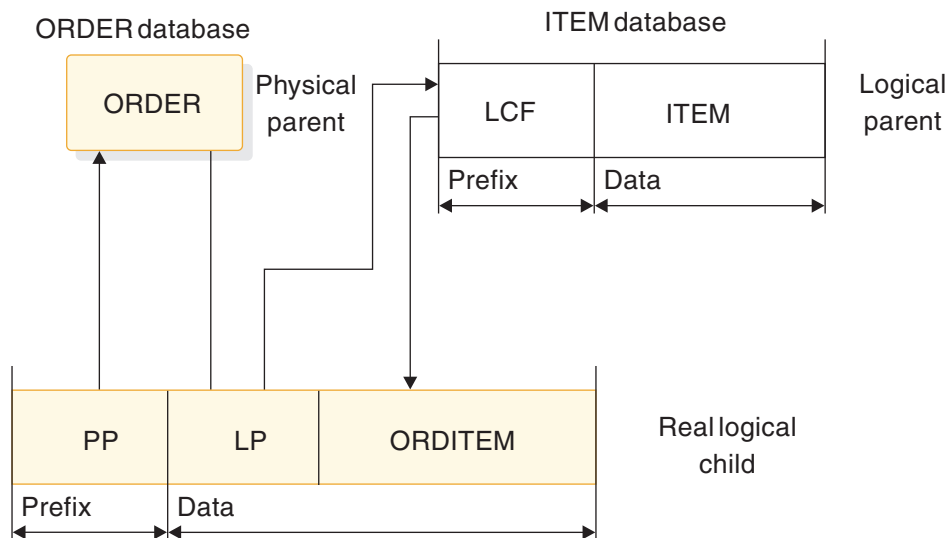
Figure 81. Physical parent (PP) pointer

In the preceding figure, the PP pointer points from the logical child ORDITEM to its physical parent ORDER. It is generated automatically by IMS for all logical child and logical parent segments in HD databases. In addition, it is in the prefix of the segment that contains it and consists of the 4-byte direct address of its physical parent. PP pointers are generated in all segments from the logical child or logical parent back up to the root.

Logical twin pointer

Logical twin pointers are used only in logical relationships with virtual pairing. Logical twins are multiple logical child segments that point to the same occurrence of a logical parent. Two types of logical twin pointers can be used:
- Logical twin forward (LTF) pointers, or
- The combination of logical twin forward (LTF) and logical twin backward (LTB) pointers

An LTF pointer points from a specific logical twin to the logical twin stored after it. An LTB pointer can only be specified in conjunction with an LTF pointer. When

specified, an LTB points from a given logical twin to the logical twin stored before it. Logical twin pointers work in a similar way to the physical twin pointers used in HD databases. As with physical twin backward pointers, LTB pointers improve performance on delete operations. They do this when the delete that causes DASD space release is a delete from the physical access path. Similarly, PTB pointers improve performance when the delete that causes DASD space release is a delete from the logical access path.

The following figure shows use of the LTF pointer. In this example, ORDER 123 has two items: bolt and washer. The ITEMORD segments beneath the two ITEM segments use LTF pointers. If the ORDER database is entered, it can be crossed to the ITEMORD segment for bolts in the ITEM database. Then, to retrieve all items for ORDER 123, the LTF pointers in the ITEMORD segment can be followed. In the following figure only one other ITEMORD segment exists, and it is for washers. The LTF pointer in this segment, because it is the last twin in the chain, contains zeros.

LTB pointers on dependent segments improve performance when deleting a real logical child in a virtually paired logical relationship. This improvement occurs when the delete is along the physical path.

LTF and LTB pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. This means LTF and LTB pointers can only exist in HD databases. The following figure shows a LTF pointer.
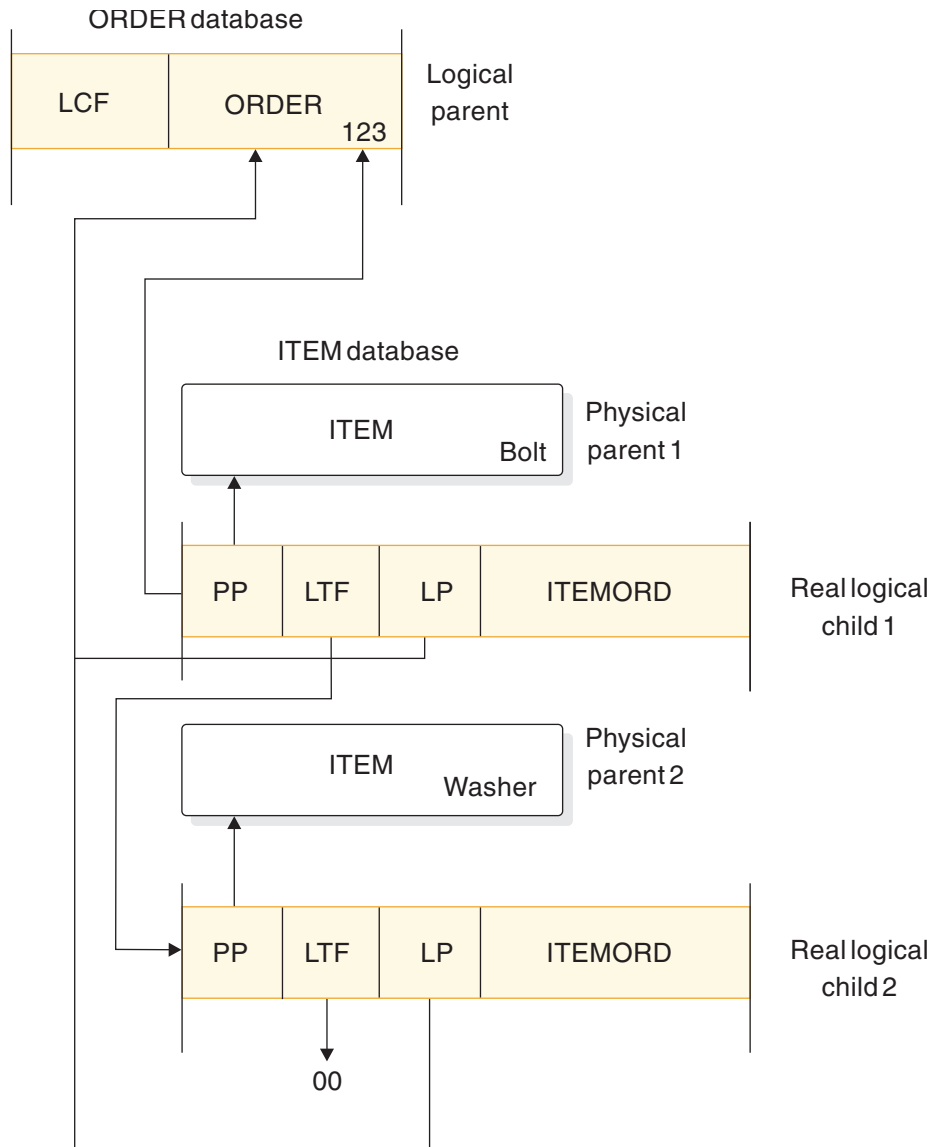
Figure 82. Logical twin forward (LTF) pointer (used in virtual pairing only)

In the preceding figure, the LTF pointer points from a specific logical twin to the logical twin stored after it. In this example, it points from the ITEMORD segment for bolts to the ITEMORD segment for washers. Because it is a direct pointer, the LTF pointer can only point to an HD database. The LTF pointer is in the prefix of a logical child segment and consists of the 4-byte RBA of the logical twin stored after it.

Indirect pointers

HALDB databases (PHDAM, PHIDAM, and PSINDEX databases) use direct and indirect pointers for pointing from one database record to another database record. The following figure shows how indirect pointers are used.
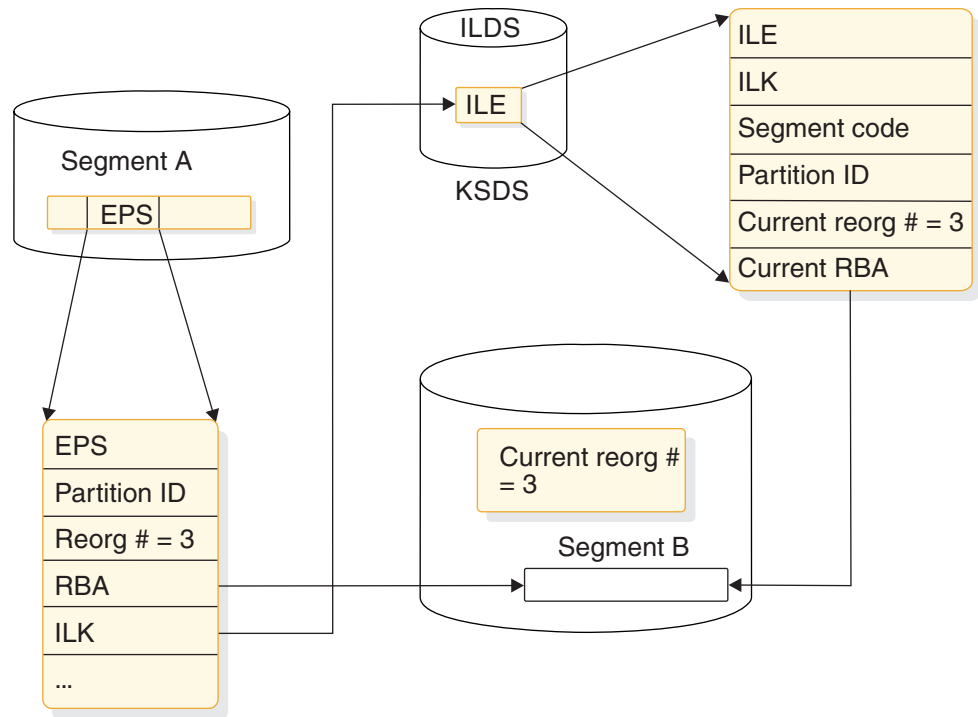
Figure 83. Self-healing pointers

The use of indirect pointers prevents the problem of misdirected pointers that would otherwise occur when a database is reorganized.

The repository for the indirect pointers is the indirect list data set. The misdirected pointers after reorganization are self-healing using indirect pointers.

Related concepts:

"The HALDB self-healing pointer process" on page 691

Paths in logical relationships

The relationship between physical parent and logical child in a physical database and the LP pointer in each logical child creates a physical parent to logical parent path.

To define use of the path, the logical child and logical parent are defined as a concatenated segment type that is a physical child of the physical parent, as shown in the following figure. Definition of the path and the concatenated segment type is done in what is called a logical database.
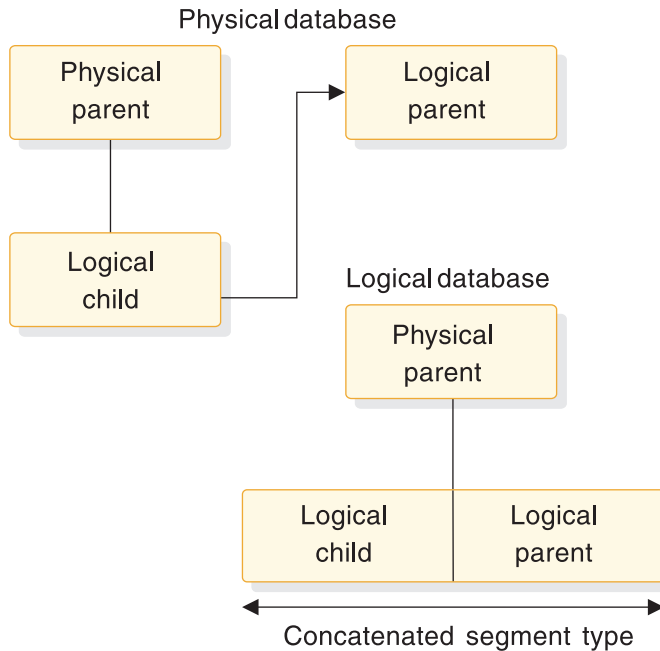
Figure 84. Defining a physical parent to logical parent path in a logical database

In addition, when LC pointers are used in the logical parent and logical twin and PP pointers are used in the logical child, a logical parent to physical parent path is created. To define use of the path, the logical child and physical parent are defined as one concatenated segment type that is a physical child of the logical parent, as shown in the following figure. Again, definition of the path is done in a logical database.
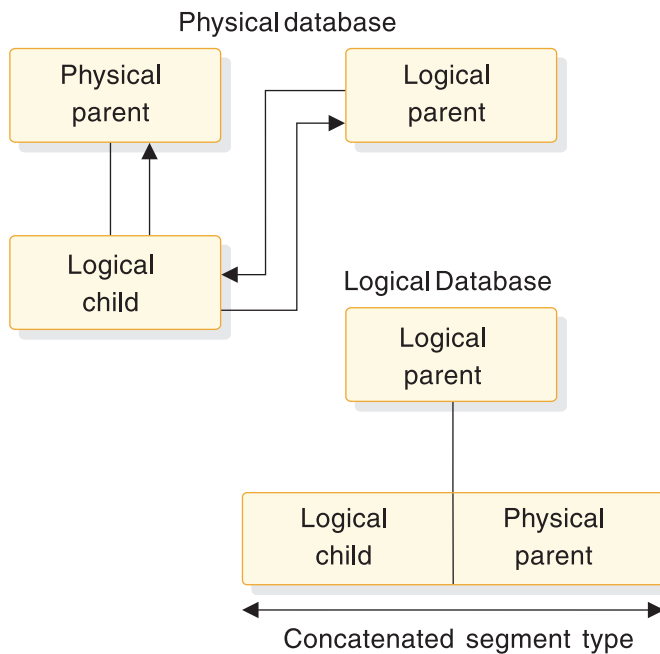
Figure 85. Defining a logical parent to physical parent path in a logical database

When use of a physical parent to logical parent path is defined, the physical parent is the parent of the concatenated segment type. When an application program

retrieves an occurrence of the concatenated segment type from a physical parent, the logical child and its logical parent are concatenated and presented to the application program as one segment. When use of a logical parent to physical parent path is defined, the logical parent is the parent of the concatenated segment type. When an application program retrieves an occurrence of the concatenated segment type from a logical parent, an occurrence of the logical child and its physical parent are concatenated and presented to the application program as one segment.

In both cases, the physical parent or logical parent segment included in the concatenated segment is called the *destination parent*. For a physical parent to logical parent path, the logical parent is the destination parent in the concatenated segment. For a logical parent to physical parent path, the physical parent is the destination parent in the concatenated segment.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

The logical child segment

When defining a logical child in its physical database, the length specified for it must be large enough to contain the concatenated key of the logical parent.

Any length greater than that required for the concatenated key of the logical parent can be used to store *intersection data*, a type of data that is unique to a particular logical relationship.

To identify which logical parent is pointed to by a logical child, the concatenated key of the logical parent must be present. Each logical child segment must be present in the application program's I/O area when the logical child is initially presented for loading into the database. However, if the logical parent is in an HD database, its concatenated key might not be written to storage when the logical child is loaded. If the logical parent is in a HISAM database, a logical child in storage must contain the concatenated key of its logical parent.

For logical child segments, you can define a special operand on the PARENT= parameter of the SEGM statement. This operand determines whether a symbolic pointer to the logical parent is stored as part of the logical child segment on the storage device. If PHYSICAL is specified, the concatenated key of the logical parent is stored with each logical child segment. If VIRTUAL is specified, only the intersection data portion of each logical child segment is stored.

When a concatenated segment is retrieved through a logical database, it contains the logical child segment, which consists of the concatenated key of the destination parent, followed by any intersection data. In turn, this is followed by data in the destination parent. The following figure shows the format of a retrieved concatenated segment in the I/O area. The concatenated key of the destination parent is returned with each concatenated segment to identify which destination parent was retrieved. IMS gets the concatenated key from the logical child in the concatenated segment or by constructing the concatenated key. If the destination parent is the logical parent and its concatenated key has not been stored with the logical child, IMS constructs the concatenated key and presents it to the application program. If the destination parent is the physical parent, IMS must always construct its concatenated key.
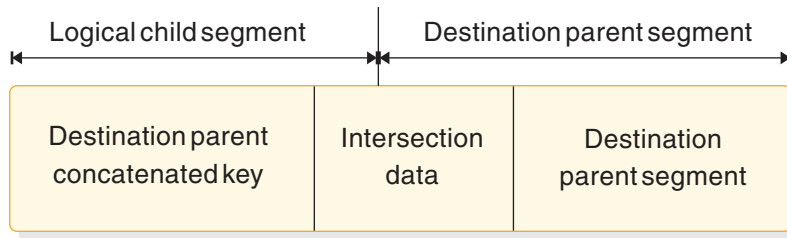
Figure 86. Format of a concatenated segment returned to user I/O area

Related concepts:
"Intersection data" on page 265
Related reference:
"LCHILD segment type format" on page 83

Segment prefix information for logical relationships

You should be aware of two things regarding the prefix of a segment involved in a logical relationship.

First, IMS places pointers in the prefix in a specific sequence and, second, IMS places a counter in the prefix for logical parents that do not have logical child pointers.

Sequence of pointers in a segment's prefix

When a segment contains more than one type of pointer and is involved in a logical relationship, pointers are put in the segment's prefix in the following sequence:
1. HF
2. HB
3. PP
4. LTF
5. LTB
6. LP

Or:
1. TF
2. TB
3. PP
4. LTF
5. LTB
6. LP
7. PCF
8. PCL

Or:
1. TF
2. TB
3. PP

4. PCF
5. PCL
6. EPS

Multiple PCF and PCL pointers can exist in a segment type; however, more than one of the other types of pointers can not.

Counter used in logical relationships

IMS puts a 4-byte counter in all logical parents that do not have logical child pointers. The counter is stored in the logical parent's prefix and contains a count of the number of logical children pointing to this logical parent. The counter is maintained by IMS and is used to handle delete operations properly. If the count is greater than zero, the logical parent cannot be deleted from the database because there are still logical children pointing to it.

Intersection data

When two segments are logically related, data can exist that is unique to only that relationship.

In the following figure, for example, one of the items ordered in ORDER 123 is 5000 bolts. The quantity 5000 is specific to this order (ORDER 123) and this item (bolts). It does not belong to either the order or item on its own. Similarly, in ORDER 123, 6000 washers are ordered. Again, this data is concerned only with that particular order and item combination.

This type of data is called intersection data, since it has meaning only for the specific logical relationship. The quantity of an item could not be stored in the ORDER 123 segment, because different quantities are ordered for each item in ORDER 123. Nor could it be stored in the ITEM segment, because for each item there can be several orders, each requesting a different quantity. Because the logical child segment links the ORDER and ITEM segments together, data that is unique to the relationship between the two segments can be stored in the logical child.

The two types of intersection data are: fixed intersection data and variable intersection data.

Fixed intersection data

Data stored in the logical child is called fixed intersection data. When symbolic pointing is used, it is stored in the data part of the logical child after the LPCK. When direct pointing is used, it is the only data in the logical child segment. Because symbolic pointing is used in the following figure, BOLT and WASHER are the LPCK, and the 5000 and 6000 are the fixed intersection data. The fixed intersection data can consist of several fields, all of them residing in the logical child segment.
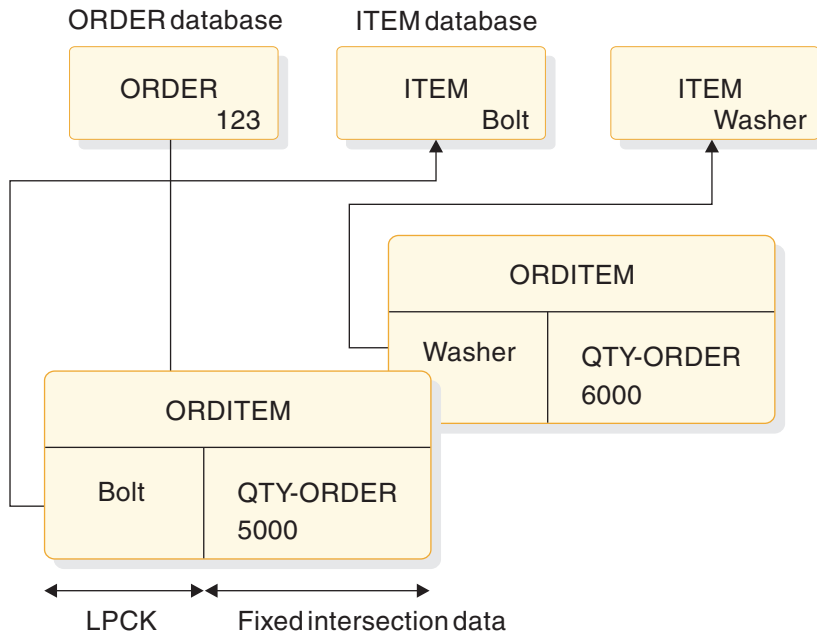
Figure 87. Fixed intersection data

Variable intersection data

Variable intersection data is used when you have data that is unique to a relationship, but several occurrences of it exist. For example, suppose you cannot supply in one shipment the total quantity of an item required for an order. You need to store delivery data showing the quantity delivered on a specified date. The delivery date is not dependent on either the order or item alone. It is dependent on a specific order-item combination. Therefore, it is stored as a dependent of the logical child segment. The data in this dependent of the logical child is called variable intersection data. For each logical child occurrence, there can be as many occurrences of dependent segments containing intersection data as you need.

The following figure shows variable intersection data. In the ORDER 123 segment for the item BOLT, 3000 were delivered on March 2 and 1000 were delivered on April 2. Because of this, two occurrences of the DELIVERY segment exist. Multiple segment types can contain intersection data for a single logical child segment. In addition to the DELIVERY segment shown in the figure, note the SCHEDULE segment type. This segment type shows the planned shipping date and the number of items to be shipped. Segment types containing variable intersection data can all exist at the same level in the hierarchy as shown in the figure, or they can be dependents of each other.
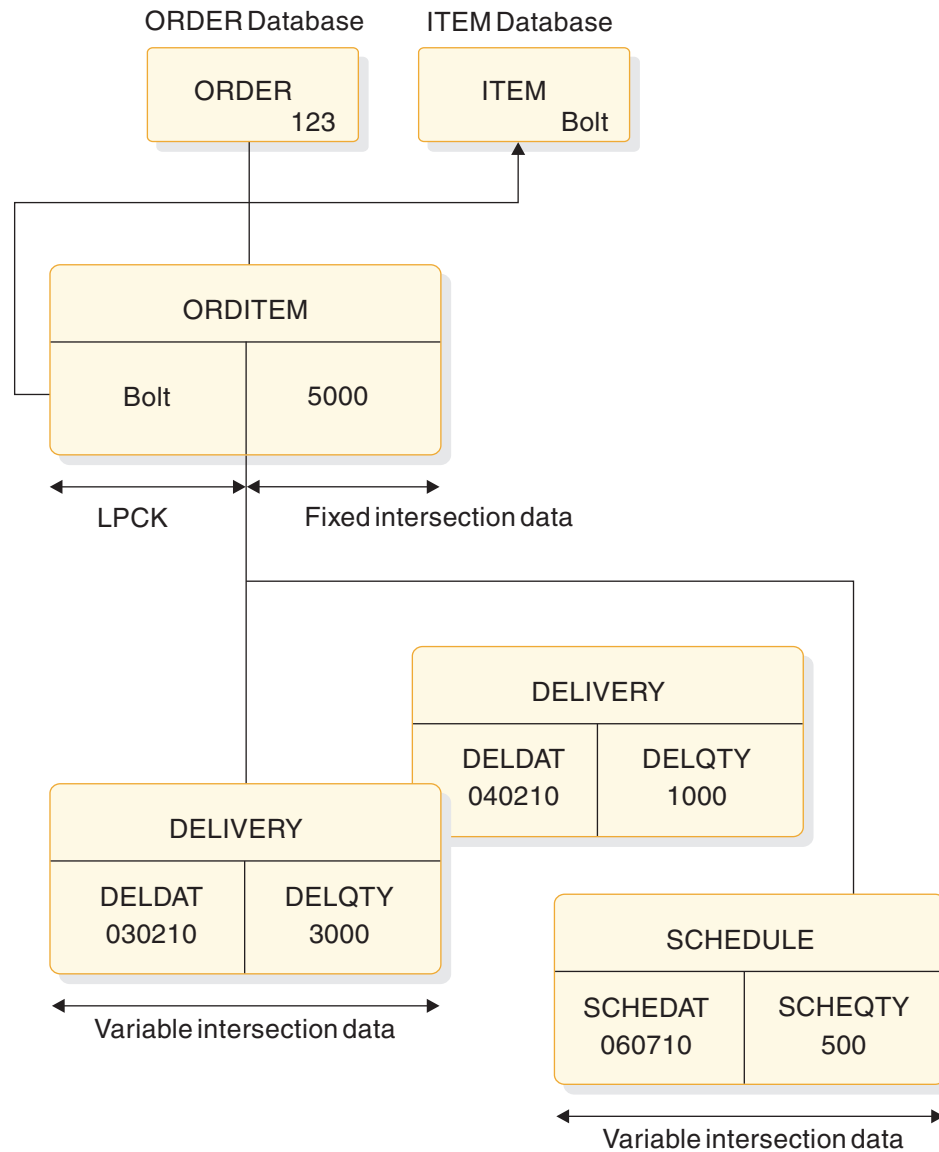
Figure 88. Variable intersection data

Fixed intersection data, variable intersection data, and physical pairing

In the previous figures, intersection data has been stored in a unidirectional logical relationship. It works exactly the same way in the two bidirectional logical relationships. However, when physical pairing is used, variable intersection data can only be stored on one side of the relationship. It does not matter on which side it is stored. An application program can access it using either the ORDER or ITEM database. Fixed intersection data must be stored on both sides of the relationship when physical pairing is used. IMS automatically maintains the fixed intersection data on both sides of the relationship when it is changed on one side. However, extra time is required for maintenance, and extra space is required on DASD for fixed intersection data in a physically paired relationship.

Related concepts:

"The logical child segment" on page 263

Recursive structures: same database logical relationships

Logical relationships can be established between segments in two or more physical databases. Logical relationships can also be established between segments in the same database. The logical data structure that results is called a *recursive structure*.

Most often, recursive structures are defined in manufacturing for bill-of-materials type applications. Suppose, for example, a company manufactures bicycles. The first model the manufacturer makes is Model 1, which is a boy's bicycle. The following table lists the parts needed to manufacture this bicycle and the number of each part needed to manufacture one Model 1 bicycle.

Table 57. Parts list for the Model 1 bicycle example

Part	Number needed
21-inch boy's frame	1
Handlebar	1
Seat	1
Chain	1
Front fender	1
Rear fender	1
Pedal	2
Crank	1
Front sprocket	1
26-inch tube and tire	2
26-inch rim	2
26-inch spoke	72
Front hub	1
Housing	1
Brake	1
Rear sprocket	1

In manufacturing, it is necessary to know the steps that must be executed to manufacture the end product. For each step, the parts needed must be available and any subassemblies used in a step must have been assembled in previous steps. The following figure shows the steps required to manufacture the Model 1 bicycle. A housing, brake, and rear sprocket are needed to make the rear hub assembly in step 2. Only then can the part of step 3 that involves building the rear wheel assembly be executed. This part of step 3 also requires availability of a 26-inch tire, a rim, and 36 spokes.
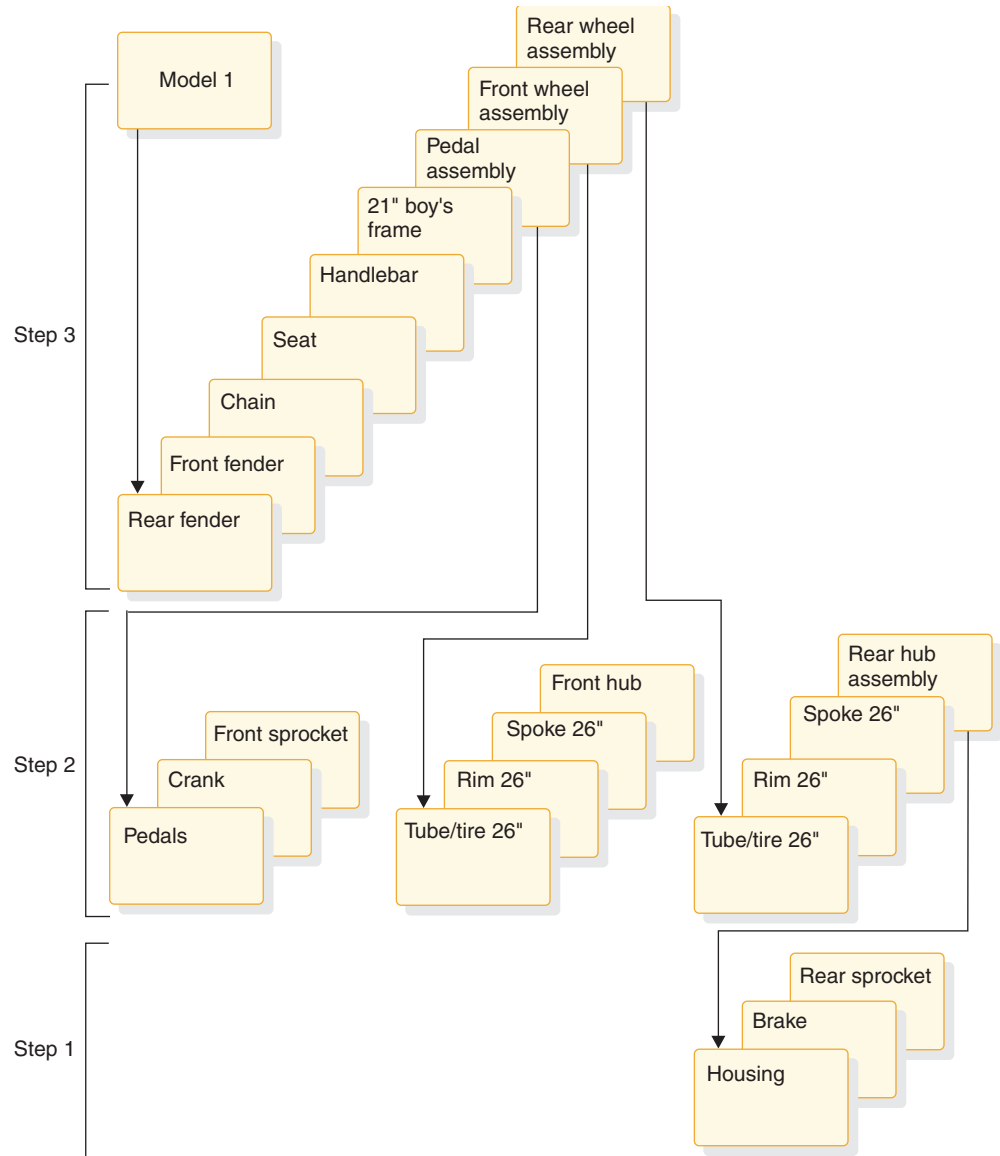
Figure 89. Model 1 components and subassemblies

The same company manufactures a Model 2 bicycle, which is for girls. The parts and assembly steps for this bicycle are exactly the same, except that the bicycle frame is a girl's frame.

If the manufacturer stored all parts and subassemblies for both models as separate segments in the database, a great deal of duplicate data would exist. The preceding figure shows the segments that must be stored just for the Model 1 bicycle. A similar set of segments must be stored for the Model 2 bicycle, except that it has a girl's bicycle frame. As you can see, this leads to duplicate data and the associated maintenance problems. The solution to this problem is to create a recursive structure. The following figure shows how this is done using the data for the Model 1 bicycle.
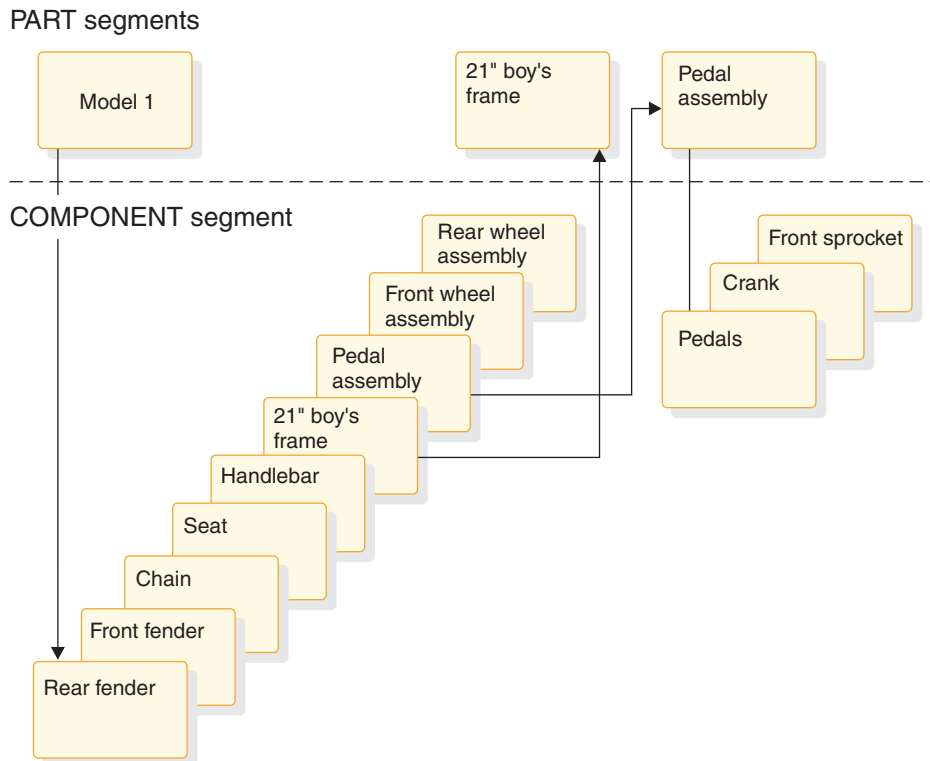
PART segments

Figure 90. Database records for the Model 1 bicycle

In the above figure, two types of segments exist: PART and COMPONENT segments. A unidirectional logical relationship has been established between them. The PART segment for Model 1 is a root segment. Beneath it are nine occurrences of COMPONENT segments. Each of these is a logical child that points to another PART root segment. (Only two of the pointers are actually shown to keep the figure simple.) However, the other PART root segments show the parts required to build the component.

For example, the pedal assembly component points to the PART root segment for assembling the pedal. Stored beneath this segment are the following parts that must be assembled: one front sprocket, one crank, and two pedals. With this structure, much of the duplicate data otherwise stored for the Model 2 bicycle can be eliminated.

The following figure shows the segments, in addition to those in the preceding figure, that must be stored in the database record for the Model 2 bicycle. The logical children in the figure, except the one for the unique component, a 21" girl's frame, can point to the same PART segments as are shown in the preceding figure. A separate PART segment for the pedal assembly, for example, need not exist. The database record for both Model 1 and 2 have the same pedal assembly, and by using the logical child, it can point to the same PART segment for the pedal assembly.

PART segments
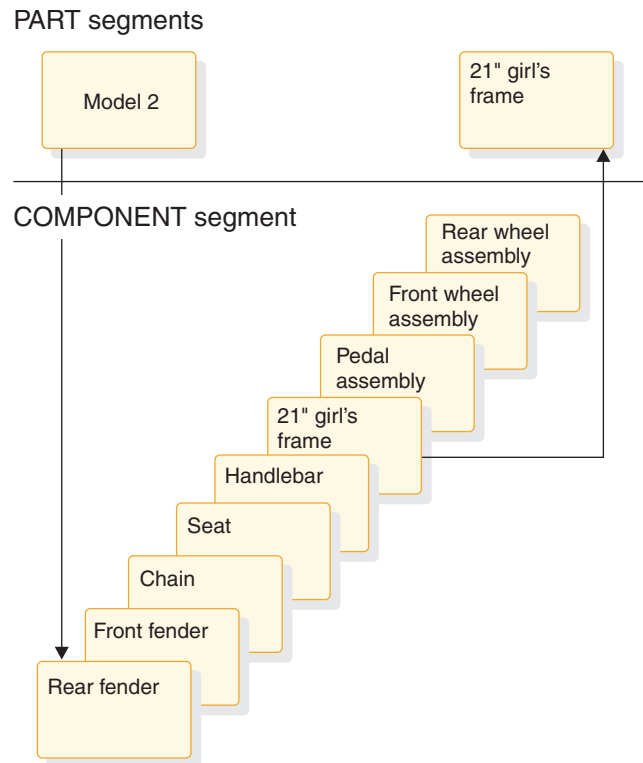
Figure 91. Extra database records required for the Model 2 bicycle

One thing to note about recursive structures is that the physical parent and the logical parent of the logical child are the same segment type. For example, in Figure 90 on page 270, the PART segment for *Model 1* is the physical parent of the COMPONENT segment for pedal assembly. The PART segment for pedal assembly is the logical parent of the COMPONENT segment for pedal assembly.

Related concepts:

"Secondary indexes versus logical relationships" on page 246

Defining sequence fields for logical relationships

When you use logical relationships, certain rules and recommendations should be followed when defining sequence fields.

Logical parent sequence fields

To avoid potential problems in processing databases using logical relationships, unique sequence fields should be defined in all logical parent segments and in all segments a logical parent is dependent on in its physical database. When unique sequence fields are not defined in all segments on the path to and including a logical parent, multiple logical parents in a database can have the same concatenated key. When this happens, problems can arise during and after initial database load when symbolic logical parent pointers in logical child segments are used to establish position on a logical parent segment.

At initial database load time, if logical parents with non-unique concatenated keys exist in a database, the resolution utilities attach all logical children with the same concatenated key to the first logical parent in the database with that concatenated key.

When inserting or deleting a concatenated segment and position for the logical parent, part of the concatenated segment is determined by the logical parent's concatenated key. Positioning for the logical parent starts at the root and stops on the first segment at each level of the logical parent's database that satisfies the key equal condition for that level. If a segment is missing on the path to the logical parent being inserted, a GE status code is returned to the application program when using this method to establish position in the logical parent's database.

Real logical children sequence fields

If the sequence field of a real logical child consists of any part of the logical parent's concatenated key, PHYSICAL must be specified on the PARENT= parameter in the SEGM statement for the logical child. This will cause the concatenated key of the logical parent to be stored with the logical child segment.

Virtual logical children sequence fields

As a general rule, a segment can have only one sequence field. However, in the case of virtual pairing, multiple FIELD statements can be used to define a logical sequence field for the virtual logical child.

A sequence field must be specified for a virtual logical child if, when accessing it from its logical parent, you need real logical child segments retrieved in an order determined by data in a field of the virtual logical child as it could be seen in the application program I/O area. This sequence field can include any part of the segment as it appears when viewed from the logical parent (that is, the concatenated key of the real logical child's physical parent followed by any intersection data). Because it can be necessary to describe the sequence field of a logical child as accessed from its logical parent in non-contiguous pieces, multiple FIELD statements with the SEQ parameter present are permitted. Each statement must contain a unique fldname1 parameter.

Related concepts:

"Altering IMS logical relationships" on page 510

Related reference:

➡ Database Prefix Resolution utility (DFSURG10) (Database Utilities)

➡ Database Prefix Update utility (DFSURGP0) (Database Utilities)

➡ Database Prereorganization utility (DFSURPR0) (Database Utilities)

PSBs, PCBs, and DBDs in logical relationships

When a logical relationship is used, you must define the physical databases involved in the relationship to IMS by using a *physical* DBD. In addition, many times you must define the logical structure of the databases to IMS because this is the structure that the application program perceives. This is done using a *logical* DBD.

A logical DBD is needed because the application program's PCB references a DBD, and the physical DBD does not reflect the logical data structure the application program needs to access. Finally, the application program needs a PSB, consisting of one or more PCBs. The PCB that is used when processing with a logical relationship points to the logical DBD when one has been defined. This PCB

indicates which segments in the logical database the application program can process. It also indicates what type of processing the application program can perform on each segment.

Internally, the PSB and PCBs, the logical DBD, and the physical DBD are represented to IMS as control blocks. The following figure shows the relationship between these three control blocks. It assumes that the logical relationship is established between two physical databases.
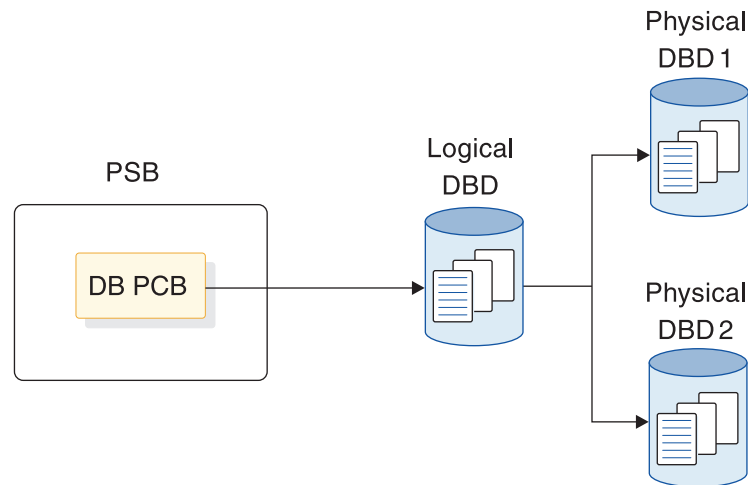
Figure 92. Relationship of control blocks when a logical relationship is used

Related tasks:

"Specifying logical relationships in the physical DBD"

"Specifying logical relationships in the logical DBD" on page 277

Specifying logical relationships in the physical DBD

For each of the databases involved in a logical relationship, you must code a physical DBD.

All statements in the physical DBD are coded with the same format used when a logical relationship is not defined, except for the SEGM and LCHILD statements. The SEGM statement, which describes a segment and its length and position in the database hierarchy, is expanded to include the new types of pointers. The LCHILD statement is added to define the logical relationship between the two segment types.

In the SEGM statements of the examples associated with the following figures, only the pointers required with logical relationships are shown. No pointers required for use with HD databases are shown. When actually coding a DBD, you must ask for these pointers in the PTR= parameter. Otherwise, IMS will not generate them once another type of pointer is specified.

The following figure shows the layout of segments.

ORDER

ORDKEY	DESCRIPTION OF LOCATION	ORDATE	*

Bytes 1 11 41 47 50

ORDITEM

ITEMNO	ORDITQTY

Bytes 1 9 17

DELIVERY

DELDAT	QUANTITY	DESCRIPTION	*

Bytes 1 7 15 45 50

SCHEDULE

SCHEDAT	QUANTITY	PLANNED SHIPPING DATE	INVENTORY PLACE	DESCRIPTION

Bytes 1 7 15 21 25 50

ITEM

ITEMKEY	DESCRIPTION	DATE OF CREATION	*

Bytes 1 9 49 55 60

Figure 93. Layouts of segments used in the examples

This is the hierarchical structure of the two databases involved in the logical relationship. In this example, we are defining a unidirectional relationship using symbolic pointing. ORDITEM has an LPCK and fixed intersection data, and DELIVERY and SCHEDULE are variable intersection data.

The following figure shows physical DBDs for unidirectional relationships.
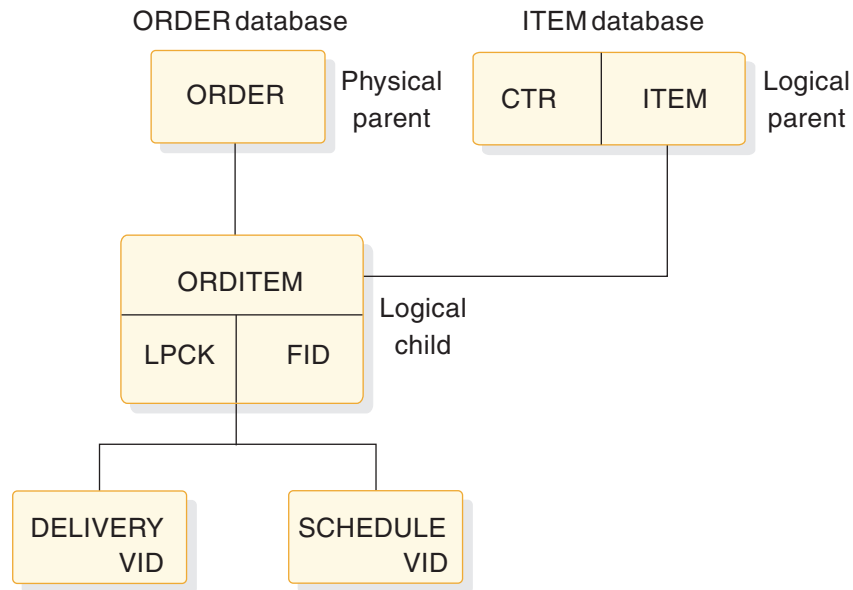
Figure 94. Physical DBDs for unidirectional relationship using symbolic pointing

The following DBD is for the ORDER database:

```
DBD     NAME=ORDDB
SEGM    NAME=ORDER,BYTES=50,FREQ=28000,PARENT=0
FIELD   NAME=(ORDKEY,SEQ),BYTES=10,START=1,TYPE=C
FIELD   NAME=ORDATE,BYTES=6,START=41,TYPE=C
SEGM    NAME=ORDITEM,BYTES=17,PARENT=((ORDER),(ITEM,P,ITEMDB))
FIELD   NAME=(ITEMNO,SEQ),BYTES=8,START=1,TYPE=C
FIELD   NAME=ORDITQTY,BYTES=9,START=9,TYPE=C
SEGM    NAME=DELIVERY,BYTES=50,PARENT=ORDITEM
FIELD   NAME=(DELDAT,SEQ),BYTES=6,START=1,TYPE=C
SEGM    NAME=SCHEDULE,BYTES=50,PARENT=ORDITEM
FIELD   NAME=(SCHEDAT,SEQ),BYTES=6,START=1,TYPE=C
DBDGEN
FINISH
END
```

The following DBD is for the ITEM database:

```
DBD     NAME=ITEMDB
SEGM    NAME=ITEM,BYTES=60,FREQ=50000,PARENT=0
FIELD   NAME=(ITEMKEY,SEQ),BYTES=8,START=1,TYPE=C
LCHILD  NAME=(ORDITEM,ORDDB)
DBDGEN
FINISH
END
```

In the ORDER database, the DBD coding that differs from normal DBD coding is that for the logical child ORDITEM.

In the SEGM statement for ORDITEM:

1. The BYTES= parameter is 17. The length specified is the length of the LPCK, plus the length of the fixed intersection data. The LPCK is the key of the ITEM segment, which is 8 bytes long. The length of the fixed intersection data is 9 bytes.
2. The PARENT= parameter has two parents specified. Two parents are specified because ORDITEM is a logical child and therefore has both a physical and logical parent. The physical parent is ORDER. The logical parent is ITEM, specified after ORDER. Because ITEM exists in a different physical database

from ORDITEM, the name of its physical database, ITEMDB, must be specified. Between the segment name ITEM and the database name ITEMDB is the letter P. The letter P stands for physical. The letter P specifies that the LPCK is to be stored on DASD as part of the logical child segment.

In the FIELD statements for ORDITEM:

1. ITEMNO is the sequence field of the ORDITEM segment and is 8 bytes long. ITEMNO is the LPCK. The logical parent is ITEM, and if you look at the FIELD statement for ITEM in the ITEM database, you will see ITEM's sequence field is ITEMKEY, which is 8 bytes long. Because ITEM is a root segment, the LPCK is 8 bytes long.

2. ORDITQTY is the fixed intersection data and is coded normally.

In the ITEM database, the DBD coding that differs from normal DBD coding is that an LCHILD statement has been added. This statement names the logical child ORDITEM. Because the ORDITEM segment exists in a different physical database from ITEM, the name of its physical database, ORDDB, must be specified.

Related concepts:

"PSBs, PCBs, and DBDs in logical relationships" on page 272

Specifying bidirectional logical relationships

When defining a bidirectional relationship with physical pairing, you need to include an LCHILD statement under both logical parents and, in addition to other pointers, the PAIRED operand on the POINTER= parameter of the SEGM statements for both logical children.

When defining a bidirectional relationship with virtual pairing, you need to code an LCHILD statement only for the real logical child. On the LCHILD statement, you code POINTER=SNGL or DBLE to get logical child pointers. You code the PAIR= operand to indicate the virtual logical child that is paired with the real logical child. When you define the SEGM statement for the real logical child, the PARENT= parameter identifies both the physical and logical parents. You should specify logical twin pointers (in addition to any other pointers) on the POINTER= parameter. Also, you should define a SEGM statement for the virtual logical child even though it does not exist. On this SEGM statement, you specify PAIRED on the POINTER= parameter. In addition, you specify a SOURCE= parameter. On the SOURCE= parameter, you specify the SEGM name and DBD name of the real logical child. DATA must always be specified when defining SOURCE= on a virtual logical child SEGM statement.

Related reference:

➡ DBD statements (System Utilities)

Checklist of rules for defining logical relationships in physical databases

You must follow certain rules when defining logical relationships in physical databases.

In the following subtopics, all references are to segment types, not occurrences.

Logical child rules

Several rules govern the definition of the logical child segment type in a physical database.

- A logical child must have a physical and a logical parent.

- A logical child can have only one physical and one logical parent.
- A logical child is defined as a physical child in the physical database of its physical parent.
- A logical child is always a dependent segment in a physical database, and can, therefore, be defined at any level except the first level of a database.
- A logical child in its physical database cannot have a physical child defined at the next lower level in the database that is also a logical child.
- A logical child can have a physical child. However, if a logical child is physically paired with another logical child, only one of the paired segments can have physical children.

Logical parent rules

Several rules govern the definition of the logical parent segment type in a physical database.

- A logical parent can be defined at any level in a physical database, including the root level.
- A logical parent can have one or more logical children. Each logical child related to the same logical parent defines a logical relationship.
- A segment in a physical database cannot be defined as both a logical parent and a logical child.
- A logical parent can be defined in the same physical database as its logical child, or in a different physical database.

Physical parent rules

The only rule for defining a physical parent segment type in a physical database is that a physical parent of a logical child cannot also be a logical child.

Specifying logical relationships in the logical DBD

To identify which segment types are used in a logical data structure, you must code a logical DBD.

The following figure shows an example of how the logical DBD is coded. The example is a logical DBD for the same physical databases defined in "Specifying logical relationships in the physical DBD" on page 273.

When defining a segment in a logical database, you can specify whether the segment is returned to the program's I/O area by using the KEY or DATA operand on the SOURCE= parameter of the SEGM statement. DATA returns both the key and data portions of the segment to the I/O area. KEY returns only the key portion, and not the data portion of the segment to the I/O area.

When the SOURCE= parameter is used on the SEGM statement of a concatenated segment, the KEY and DATA parameters control which of the two segments, or both, is put in the I/O area on retrieval calls. In other words, you define the SOURCE= parameter twice for a concatenated segment type, once for the logical child portion and once for the destination parent portion.

The following figure illustrates the logical data structure you need to create in the application program. It is implemented with a unidirectional logical relationship using symbolic pointing. The root segment is ORDER from the ORDER database. Dependent on ORDER is ORDITEM, the logical child, concatenated with its logical parent ITEM. The application program receives both segments in its I/O area when a single call is issued for ORDIT. DELIVERY and SCHEDULE are variable

intersection data.
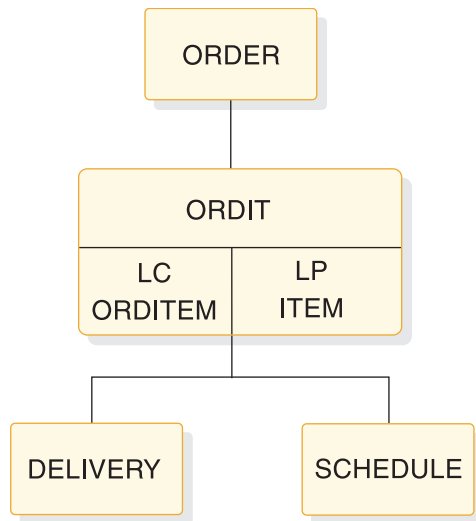
Figure 95. Logical data structure for a unidirectional relationship using symbolic pointing

The following logical DBD is for the logical data structure shown in the above figure:

```
DBD  NAME=ORDLOG,ACCESS=LOGICAL
DATASET LOGICAL
SEGM NAME=ORDER,SOURCE=((ORDER,DATA,ORDDB))
SEGM NAME=ORDIT,PARENT=ORDER,                                         X
          SOURCE=((ORDITEM,DATA,ORDDB),(ITEM,DATA,ITEMDB))
SEGM NAME=DELIVERY,PARENT=ORDIT,SOURCE=((DELIVERY,DATA,ORDDB))
SEGM NAME=SCHEDULE,PARENT=ORDIT,SOURCE=((SCHEDULE,DATA,ORDDB))
DBDGEN
FINISH
END
```

Notes to the preceding figure:

1. The DBD statement has the name of the logical DBD, in this example
   ORDLOG. As with physical DBDs, this name must be unique and must be the
   same name as specified in the MBR operand of the DBDGEN procedure.
   ACCESS=LOGICAL simply says this is a logical DBD.

2. The DATASET statement always says LOGICAL, meaning a logical DBD. No
   other parameters can be specified on this statement; however, ddnames for data
   sets are all specified in the DATASET statements in the physical DBDs.

3. The SEGM statements show which segments are to be included in the logical
   database. The only operands allowed on the SEGM statements for a logical
   DBD are NAME=, PARENT=, and SOURCE=. All other information about the
   segment is defined in the physical DBD.

   • The first SEGM statement defines the root segment ORDER.

     The NAME= operand specifies the name used in the PCB to refer to this
     segment. This name is used by application programmers when coding SSAs.
     In this example, the segment name is the same as the name used in the
     physical DBD - ORDER. However, the segment could have a different name
     from that specified in its physical DBD.

     The SOURCE= operand tells IMS where the data for this segment is to come
     from. First the name of the segment type appears in its physical database,
     which is ORDER. DATA says that the data in this segment needs to be put in

the application program's I/O area. ORDDB is the name of the physical database in which the ORDER segment exists.

No FIELD statements are coded in the logical DBD. IMS picks the statements up from the physical DBD, so when accessing the ORDER segment in this logical data structure, the application program could have SSAs referring to ORDKEY or ORDATE. These fields were defined for the ORDER segments in its physical DBD, as shown in Figure 94 on page 275.

- The second SEGM statement is for the ORDIT segment. The ORDIT segment is made up of the logical child ORDITEM, concatenated with its logical parent ITEM. As you can see, the SOURCE= operand identifies both the ORDITEM and ITEM segments in their different physical databases.
- The third and fourth SEGM statements are for the variable intersection data DELIVERY and SCHEDULE. These SEGM statements must be placed in the logical DBD in the same relative order they appear in the physical DBD. In the physical DBD, DELIVERY is to the left of SCHEDULE.

Related concepts:

"PSBs, PCBs, and DBDs in logical relationships" on page 272

"Paths in logical relationships" on page 261

Checklist of rules for defining logical databases

Logical relationships can become very complex. To help you to you properly define databases that use logical relationships, you must understand and follow the rules that govern logical relationships.

Before the rules for defining logical databases can be described, you need to know the following definitions:

- Crossing a logical relationship
- The first and additional logical relationships crossed

Also, a logical DBD is needed only when an application program needs access to a concatenated segment or needs to cross a logical relationship.

Definition of crossing a logical relationship

A logical relationship is considered crossed when it is used in a logical database to access a segment that is:

- A physical parent of a destination parent in the destination parent's database
- A physical dependent of a destination parent in the destination parent's physical database

If a logical relationship is used in a logical database to access a destination parent only, the logical relationship is not considered crossed.

In the following figure, DBD1 and DBD2 are two physical databases with a logical relationship defined between them. DBD3 through DBD6 are four logical databases that can be defined from the logical relationship between DBD1 and DBD2. With DBD3, no logical relationship is crossed, because no physical parent or physical dependent of a destination parent is included in DBD3. With DBD4 through DBD6, a logical relationship is crossed in each case, because each contains a physical parent or physical dependent of the destination parent.

Physical databases
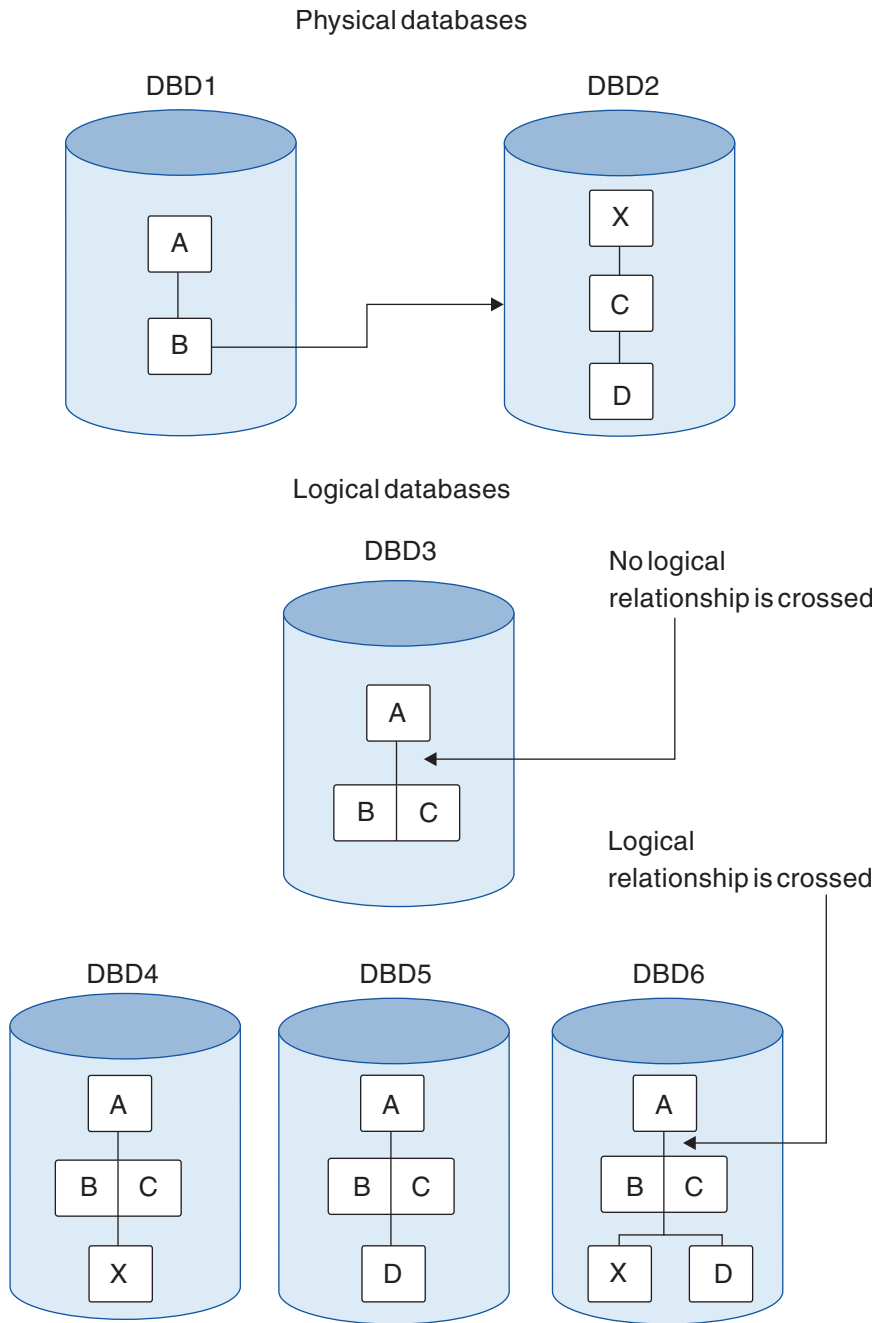
Logical databases

Figure 96. Definition of crossing a logical relationship

Definition of first and additional logical relationships crossed

More than one logical relationship can be crossed in a hierarchical path in a logical database. The following figure shows three physical databases (DBD1, DBD2 and DBD3) in which logical relationships have been defined. Also in the figure are two (of many) logical databases (DBD4 and DBD5) that can be defined from the logical relationships in the physical databases. In DBD4, the two concatenated segments BF and DI allow access to all segments in the hierarchical paths of their destination parents. If either logical relationship or both is crossed, each is considered the first logical relationship crossed. This is because each concatenated segment type is reached by following the physical hierarchy of segment types in DBD1.
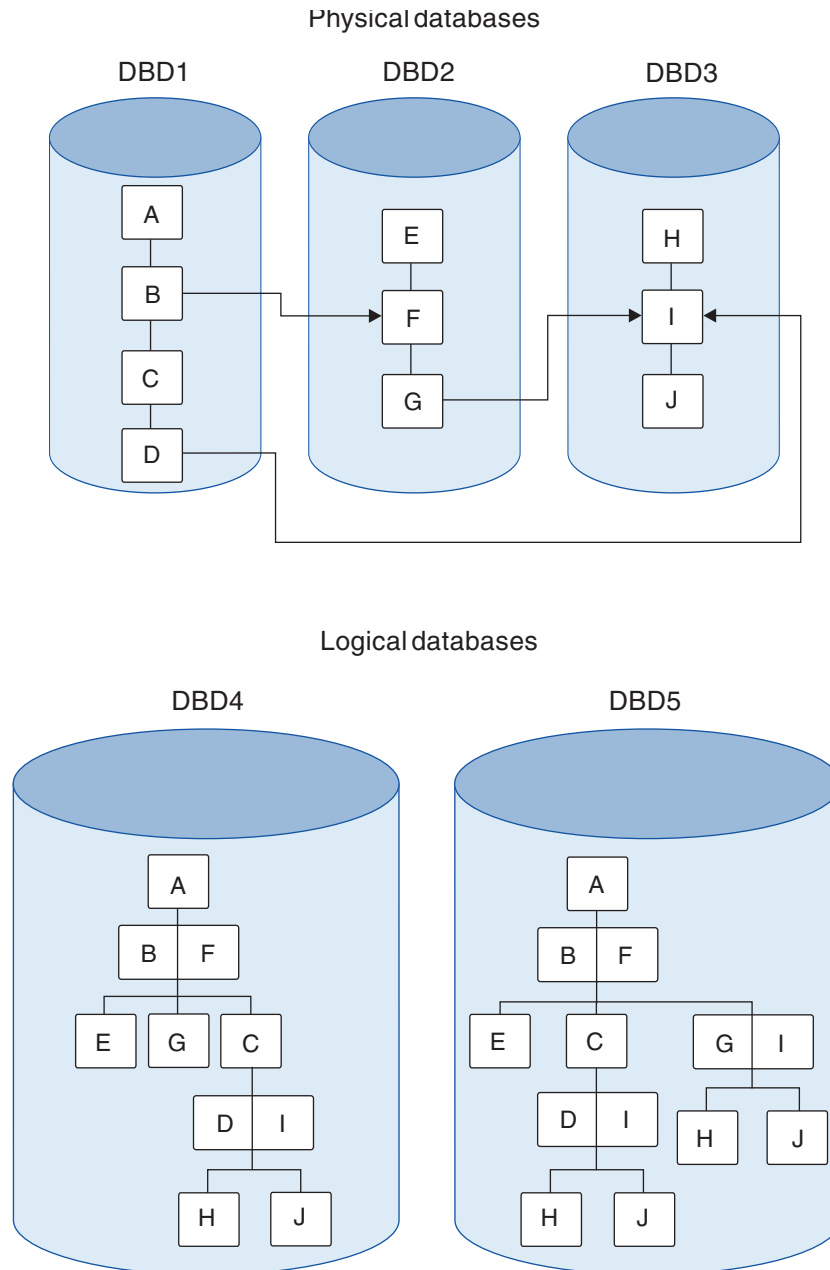
Figure 97. The first logical relationship crossed in a hierarchical path of a logical database

In DBD5 in the preceding figure, an additional concatenated segment type GI, is defined that was not included in DBD4. GI allows access to segments in the hierarchical path of the destination parent if crossed. When the logical relationship made possible by concatenated segment GI is crossed, this is an additional logical relationship crossed. This is because, from the root of the logical database, the logical relationship made possible by concatenated segment type BF must be crossed to allow access to concatenated segment GI.

When the first logical relationship is crossed in a hierarchical path of a logical database, access to all segments in the hierarchical path of the destination parent is made possible as follows:

- Parent segments of the destination parent are included in the logical database as dependents of the destination parent in reverse order, as shown in the following figure.
- Dependent segments of the destination parent are included in the logical database as dependents of the destination parent without their order changed, as shown in the following figure.

When an additional logical relationship is crossed in a logical database, access to all segments in the hierarchical path of the destination parent is made possible, just as in the first crossing.

Hierarchic path of a
physical database

```
        ┌─────────┐
        │    A    │
        └────┬────┘
        ┌────┴────┐
        │    B    │
        └────┬────┘
        ┌────┴────┐
        │    C    │
        └────┬────┘
    ┌──────────────┐
    │ Destination  │
    │   parent     │
    └──────┬───────┘
      ┌────┴──────┐
  ┌───┴───┐   ┌───┴───┐
  │   E   │   │   F   │
  └───────┘   └───────┘
```
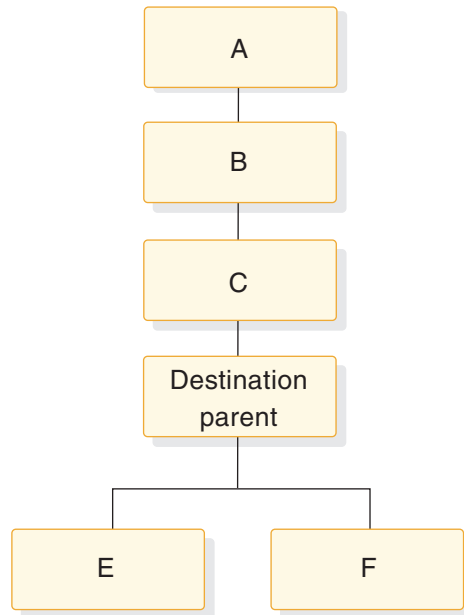
Resulting order in the
hierarchic path of a logical database

```
        ┌──────────────┐
        │ Destination  │
        │   parent     │
        └──────┬───────┘
     ┌─────────┼─────────┐
  ┌──┴──┐   ┌──┴──┐   ┌──┴──┐
  │  C  │   │  E  │   │  F  │
  └──┬──┘   └─────┘   └─────┘
  ┌──┴──┐
  │  B  │
  └──┬──┘
  ┌──┴──┐
  │  A  │
  └─────┘
```
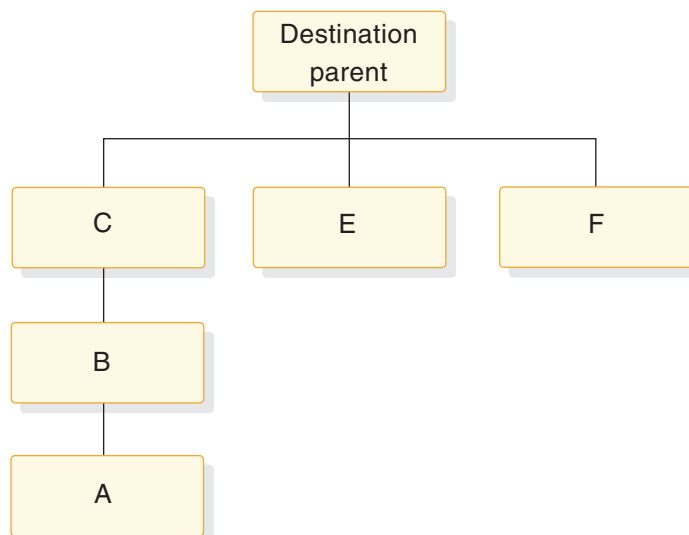
Figure 98. Logical database hierarchy enabled by crossing the first logical relationship

Rules for defining logical databases

- The root segment in a logical database must be the root segment in a physical database.
- A logical database must use only those segments and physical and logical relationship paths defined in the physical DBD referenced by the logical DBD.
- The path used to connect a parent and child in a logical database must be defined as a physical relationship path or a logical relationship path in the physical DBD referenced by the logical DBD.

- Physical and logical relationship paths can be mixed in a hierarchical segment path in a logical database.
- Additional physical relationship paths, logical relationship paths, or both paths can be included after a logical relationship is crossed in a hierarchical path in a logical database. These paths are included by going in upward directions, downward directions, or both directions, from the destination parent. When proceeding downward along a physical relationship path from the destination parent, direction cannot be changed except by crossing a logical relationship. When proceeding upward along a physical relationship path from the destination parent, direction can be changed.
- Dependents in a logical database must be in the same relative order as they are under their parent in the physical database. If a segment in a logical database is a concatenated segment, the physical children of the logical child and children of the destination parent can be in any order. The relative order of the children or the logical child and the relative order of the children of the destination parent must remain unchanged.
- The same concatenated segment type can be defined multiple times with different combinations of key and data sensitivity. Each must have a distinct name for that view of the concatenated segment. Only one of the views can have dependent segments. The following figure shows the four views of the same concatenated segment that can be defined in a logical database. A PCB for the logical database can be sensitive to only one of the views of the concatenated segment type.
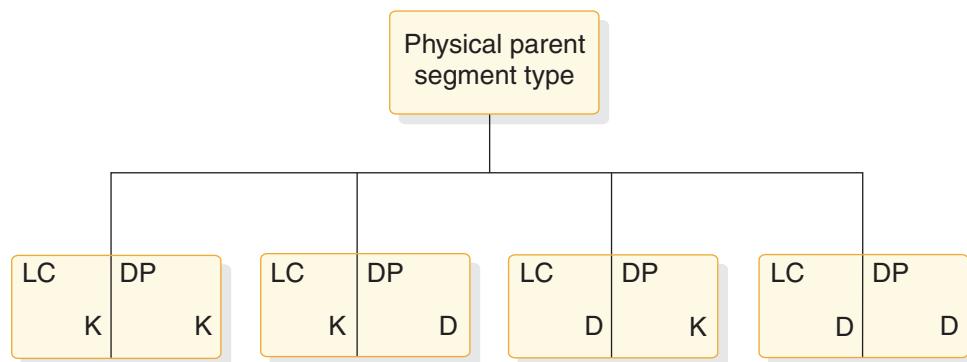
Figure 99. Single concatenated segment type defined multiple times with different combinations of key and data sensitivity

LC   Logical child segment type

DP   Destination parent segment type

K   KEY sensitivity specified for the segment type

D   DATA sensitivity specified for the segment type

Choosing replace, insert, and delete rules for logical relationships

You must establish insert, delete, and replace rules when a segment is involved in a logical relationship, because such segments can be updated from two paths: a physical path and a logical path.

The following figure and Figure 101 on page 286 show example insert, delete, and replace rules. Consider the following questions:

1. Should the CUSTOMER segment in the following figure be able to be inserted by both its physical and logical paths?
2. Should the BORROW segment be replaceable using only the physical path, or using both the physical and logical paths?
3. If the LOANS segment is deleted using its physical path, should it be erased from the database? Or should it be marked as physically deleted but remain accessible using its logical path?
4. If the logical child segment BORROW or the concatenated segment BORROW/LOANS is deleted from the physical path, should the logical path CUST/CUSTOMER also be automatically deleted? Or should the logical path remain?
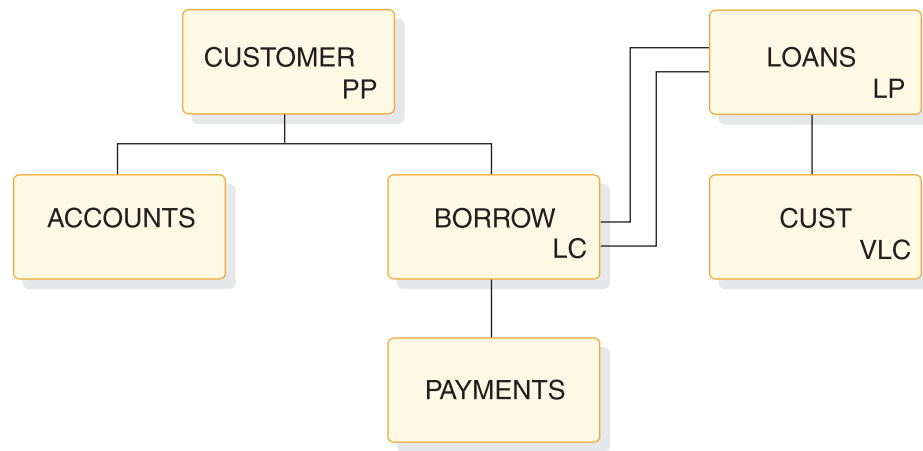
Figure 100. Example of the replace, insert, and delete rules

Abbreviation
    Explanation

PP   Physical parent segment type

LC   Logical child segment type

LP   Logical parent segment type

VLC   Virtual logical child segment type

Physical path to
CUSTOMER and
BORROW

Physical path
to LOANS

CUSTOMER		LOANS
BORROW/ LOANS		CUST/ CUSTOMER

Logical path to
LOANS

Logical path to
CUSTOMER and
BORROW
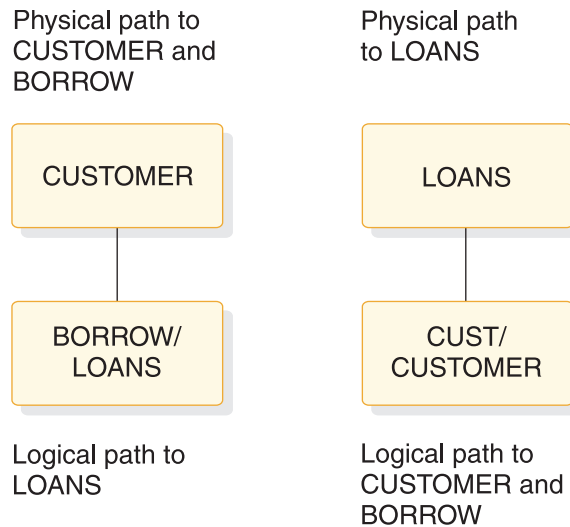
Figure 101. Example of the replace, insert, and delete rules: before and after

The answer to these questions depends on the application. The enforcement of the answer depends on your choosing the correct insert, delete, and replace rules for the logical child, logical parent, and physical parent segments. You must first determine your application processing requirements and then the rules that support those requirements.

For example, the answer to question 1 depends on whether the application requires that a CUSTOMER segment be inserted into the database before accepting the loan. An insert rule of physical (P) on the CUSTOMER segment prohibits insertion of the CUSTOMER segment except by the physical path. An insert rule of virtual (V) allows insertion of the CUSTOMER segment by either the physical or logical path. It probably makes sense for a customer to be checked (past credit, time on current job, and so on.) and the CUSTOMER segment inserted before approving the loan and inserting the BORROW segment. Thus, the insert rule for the CUSTOMER segment should be P to prevent the segment from being inserted logically. (Using the insert rule in this example provides better control of the application.)

Or consider question 3. If it is possible for this loan institution to cancel a type of loan (cancel 10% car loans, for instance, and create 12% car loans) before everyone with a 10% loan has fully paid it, then it is possible for the LOANS segment to be physically deleted and still be accessible from the logical path. This can be done by specifying the delete rule for LOANS as either logical (L) or V, but not as P.

The P delete rule prohibits physically deleting a logical parent segment before all its logical children have been physically deleted. This means the logical path to the logical parent is deleted first.

You need to examine all your application requirements and decide who can insert, delete, and replace segments involved in logical relationships and how those updates should be made (physical path only, or physical and logical path). The insert, delete, and replace rules in the physical DBD and the PROCOPT= parameter in the PCB are the means of control.

Related concepts:

"Insert, delete, and replace rules for logical relationships" on page 287

Insert, delete, and replace rules for logical relationships

You need to examine all your application requirements and decide who can insert, delete, and replace segments involved in logical relationships and how those updates are to be made (physical path only or physical and logical path).

The insert, delete, and replace rules in the physical DBD determine how updates apply across logical relationships.

This topic contains General-use Programming Interface information.
Related concepts:
"Choosing replace, insert, and delete rules for logical relationships" on page 284
"Utilization of available real storage" on page 718
Related reference:
"Bits in the prefix descriptor byte" on page 326

Specifying rules in the physical DBD

Insert, delete, and replace rules are specified using the RULES= keyword of a SEGM statement in the DBD for logical relationships.

The following figure contains a diagram of the RULES= keyword and its parameters.
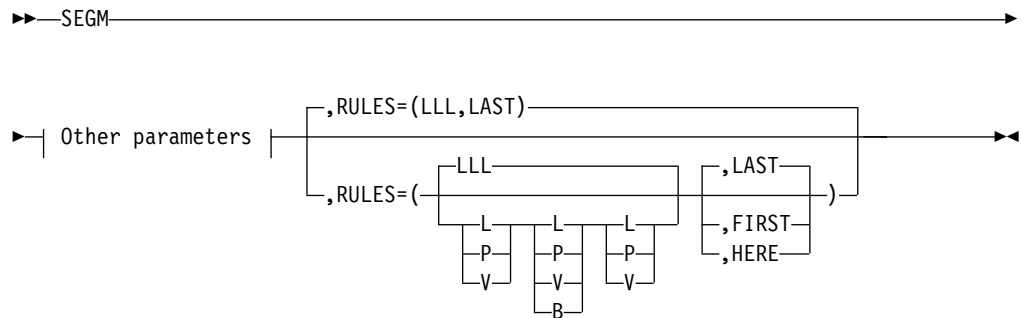
Figure 102. Insert, delete, and replace rules in the DBD

The valid parameter values for the insert replace and delete rules are:

B Specifies a bidirectional virtual delete rule. It is not a valid value for either the first or last positional parameter of the RULES= keyword.

L Specifies a logical insert, delete, or replace rule.

P Specifies a physical insert, delete, or replace rule.

V Specifies a virtual insert, delete, or replace rule.

The first three values that the RULES= keyword accepts are positional parameters:
- The first positional parameter sets the insert rule
- The second positional parameter sets the delete rule
- The third positional parameter sets the replace rule

For example, RULES=(PLV) says the insert rule is physical, the delete rule is logical, and the replace rule is virtual. The B rule is only applicable for delete. In general, the P rule is the most restrictive, the V rule is least restrictive, and the L rule is somewhere in between.

The RULES= parameter is applicable only to segments involved in logical paths, that is, the logical child, logical parent, and physical parent segments. The RULES= parameter is not coded for the virtual logical child.

Insert rules

The insert rules apply to the destination parent segments, but not to the logical child segment.

A destination parent can be a logical or physical parent. The insert rule has no meaning for the logical child segment except to satisfy the RULES= macro's coding scheme. Therefore, any insert rule (P, L, V) can be coded for a logical child. A logical child can be inserted provided:

- The insert rule of the destination parent is not violated
- The logical child being inserted does not already exist (it cannot be a duplicate)

A description of how the insert rules work for the destination parent is a follows:

- When RULES=P is specified, the destination parent can be inserted *only* using the physical path. This means the destination parent must exist before inserting a logical path. A concatenated segment is not needed, and the logical child is inserted by itself.
- When RULES=L is specified, the destination parent can be inserted either using the physical path or concatenated with the logical child and using the logical path. When a logical child/destination parent concatenated segment is inserted, the destination parent is inserted if it does not already exist and the I/O area key check does not fail. If the destination parent does exist, it will remain unchanged and the logical child will be connected to it.
- When RULES=V is specified, the destination parent can be inserted either using the physical path or concatenated with the logical child and using the logical path. When a logical child/destination parent concatenated segment is inserted, the destination parent is replaced if it already exists. If it does not already exist, the destination parent is inserted.

Related concepts:

"Status codes that can be issued after an ISRT call" on page 289

The logical child insert call

To insert the logical child segment, the I/O area in an application program must contain either the logical child or the logical child/destination parent concatenated segment in accordance with the destination parent's insert rule

For all DL/I calls, either an error is detected and an error status code returned (in which case no data is changed), or the required changes are made to all segments effected by the call. Therefore, if the required function cannot be performed for both parts of the concatenated segment, an error status code is returned, and no change is made to either the logical child or the destination parent.

The insert operation is not affected by KEY or DATA sensitivity as specified in a logical DBD or a PCB. This means that if a program is other than DATA sensitive to both the logical child and destination parent of a concatenated segment, and if the insert rules is L or V, the program must still supply both of them in the I/O

area when inserting using a logical path. Because of this, maintenance programs that insert concatenated segments should be DATA sensitive to both segments in the concatenation.

Status codes that can be issued after an ISRT call

The nonblank status codes that can be returned to an application program after an ISRT call are as follows.

- AM—An insert was attempted and the value of PROCOPT is not "I"
- GE—The parent of the destination parent or logical child was not found
- II—An attempt was made to insert a duplicate segment
- IX—The rule specified was P, but the destination parent was not found

  One reason for getting an IX status code is that the I/O area key check failed. Concatenated segments must contain the destination parent's key twice—once as part of the logical child's LPCK and once as a field in the parent. These keys must be equal.

The following two figures show a physical insert rule example.
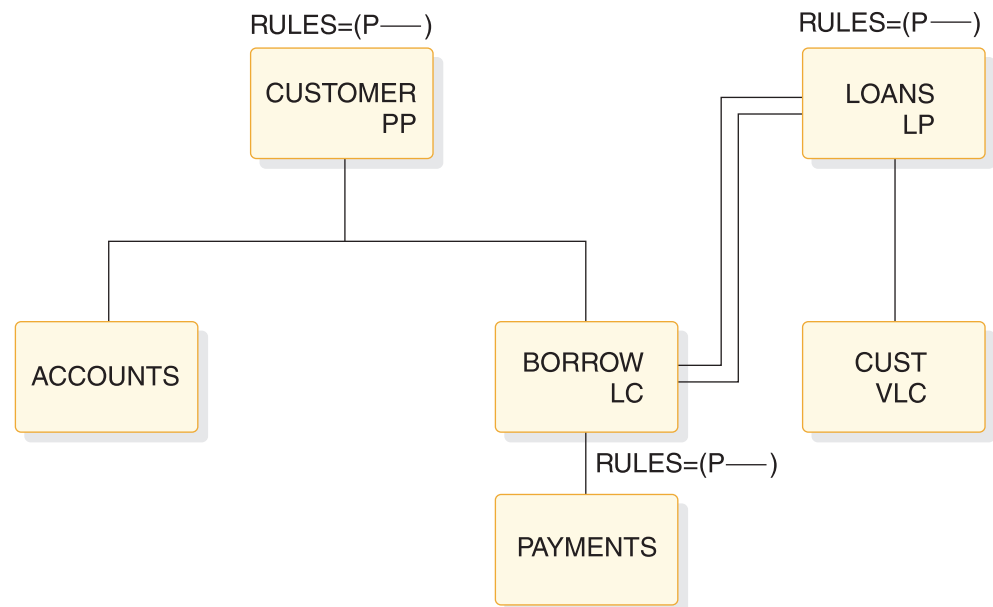
Figure 103. Physical insert rule example

Physical path to
CUSTOMER and BORROW

Physical path to
LOANS
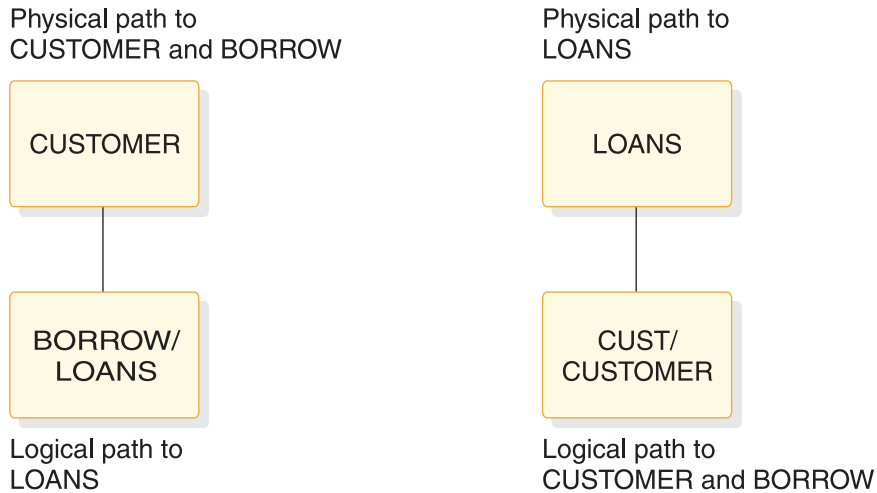
Figure 104. Paths for physical insert rule example

```
   ISRT 'CUSTOMER'  STATUS CODE='  '
ISRT 'BORROW'  STATUS CODE='  ' ('IX' if LOANS does not exist)
```

Figure 105. ISRT and status codes for physical insert rule example

The following two figures show a logical insert rule example.
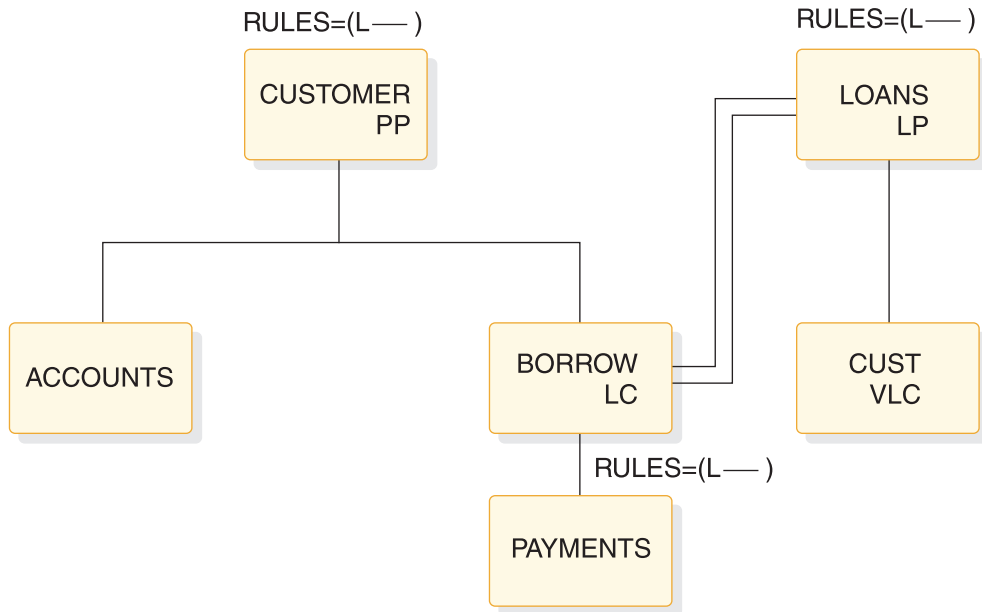
Figure 106. Logical insert rule example

```
ISRT 'LOANS'  STATUS CODE='  '
ISRT 'CUST'  STATUS CODE='IX'
```

Figure 107. ISRT and status codes for logical insert rule example

The IX status code shown in the preceding figure is the result of omitting the
concatenated segment CUST/CUSTOMER in the second call. IMS checked for the
key of the CUSTOMER segment (in the I/O area) and failed to find it. With the L
insert rule, the concatenated segment must be inserted to create a logical path.

The following two figures show a virtual insert rule example.
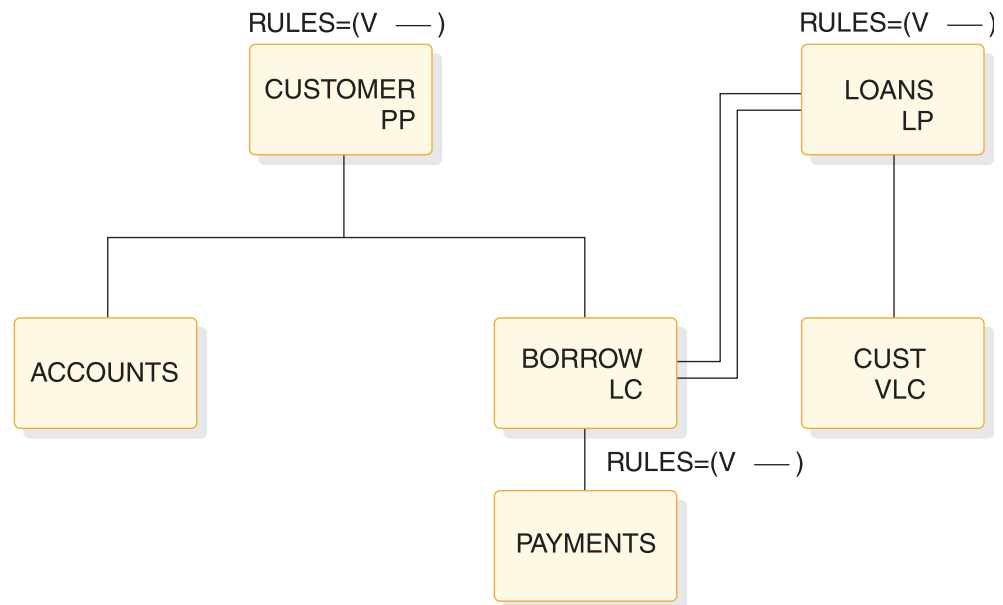
Figure 108. Virtual insert rule example

```
ISRT 'CUSTOMER'   STATUS CODE='  '
ISRT 'BORROW/LOANS'  STATUS CODE='  '
```

Figure 109. ISRT and status codes for virtual insert rule example

The code above replaces the LOANS segment if present, and insert the LOANS segment if not. The V insert rule is a powerful option.

Insert rules summary

The following summarizes the insert rules P, L, and V.

Specifying the insert rule as P prevents inserting the destination parent as part of a concatenated segment. A destination parent can only be inserted using the physical path. If the insert creates a logical path, only the logical child needs to be inserted.

Specifying the insert rule as L on the logical and physical parent allows insertion using either the physical path or the logical path as part of a concatenated segment. When inserting a concatenated segment, if the destination parent already exists it remains unchanged and the logical child is connected to it. If the destination parent does not exist, it is inserted. In either case, the logical child is inserted if it is not a duplicate, and the destination parent's insert rule is not violated.

The V insert rule is the most powerful of the three rules. The V insert rule is the most powerful because it will insert the destination parent (inserted as a concatenated segment using the logical path) if the parent did not previously exist, or otherwise replace the existing destination parent with the inserted destination parent.

Replace rules

The replace rules are applicable to the physical parent, logical parent, and logical child segments of a logical path.

The following is a description of how the replace rules work:

- When RULES=P is specified, the segment can only be replaced when retrieved using a physical path. If this rule is violated, no data is replaced and an RX status code is returned.
- When RULE=L is specified, the segment can only be replaced when retrieved using a physical path. If this rule is violated, no data is replaced. However, no RX status code is returned, and a blank status code is returned.
- When RULES=V is specified, the segment can be replaced when retrieved by either a physical or logical path.

Related concepts:

"Replace rule status codes"

The replace call

A replace operation can be done only on that portion of a concatenated segment to which an application program is data sensitive.

If no data is changed in a segment, no data is replaced. Therefore, no replace rule is violated. The replace rule is not checked for a segment that is part of a concatenated segment but is not retrieved.

For all DL/I calls, either an error is detected and an error status code returned (in which case no data is changed), or the required changes are made to all segments affected by the call. Therefore, if the required function cannot be performed for both parts of the concatenated segment, an error status code is returned, and no change is made to either the logical child or the destination parent.

Replace rule status codes

The status code returned to an application program indicates the first violation of the replace rule that was detected.

These status codes are as follows:

- AM—a replace was attempted and the value of PROCOPT is not "R"
- DA—the key field of a segment or a non-replaceable field was changed
- RX—the replace rule was violated

Replace rules summary

The tables below summarize the replace rules.

Specifying the replace rule as P, on any segment in a logical relationship, prevents replacing that segment except when it is retrieved using its physical path. When the replace rule for the logical parent is specified as L, IMS returns a blank status code without replacing any data when the logical parent is accessed concatenated with the logical child. Because the logical child has been accessed by its physical path, its replace rule can be any of the three. So, using the replace rule allows the selective replacement of the logical child half of the concatenation and a blank status code. Specifying a replace rule of V, on any segment of a logical relationship, allows replacing that segment by either its physical or logical path.

The tables that follow the figures below show all of the possible combinations of replace rules that can be specified. They show what actions take place for each combination when a call is issued to replace a concatenated segment in a logical database. The tables are based on the databases and logical views shown in the following figures.
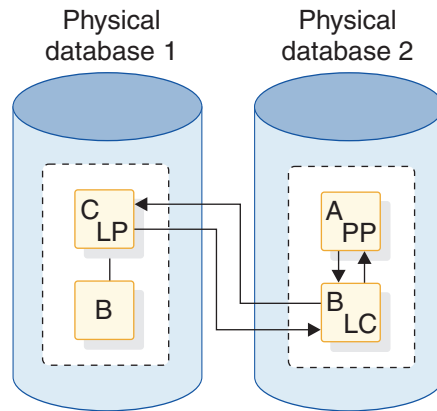
Figure 110. Physical databases for replace rules tables
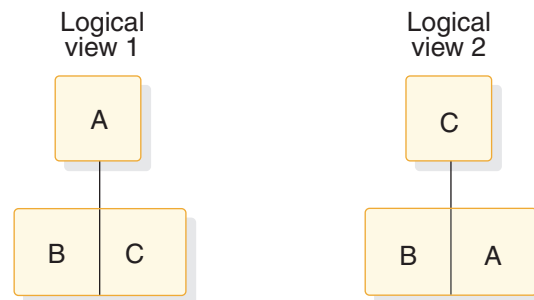
Figure 111. Logical views for replace rules table

Replace rules for logical view 1

Table 58. Replace rules for logical view 1.

Replace rule specified		Segment attempting to replace		Status code	Data replaced?	
B	**C**	**B**	**C**		**B**	**C**
P	P	X			Y	
P	P		X	RX		N
P	P	X	X	RX	N	N
P	L	X			Y	
P	L		X			N
P	L	X	X		Y	N
P	V	X			Y	
P	V		X			Y
P	V	X	X		Y	Y
L	P	X			Y	
L	P		X	RX		N
L	P	X	X	RX	N	N
L	L	X			Y	
L	L		X			N
L	L	X	X		Y	N

Table 58. Replace rules for logical view 1 (continued).

Replace rule specified		Segment attempting to replace		Status code	Data replaced?	
B	**C**	**B**	**C**		**B**	**C**
L	V	X			Y	
L	V		X			Y
L	V	X	X		Y	Y
V	P	X			Y	
V	P		X	RX		N
V	P	X	X	RX	N	N
V	L	X			Y	
V	L		X			N
V	L	X	X		Y	N
V	V	X			Y	
V	V		X			Y
V	V	X	X		Y	Y

Replace rules for logical view 2

Table 59. Replace rules for logical view 2.

Replace rule specified		Segment attempting to replace		Status code	Data replaced?	
B	**A**	**B**	**A**		**B**	**A**
P	P	X		RX	N	
P	P		X	RX		N
P	P	X	X	RX	N	N
P	L	X		RX	N	
P	L		X			N
P	L	X	X	RX	N	N
P	V	X		RX	N	
P	V		X			Y
P	V	X	X	RX	N	N
L	P	X			N	
L	P		X	RX		N
L	P	X	X	RX	N	N
L	L	X			N	
L	L		X			N
L	L	X	X		N	N
L	V	X			N	
L	V		X			Y
L	V	X	X		N	Y
V	P	X			Y	
V	P		X	RX		N

Table 59. Replace rules for logical view 2 (continued).

Replace rule specified		Segment attempting to replace		Status code	Data replaced?	
B	**A**	**B**	**A**		**B**	**A**
V	P	X	X	RX	N	N
V	L	X			Y	
V	L		X			N
V	L	X	X		Y	N
V	V	X			Y	
V	V		X			Y
V	V	X	X		Y	Y

Physical replace rule example

The following figure and the code that follows the figure show a physical replace rule example.
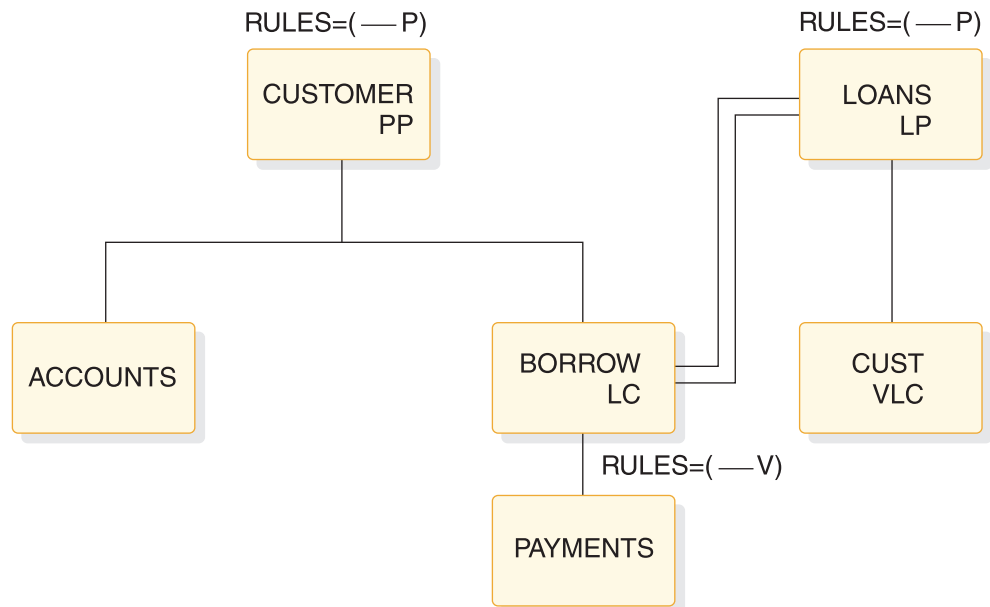
Figure 112. Physical replace rule example

```
GHU 'CUSTOMER'   STATUS CODE='  '
REPL             STATUS CODE='  '
GHN  'BORROW/LOANS'  STATUS CODE='  '
REPL             STATUS CODE='RX'
```

Figure 113. Calls and status codes for physical replace rule example

The P replace rule prevents replacing the LOANS segment as part of a concatenated segment. Replacement must be done using the segment's physical path.

Related concepts:

"Logical replace rule example" on page 296

"Virtual replace rule example" on page 296

Logical replace rule example

The following figure and the code that follows the figure show a logical replace rule example.
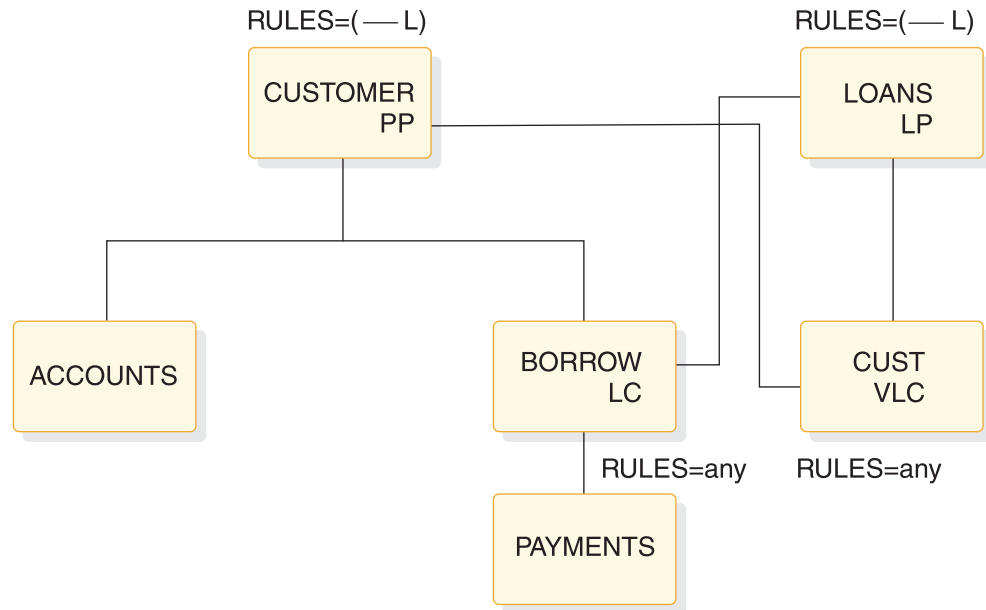
Figure 114. Logical replace rule example

```
GHU 'CUSTOMER'
    'BORROW/LOANS'   STATUS CODE='  '
REPL                 STATUS CODE='  '
```

Figure 115. Calls and status codes for logical replace rule example

As shown in the preceding figure, the L replace rule prevents replacing the LOANS segment as part of a concatenated segment. Replacement must be done using the segment's physical path. However, the status code returned is blank. The BORROW segment, accessed by its physical path, is replaced. Because the logical child is accessed by its physical path, it does not matter which replace rule is selected.

The L replace rule allows replacing only the logical child half of the concatenation, and the return of a blank status code.

Related concepts:

"Physical replace rule example" on page 295

"Virtual replace rule example"

Virtual replace rule example

The following figure and the code that follows the figure show a virtual replace rule example.
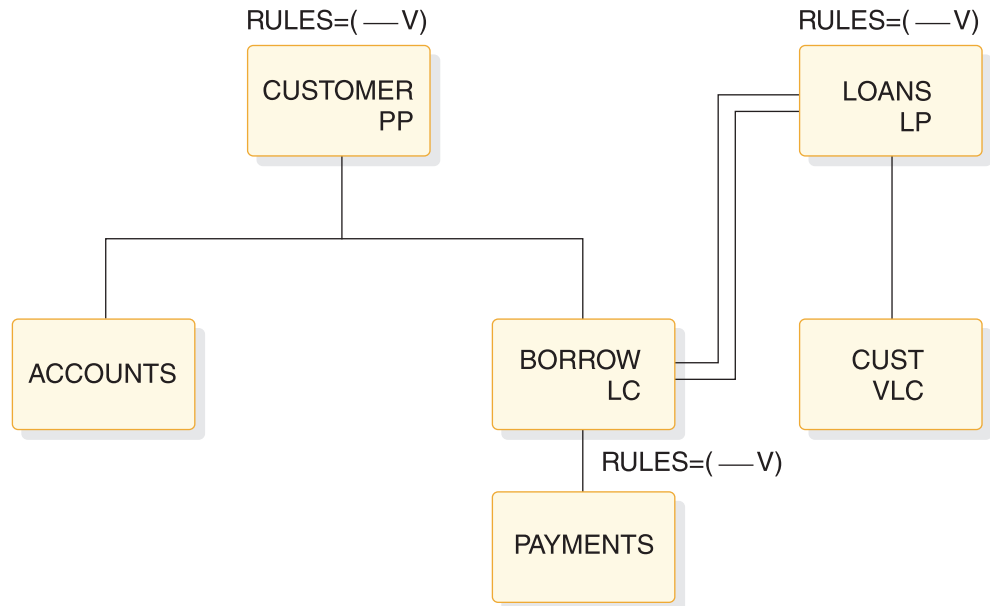
Figure 116. Virtual replace rule example

```
GHU 'LOANS'
    'CUST/CUSTOMER'  STATUS CODE='  '
REPL               STATUS CODE='  '
```

Figure 117. Calls and status codes for virtual replace rule example

As shown in the code above, the V replace rule allows replacing the CUSTOMER segment using its logical path as part of a concatenated segment.

Related concepts:

"Logical replace rule example" on page 296

"Physical replace rule example" on page 295

Delete rules

The following topics provide a description of how the delete values work for the logical parent, physical parent, and logical child.

Logical parent delete rules

The following list describes what happens when a logical parent is deleted when RULES=P, RULES=L, or RULES=V is specified.

- When RULES=P is specified, the logical parent must be logically deleted before a DLET call is effective against it or any of its physical parents. Otherwise, the call results in a DX status code, and no segments are deleted. However, if a delete request is made against a segment as a result of propagation across a logical relationship, then the P rule acts like the L rule that follows.

- When RULES=L is specified, either physical or logical deletion can occur first. When the logical parent is processed by a DLET call, all logical children are logically deleted, but the logical parent remains accessible from its logical children.

- When RULES=V is specified, a logical parent is deleted along its physical path explicitly when deleted by a DLET call. All of its logical children are logically deleted, although the logical parent remains accessible from these logical children.

A logical parent is deleted along its physical path implicitly when it is no longer involved in a logical relationship. A logical parent is no longer involved in a logical relationship when:

- It has no logical children pointing to it (its logical child counter is zero, if it has any)
- It points to no logical children (all of its logical child pointers are zero, if it has any)
- It has no physical children that are also real logical children

Physical parent (virtual pairing only) delete rules

The following list describes the delete rules for a physical parent with virtual pairing only.

- PHYSICAL/LOGICAL/VIRTUAL is meaningless.
- BIDIRECTIONAL VIRTUAL means a physical parent is automatically deleted along its physical path when it is no longer involved in a logical relationship. A physical parent is no longer involved in a logical relationship when:
  - It has no logical children pointing to it (its logical child counter is zero, if it has one)
  - It points to no logical children (all of its logical child pointers are zero, if it has any)
  - It has no physical children that are also real logical children

Logical child delete rules

The following list describes what happens when a logical child is deleted when RULES=P, RULES=L, or RULES=V is specified.

- When RULES=P is specified, the logical child segment must be logically deleted first and physically deleted second. If physical deletion is attempted first, the DLET call issued against the segment or any of its physical parents results in a DX status code, and no segments are deleted. If a delete request is made against the segment as a result of propagation across a logical relationship, or if the segment is one of a physically paired set, then the rule acts like the L rule that follows.
- When RULES=L is specified, deletion of a logical child is effective for the path for which the delete was requested. Physical and logical deletion of the logical child can be performed in any order. The logical child and any physical dependents remain accessible from the non-deleted path.
- When RULES=V is specified, a logical child is both logically and physically deleted when it is deleted through either its logical or physical path (setting either the PD or LD bits sets both bits). If this rule is coded on only one logical child segment of a physically paired set, it acts like the L rule.

Note: For logical children involved in unidirectional logical relationships, the meaning of all three rules is the same, so any of the three rules can be specified.

Examples using the delete rules

The following series of figures shows the use of the delete rules for each of the segment types for which the delete rule can be coded (logical and physical parents and their logical children).

Only the rule pertinent to the example is shown in each figure. The explanation accompanying the example applies only to the specific example.
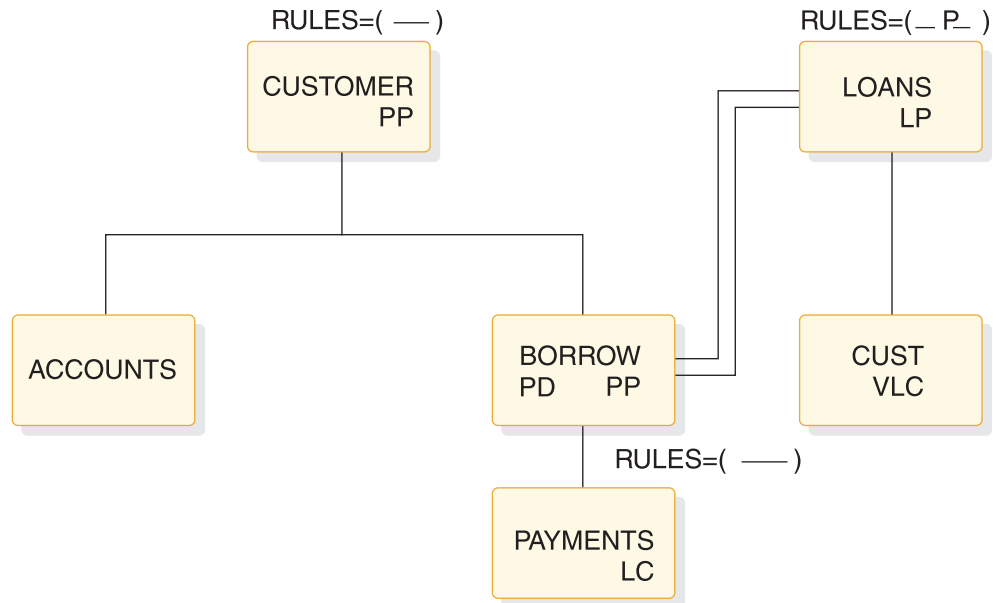
Figure 118. Logical parent, virtual pairing—physical delete rule example
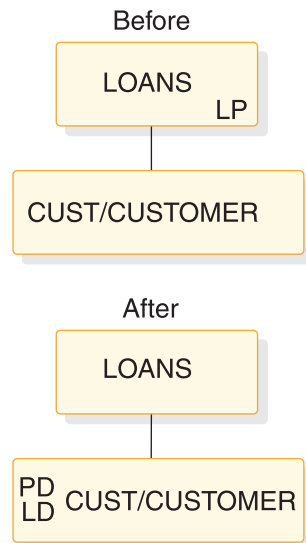
Figure 119. Logical parent, physical pairing—physical delete rule example: before and after

```
GHU 'LOANS'  STATUS='  '
DLET         STATUS='  '
```

Figure 120. Logical parent, physical pairing—physical delete rule example: database calls

The physical delete rule requires that all logical children be previously physically deleted. Physical dependents of the logical parent are physically deleted.

The DLET status code will be 'DX' if all of the logical children were not previously physically deleted. All logical children are logically deleted. The LD bit is set on in the physical logical child BORROW.
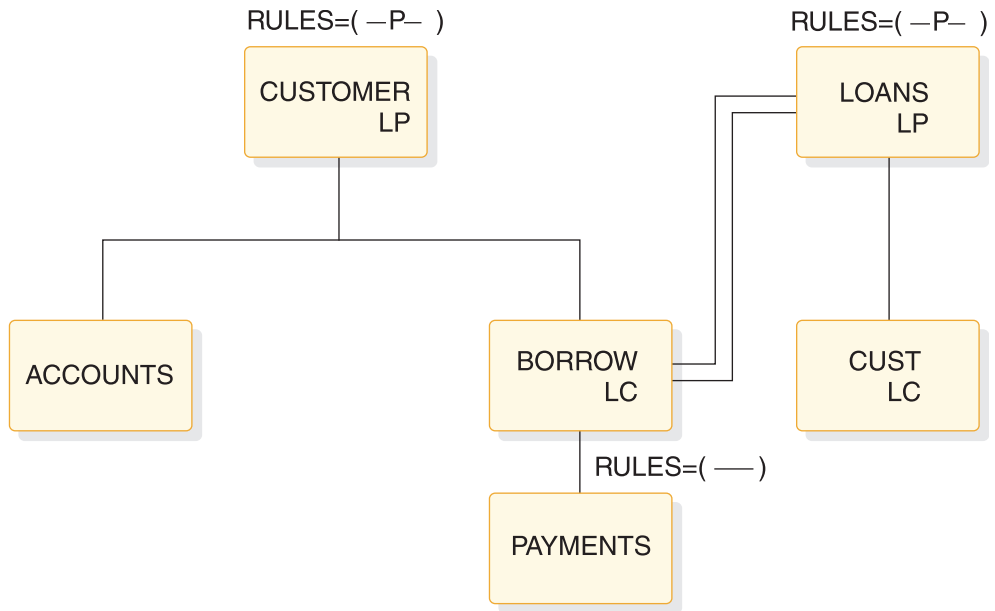
Figure 121. Logical parent, physical pairing—physical delete rule example
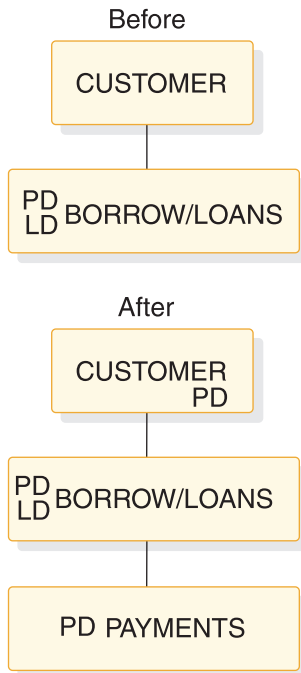
Figure 122. Logical Parent, physical pairing—physical delete rule example: before and after

```
GHU 'CUSTOMER'   STATUS='  '
DLET                STATUS='  '
```

Figure 123. Logical parent, physical pairing—physical delete rule example: calls and status codes

The physical delete rule requires that:
- All logical children be previously physically deleted.
- Physical children paired to the logical child be previously deleted.

CUSTOMER, the logical parent, has been physically deleted. Both the logical child and its pair had previously been physically deleted. (The PD and LD bits are set on the before figure of the BORROW/LOANS.)
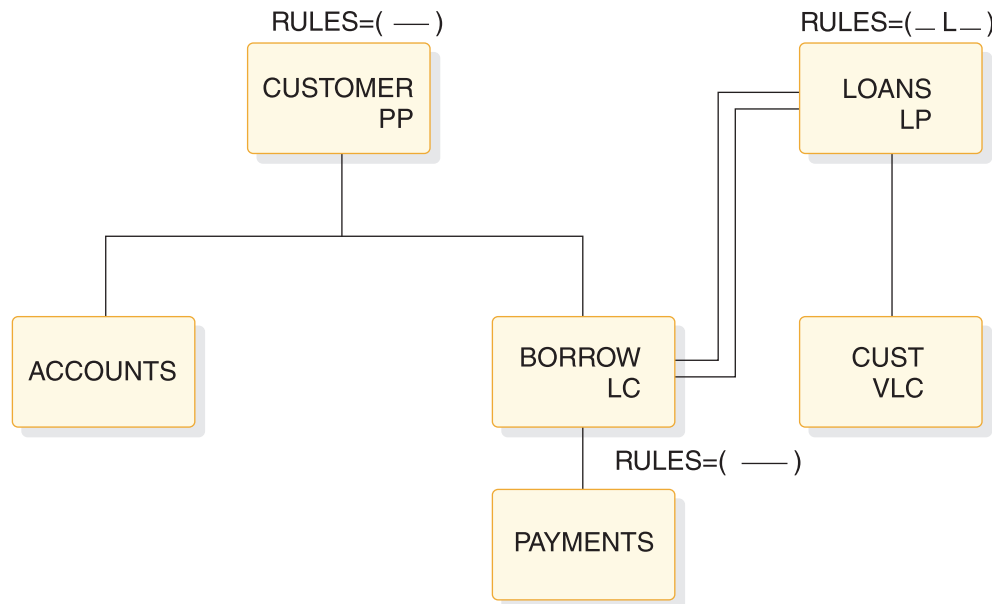
Figure 124. Logical parent, virtual pairing—logical delete rule example
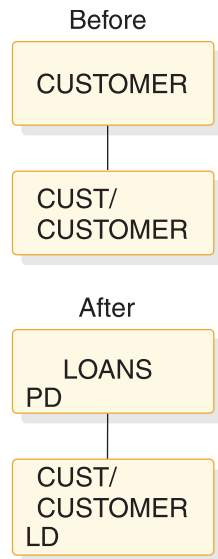
Figure 125. Logical parent, virtual pairing—logical delete rule example: before and after

```
GHU 'LOANS'   STATUS='  '
DLET          STATUS='  '
```

Figure 126. Logical parent, virtual pairing—logical delete rule example: calls and status codes

The logical delete rule allows either physical or logical deletion first; neither causes the other. Physical dependents of the logical parent are physically deleted.

The logical parent LOANS remains accessible from its logical children. All logical children are logically deleted. The LD bit is set on in the physical child BORROW.

The processing and results shown in Figure 124 on page 301 would be the same if the logical parent LOANS delete rule were virtual instead of logical. The example that follows is an additional one to explain the logical delete rule.
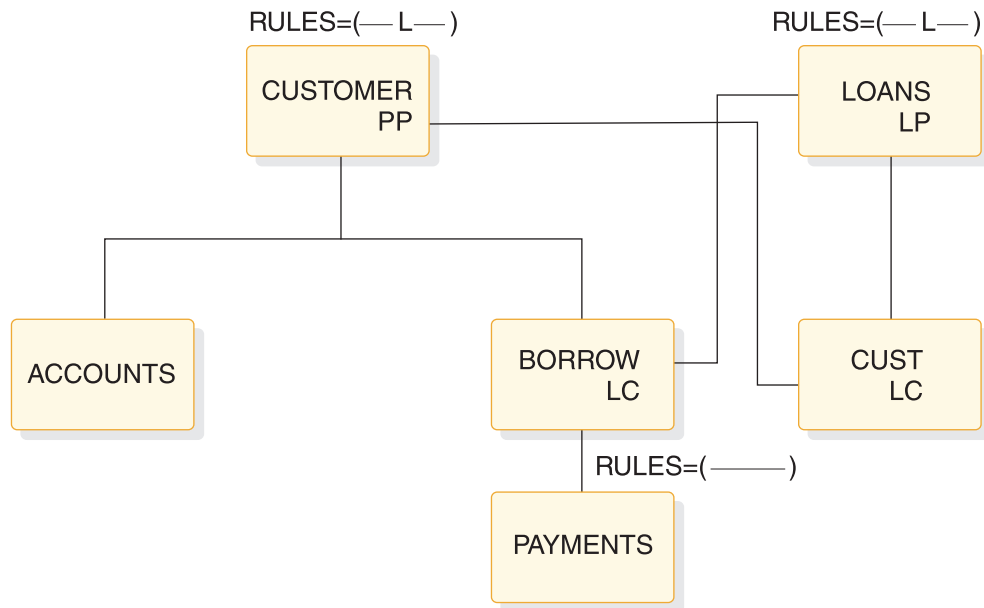
Figure 127. Logical parent, physical pairing—logical delete rule example
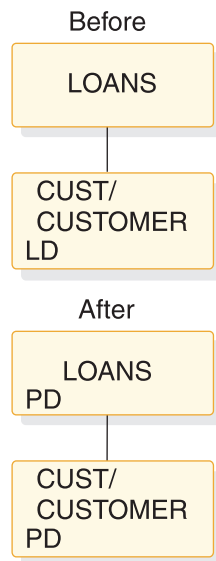
Figure 128. Logical parent, physical pairing—logical delete rule example: before and after

```
GHU 'LOANS'   STATUS='  '
DLET          STATUS='  '
```

Figure 129. Logical parent, physical pairing—logical delete rule example: calls and status codes

The logical delete rule allows either physical or logical deletion first; neither causes the other. Physical dependents of the logical parent are physically deleted.

The logical parent LOANS remains accessible from its logical children. All physical children are physically deleted. Paired logical children are logically deleted.

The processing and results shown in Figure 127 on page 302 would be the same if the logical parent LOANS delete rule were virtual instead of logical. An additional example to explain the virtual delete rule follows in the following figure.
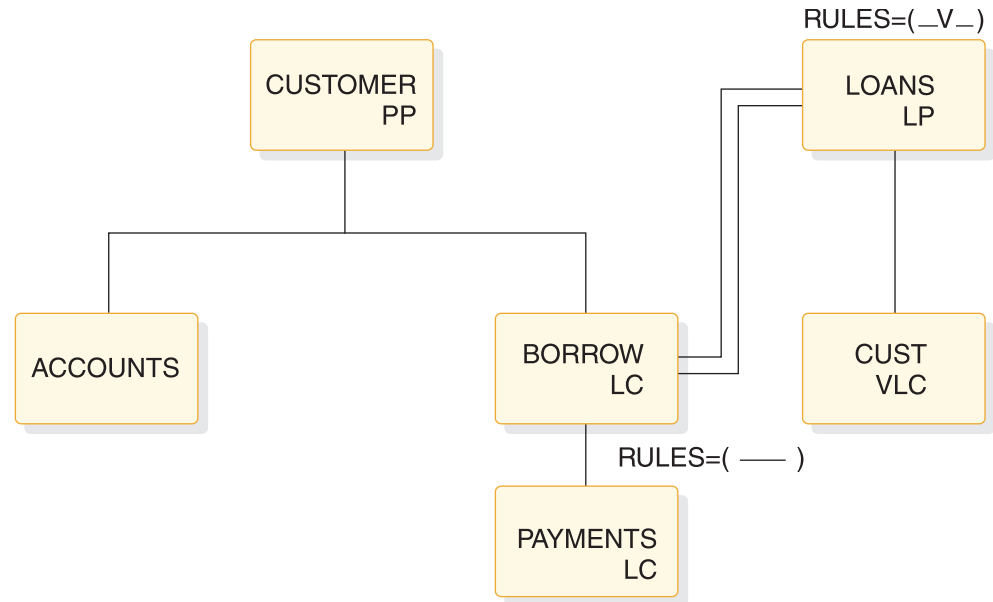
Figure 130. Logical parent, virtual pairing—virtual delete rule example
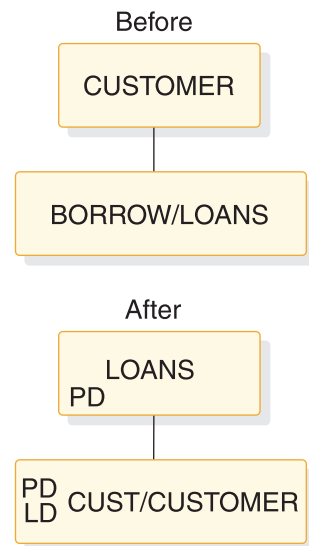
Figure 131. Logical parent, virtual pairing—virtual delete rule example: before and after

```
GHU 'CUSTOMER'
    'BORROW/LOANS'  STATUS='  '
DLET                STATUS='  '
```

Figure 132. Logical parent, virtual pairing—virtual delete rule example: calls and status codes

The virtual delete rule allows explicit and implicit deletion. Explicit deletion is the same as using the logical rule. Implicit deletion causes the logical parent to be physically deleted when the last logical child is physically deleted.

Physical dependents of the logical child are physically deleted. The logical parent is physically deleted. Physical dependents of the logical parent are physically deleted. The LD bit is set on in the physical logical child BORROW.
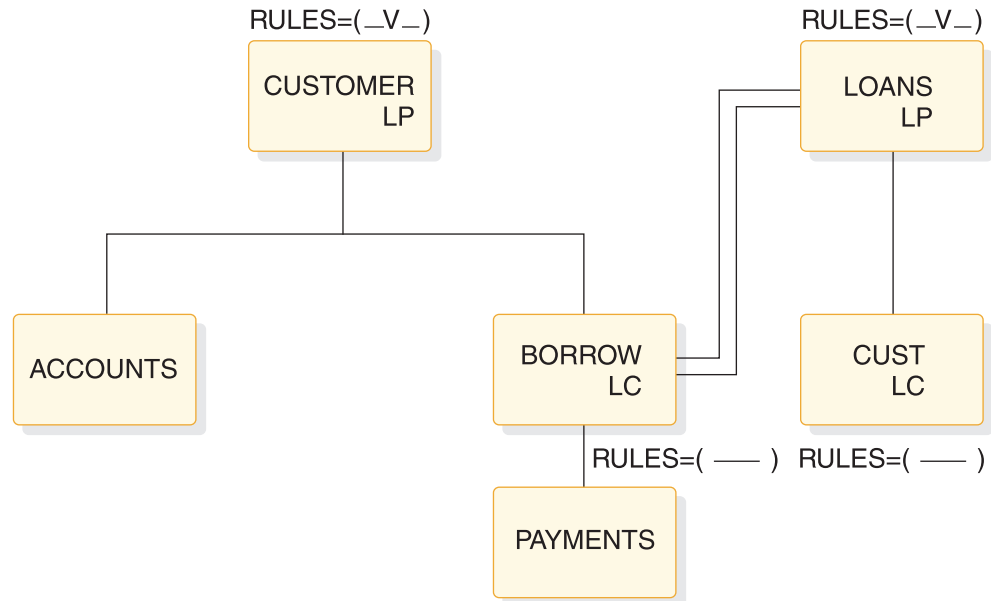
Figure 133. Logical parent, physical pairing—virtual delete rule example
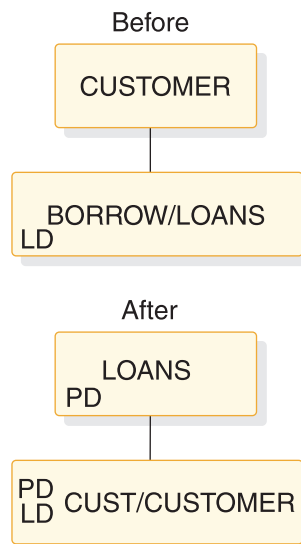
Figure 134. Logical parent, physical pairing—virtual delete rule example: before and after

```
GHU 'CUSTOMER'
    'BORROW/LOANS'  STATUS='  '
DLET                STATUS='  '
```

Figure 135. Logical parent, physical pairing—virtual delete rule example: calls and status

The virtual delete rule allows explicit and implicit deletion. Explicit deletion is the same as using the logical rule. Implicit deletion causes the logical parent to be physically deleted when the last logical child is physically and logically deleted.

The logical parent is physically deleted. Any physical dependents of the logical parent are physically deleted.

Note: The CUST segment must be physically deleted before the DLET call is issued. The LD bit is set on in the BORROW segment.
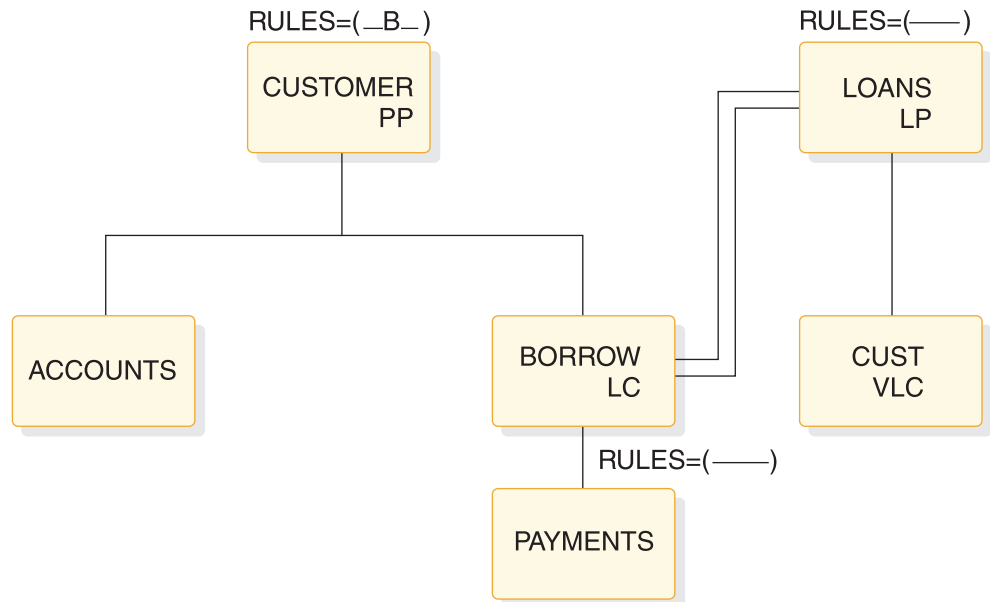
Figure 136. Physical parent, virtual pairing—bidirectional virtual example
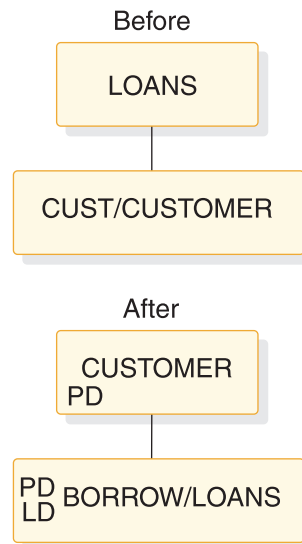
Figure 137. Physical parent, virtual pairing—bidirectional virtual example: before and after

```
GHU 'LOANS'
    'CUSTOMER'   STATUS='  '
DLET             STATUS='  '
```

Figure 138. Deleting last logical child deletes physical parent

The bidirectional virtual rule for the physical parent has the same effect as the virtual rule for the logical parent.

When the last logical child is logically deleted, the physical parent is physically deleted. The logical child (as a dependent of the physical parent) is physically deleted. All physical dependents of the physical parent are physically deleted. That is, ACCOUNTS (not shown), BORROW, and PAYMENT are physically deleted.
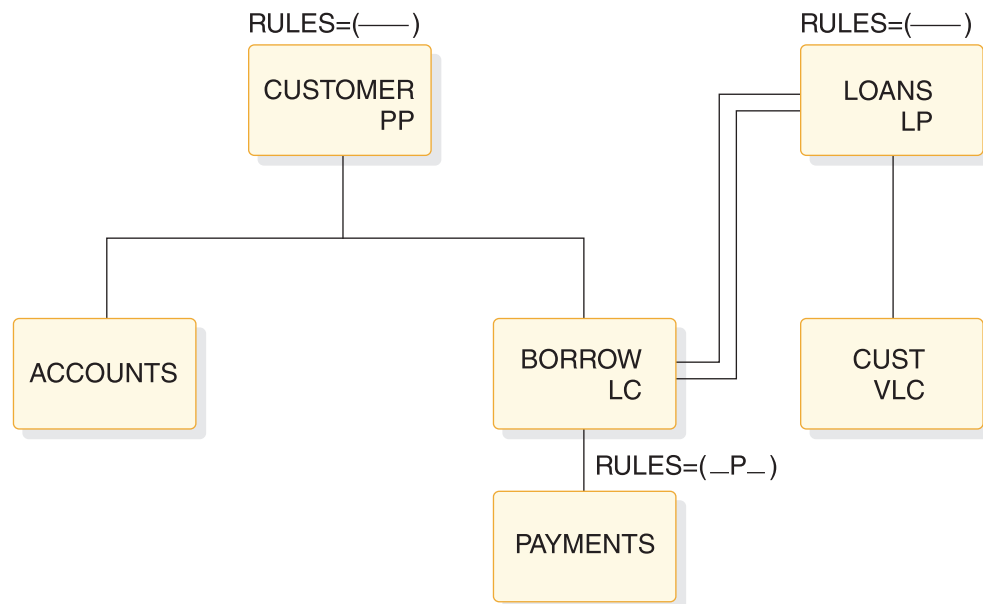
Figure 139. Logical child, virtual pairing—physical delete rule example

```
GHU 'LOANS'         STATUS='  '
    'CUST/CUSTOMER'
DLET                STATUS='  '

GHU 'CUSTOMER'      STATUS='  '
    'BORROW/LOANS'
DLET                STATUS='  '
```

Figure 140. Logical child, virtual pairing—physical delete rule example: deleting the logical child

The physical delete rule requires that the logical child be logically deleted first. The LD bit is now set in the BORROW segment.

The logical child can be physically deleted only after being logically deleted. After the second delete, the LD and PD bits are both set. The physical delete of the logical child also physically deleted the physical dependents of the logical child. The PD bit is set.
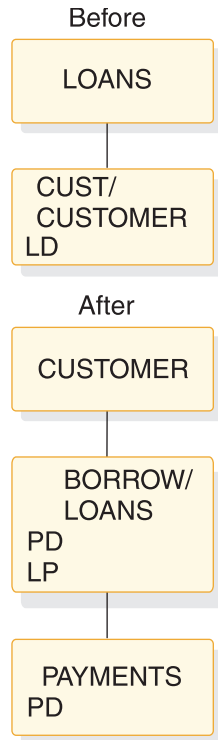
Figure 141. Logical child, virtual pairing—physical delete rule example: before and after
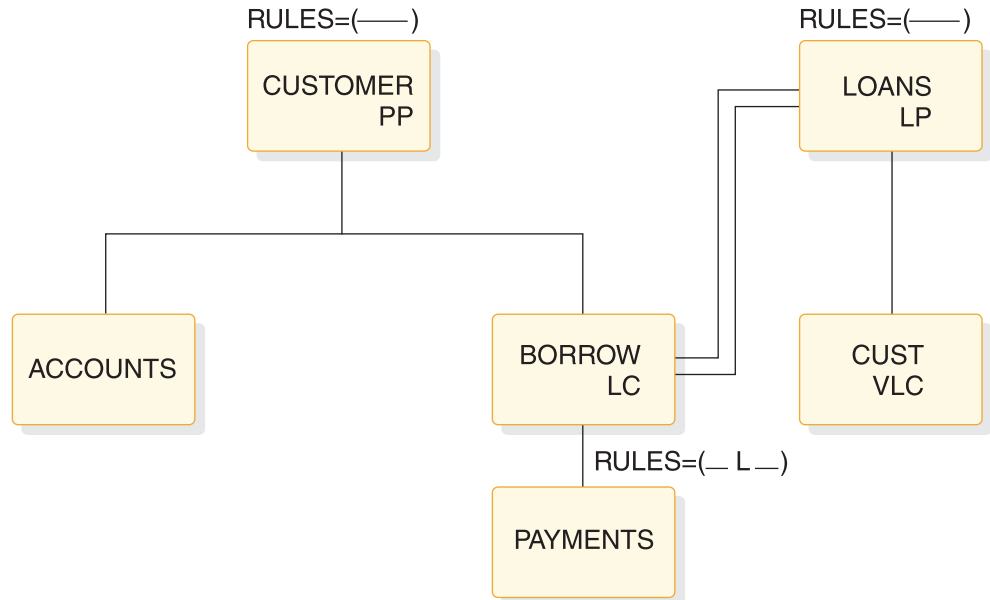
Figure 142. Logical child, virtual pairing—logical delete rule example

```
GHU 'CUSTOMER   STATUS=' '
    'BORROW/LOANS'
DLET            STATUS=' '

GHU 'LOANS'     STATUS=' '
    'CUST/CUSTOMER'
DLET            STATUS=' '
```

Figure 143. Logical child, virtual pairing—logical delete rule example: calls and status

The logical delete rule allows the logical child to be deleted physically or logically first. Physical dependents of the logical child are physically deleted, but they remain accessible from the logical path that is not logically deleted.

The delete of the virtual logical child sets the LD bit on in the physical logical child BORROW (BORROW is logically deleted).

Before

CUSTOMER

BORROW/
LOANS
PD

PAYMENTS
PD

After

LOANS
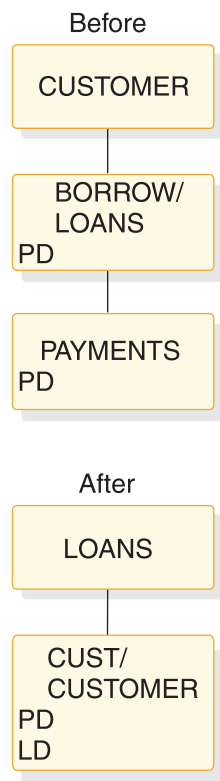
CUST/
CUSTOMER
PD
LD

Figure 144. Logical child, virtual pairing—logical delete rule example: before and after
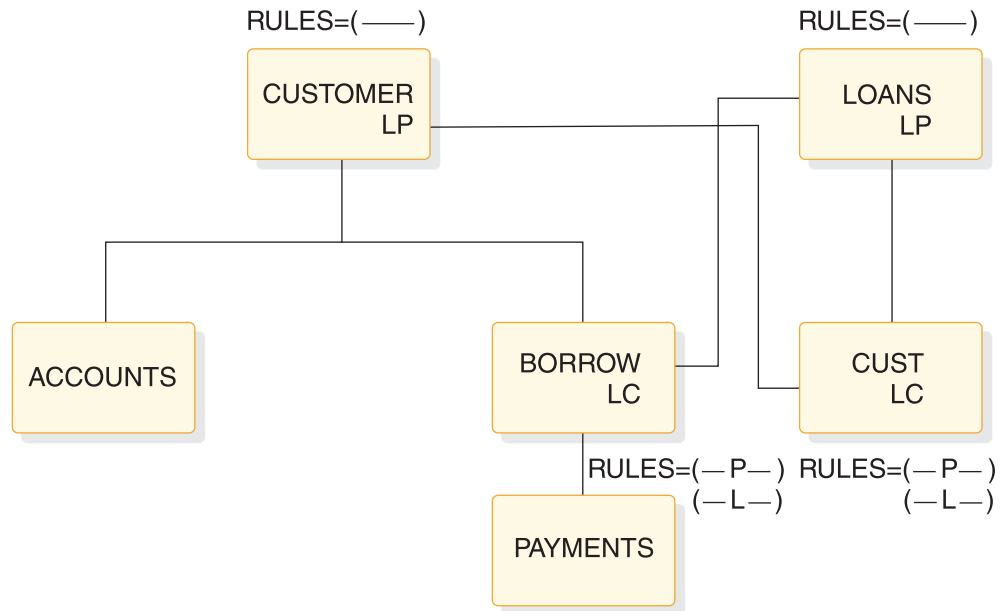
Figure 145. Logical child, physical pairing—physical or logical delete rule example

```
GHU 'CUSTOMER   STATUS='  '
    'BORROW/LOANS'
DLET            STATUS='  '

GHU 'LOANS'     STATUS='  '
    'CUST/CUSTOMER'
DLET            STATUS='  '
```

Figure 146. Logical child, physical pairing—physical or logical delete rule example: calls and status

With the physical or logical delete rule, each logical child must be deleted from its physical path. Physical dependents of the logical child are physically deleted, but they remain accessible from the paired logical child that is not deleted.

Physically deleting BORROW sets the LD bit on in CUST. Physically deleting CUST sets the LC bit on in the BORROW segment.

Before

CUSTOMER

BORROW/
LOANS
PD

PAYMENTS
PD

After

LOANS
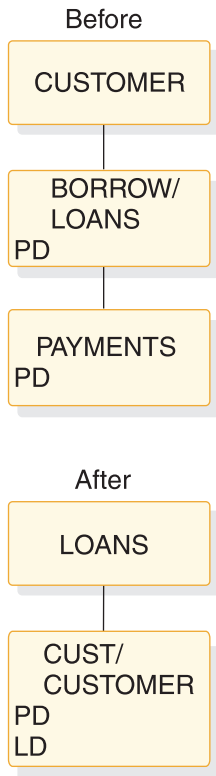
CUST/
CUSTOMER
PD
LD

Figure 147. Logical child, physical pairing—physical or logical delete rule example: before and after

RULES=(——)

CUSTOMER
PP

RULES=(——)

LOANS
LP

ACCOUNTS

BORROW
LC

CUST
VLC

RULES=(_V_)

PAYMENTS
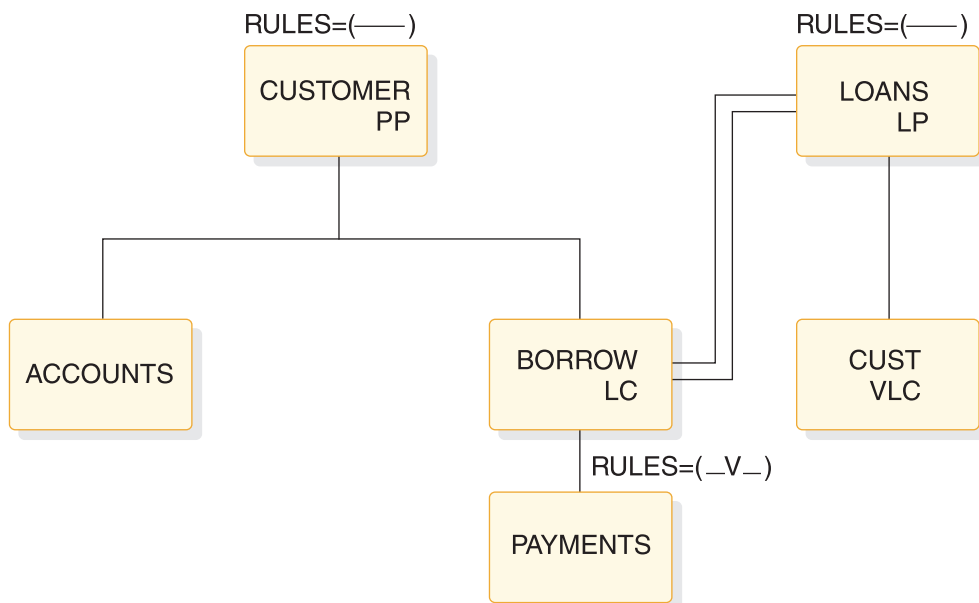
Figure 148. Logical child, virtual pairing—virtual delete rule example

```
GHU 'CUSTOMER    STATUS=' '
    'BORROW/LOANS'
DLET             STATUS=' '

GHU 'LOANS'      STATUS='GE'
    'CUST/CUSTOMER'
```

Figure 149. Logical child, virtual pairing—virtual delete rule example: calls and status

The virtual delete rule allows the logical child to be deleted physically and logically. Deleting either path deletes both parts. Physical dependents of the logical child are physically deleted.

The previous delete deleted both paths because the delete rule is virtual. Deleting either path deletes both.
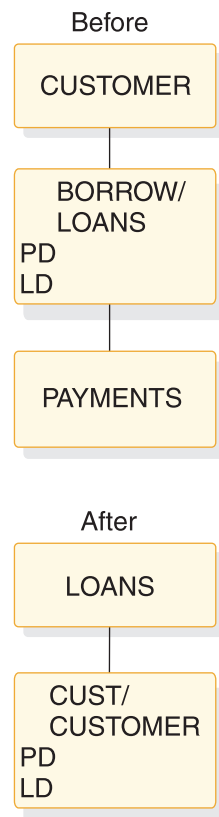
Figure 150. Logical child, virtual pairing—virtual delete rule example: before and after
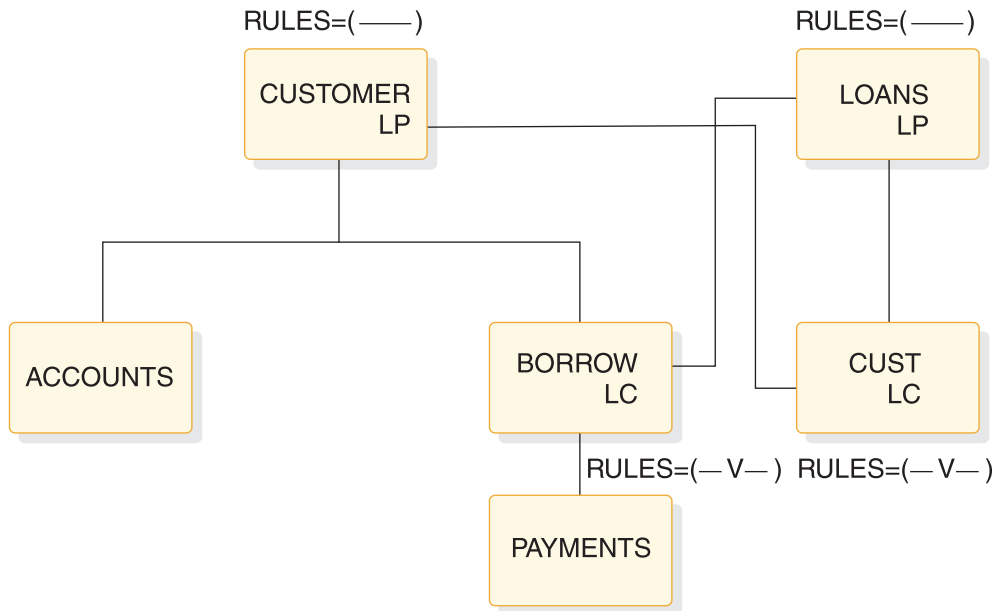
Figure 151. Logical child, physical pairing—virtual delete rule example

```
GHU 'CUSTOMER    STATUS='  '
DLET             STATUS='  '

GHU 'LOANS'      STATUS='GE'
    'CUST/CUSTOMER'
```

Figure 152. Logical child, physical pairing—virtual delete rule example: calls and status

With the virtual delete rule, deleting either logical child deletes both paired logical children. (Notice the PD and LD bit is set on in both.) Physical dependents of the logical child are physically deleted.

Physical dependents of the logical child are physically deleted.

Before

```
┌──────────────────┐
│                  │
│    CUSTOMER      │
│                  │
└──────────────────┘
         │
┌──────────────────┐
│    BORROW/       │
│    LOANS         │
│ PD               │
│ LD               │
└──────────────────┘
         │
┌──────────────────┐
│    PAYMENTS      │
│ PD               │
└──────────────────┘
```

After

```
┌──────────────────┐
│                  │
│    LOANS         │
│                  │
└──────────────────┘
         │
┌──────────────────┐
│    CUST/         │
│    CUSTOMER      │
│ PD               │
│ LD               │
└──────────────────┘
```

Figure 153. Logical child, physical pairing—virtual delete rule example: before and after

Accessibility of deleted segments

Segments that are either physically deleted or logically deleted remains accessible under certain circumstances.

A physically deleted segment remains accessible under the following circumstances:

- A physical dependent of the deleted segment is a logical parent accessible from its logical children.
- A physical dependent of the deleted segment is a logical child accessible from its logical parent.
- A physical parent of the deleted segment is a logical child accessible from its logical parent. The deleted segment in this case is variable intersection data in a bidirectional logical relationship.

A logically deleted logical child cannot be accessed from its logical parent.

Neither physical or logical deletion prevents access to a segment from its physical or logical children. Because logical relationships provide for inversion of the physical structure, a segment can be physically or logically deleted or both, and still be accessible from a dependent segment because of an active logical relationship. A physically deleted root segment can be accessed when it is defined as a dependent segment in a logical DBD. The logical DBD defines the inversion of the physical DBD. The following figure shows the accessibility of deleted segments.

When the physical dependent of a deleted segment is a logical parent with logical children that are not physically deleted, the logical parent and its physical parents are accessible from those logical children.
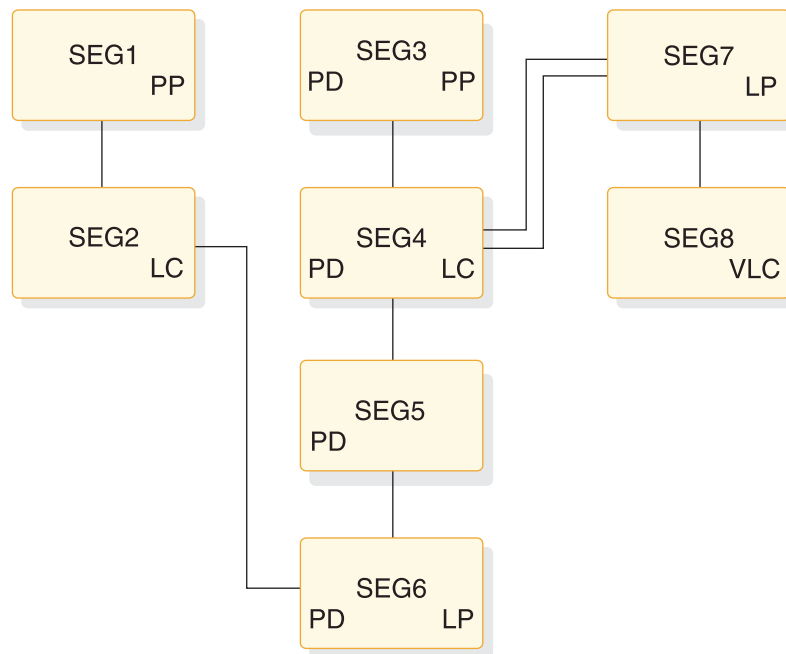
Figure 154. (Part 1 of 5). Example of deleted segments accessibility

The physical structure in preceding figure shows that SEG3, SEG4, SEG5, and SEG6 have been physically deleted, probably by issuing a DLET call for SEG3. This resulted in all of SEG3's dependents being physically deleted. (SEG6's delete rule is not P, or a 'DX' status code would be issued.)

SEG3, SEG4, SEG5, and SEG6 remain accessible from SEG2, the logical child of SEG6. This is because SEG2 is not physically deleted. However, physical dependents of SEG6 cannot be accessible, and their DASD space is released unless an active logical relationship prohibits

When the physical dependent of a deleted segment is a logical child whose logical parent is not physically deleted, the logical child, its physical parents, and its physical dependents are accessible from the logical parent.

The logical child segment SEG4 remains accessible from its logical parent SEG7 (SEG7 is not physically deleted). Also accessible are SEG5 and SEG6, which are variable intersection data. The physical parent of the logical child (SEG3) is also accessible from the logical child (SEG4).

A physically and logically deleted logical child can be accessed from its physical dependents. See the following figure.
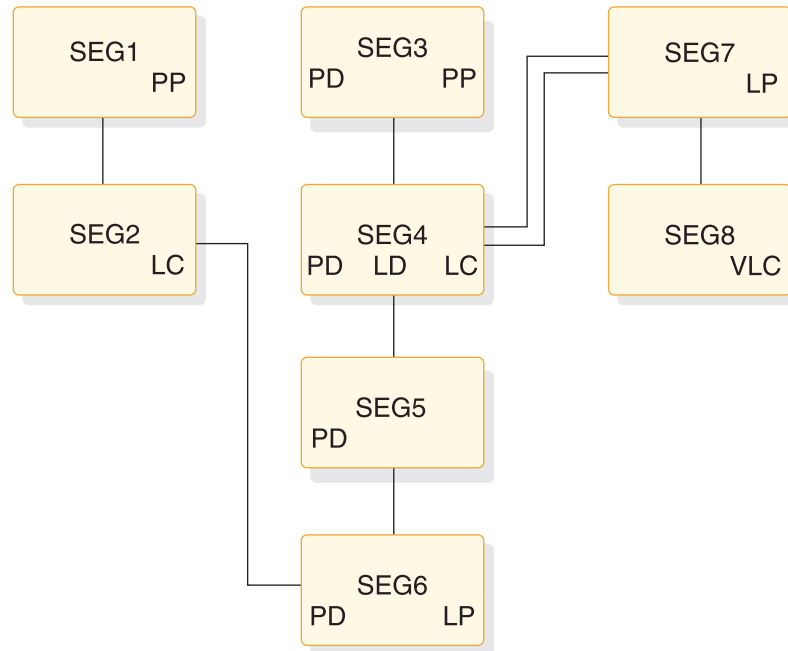
Figure 155. (Part 2 of 5). Example of deleted segments accessibility

The physical structure in the preceding figure shows that logical child SEG4 is both physically and logically deleted.

From a previous example (part 1 of 4), we know SEG6 (a logical parent) is accessible from SEG2, if that segment (its logical child) is not physically deleted. We also know that once we've accessed SEG6, its physical parents (SEG5, SEG4, SEG3) are accessible. It does not matter that the logical child is logically deleted (which is the only difference between this example and that of part 1 of 4).

The third path cannot be blocked because no delete bit exists for this path. Therefore, the logical child SEG4 is accessible from its dependents even though it is been physically and logically deleted.

When a segment accessed by its third path is deleted, it is physically deleted in its physical data base, but it remains accessible from its third path. See the following figure and code.

PCB

```
SEG1
```

```
SEG2/SEG6
```

```
SEG5
```
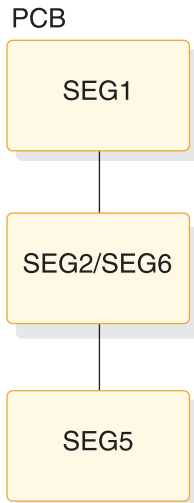
Figure 156. (Part 3 of 5). Example of deleted segments accessibility

```
GHU 'SEG5'  STATUS='  '
DLET        STATUS='  '
```

Figure 157. (Part 4 of 5). Example of deleted segments accessibility: database calls

SEG5 is physically deleted by the DLET call, and SEG 6 is physically deleted by propagation. SEG2/SEG6 has unidirectional pointers, so SEG2 was considered logically deleted before the DLET call was issued. The LD bit is only assumed to be set on. See the following figure.
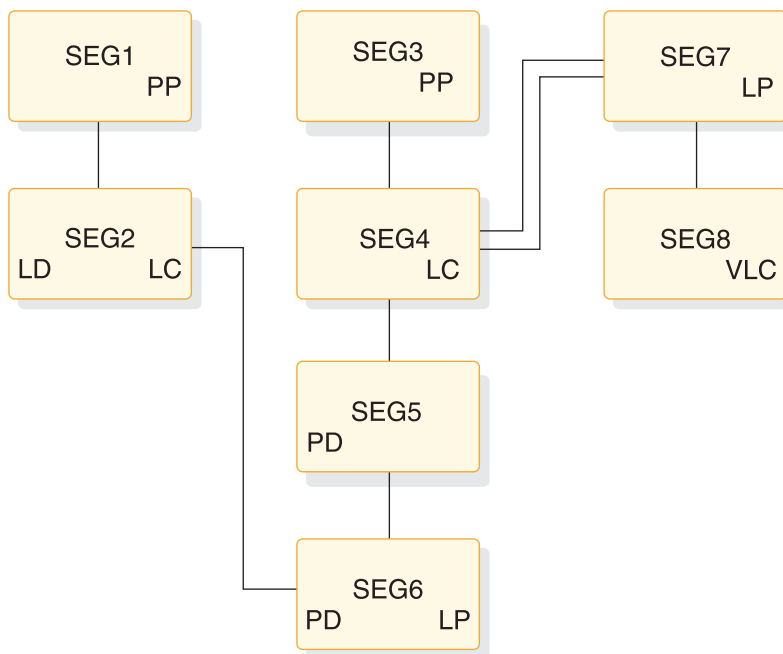
Figure 158. (Part 5 of 5). Example of deleted segments accessibility

The results are interesting. SEG5 is inaccessible from its physical parent path (from SEG4) unless SEG4 is accessed by its logical parent SEG7 (SEG5 and SEG6 are accessible as variable intersection data). SEG5 is still accessible from its third path

(from SEG6) because SEG6 is still accessible from its logical child. Thus, a segment can be physically deleted by an application program and still be accessible to that application program, using the same PCB used to delete the segment.

Possibility of abnormal termination

If a logical parent is physically and logically deleted, its DASD space is released. For this to occur, all of its logical children must be physically and logically deleted. However, the DASD space for these logical children cannot be released if the logical children have physical dependents with active logical relationships.

Accessing such a logical child from its physical dependents (both the logical child and logical parent have been physically and logically deleted) can result in a user 850 through 859 abnormal termination if one of the following occurs:

- The LPCK is not stored in the logical child
- The concatenation definition is data sensitive to the logical parent

The following figure shows an example of abnormal termination.
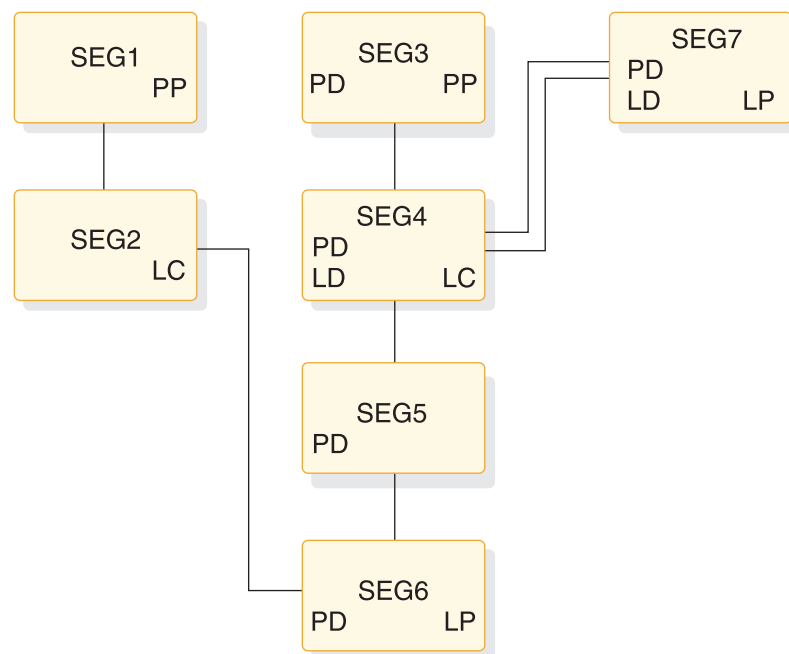
Figure 159. Example of abnormal termination

The logical parent SEG7 has been physically and logically deleted (the LD bit is never really set, but is assumed to be set. It is shown only for the purpose of illustration.) All of the logical children of the logical parent have also been physically and logically deleted. However, the logical parent has had its segment space released, whereas the logical child (SEG4) still exists. The logical child still exists because it has a physical dependent that has an active logical relationship that precludes releasing its space.

If an application program accesses SEG4 from its dependents (SEG1 to SEG2/SEG6 to SEG5), IMS must build the logical parent's concatenated key if that key is not stored in the logical child. When IMS attempts to access logical parent SEG7, abnormal termination will occur. The 850 through 859 abnormal termination codes are issued when a pointer is followed that does not lead to the expected segment.

Related concepts:

Avoiding abnormal termination

You must avoid creating a physically deleted logical child that can be accessed from below in the physical structure (using its third path). A logical child can be accessed from below if any of its physical dependents are accessible through logical paths.

Two methods exist in avoiding this situation.

- Method 1 The first method requires that logical paths to dependents be broken before the logical child is physically deleted. Breaking the logical path with method 1 is done using a P rule for the dependents as long as no physical deletes are propagated into the database. Therefore, no V rules on logical children can be allowed at or above the logical child, because, with the V rule, a propagated logical delete causes a physical delete without a P rule violation check. The L rule also causes propagation, if the PD bit is already set on, but the dependent's P rule will prevent that case. Similarly, no V rule can be allowed on any logical parent above the logical child, because the logical delete condition would cause the physical delete.

- Method 2 The second method requires breaking the logical path whenever the logical child is physically deleted. Breaking the logical path with this method is done for subordinate logical child segments using the V delete rule. Subordinate logical parent segments need to have bidirectional logical children with the V rule (must be able to reach the logical children) or physically paired logical children with the V rule. This method will not work with subordinate logical parents pointed to by unidirectional logical children.

Related concepts:

"Detecting physical delete rule violations"

Detecting physical delete rule violations

When a DLET call is issued, the delete routine scans the physical structure containing the segment to be deleted.

The delete routine scans the physical structure to determine if any segment in it uses the physical delete rule and whether that rule is being violated. The following figure and code sample show an example of violating the physical delete rule.
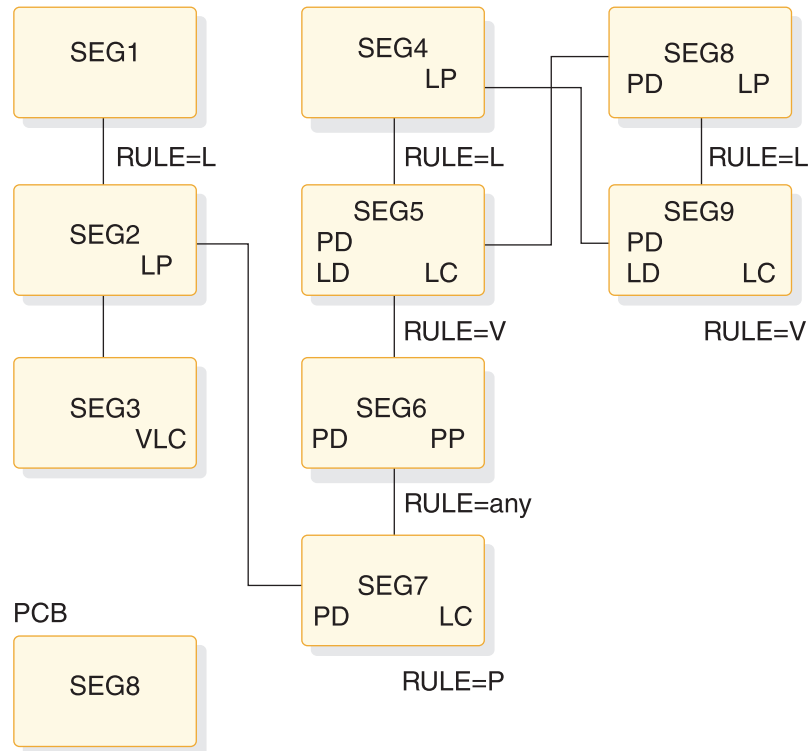
Figure 160. Example of violation of the physical delete rule

```
GHU 'SEG4'  STATUS='  '
DLET        STATUS='DX'
```

Figure 161. Example of violation of the physical delete rule: database calls

SEG7 (the logical child of SEG2) uses the physical delete rule and has not been logically deleted (the LD bit has not been set on). Therefore, the physical delete rule is violated. A 'DX' status code is returned to the application program, and no segments are deleted.

Related tasks:

"Avoiding abnormal termination" on page 318

Treating the physical delete rule as logical

If the delete routine determines that neither the segment specified in the DLET call nor any physical dependent of that segment in the physical structure uses the physical delete rule, any physical rule encountered later (logical deletion propagated to logical child or logical parent causing physical deletion—V rule—in another database) is treated as a logical delete rule.

The following figure and code show an example of treating the physical delete rule as logical.
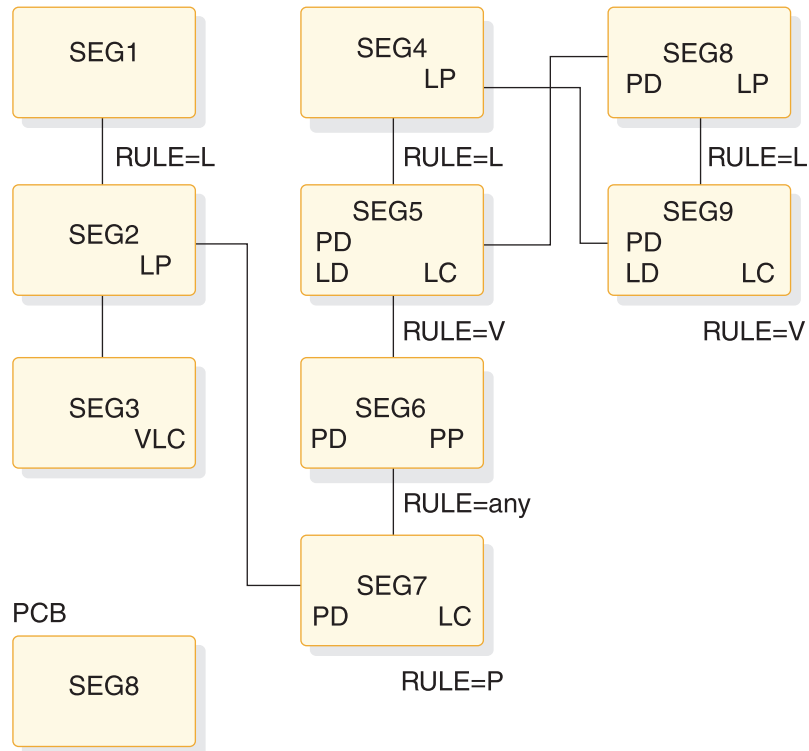
Figure 162. Example of treating the physical delete rule as logical

```
GHU 'SEG8'  STATUS='  '
DLET        STATUS='  '
```

Figure 163. Example of treating the physical delete rule as logical: database calls

SEG8 and SEG9 are both physically deleted, and SEG9 is logically deleted (V rule). SEG5 is physically and logically deleted because it is the physical pair to SEG9 (with physical pairing setting the LD bit in one set, the PID bit in the other, and vice versa). Physically deleting SEG5 causes propagation of the physical delete to SEG5's physical dependents; therefore, SEG6 and SEG7 are physically deleted.

Note that the physical deletion of SEG7 is prevented if the physical deletion started by issuing a DLET call for SEG4. But the physical rule of SEG7 is treated as logical in this case.

Inserting physically and logically deleted segments

When a segment is inserted, a replace operation is performed (space is reused), and existing dependents of the inserted segment remain if certain conditions are met.

A replace operation is performed (space is reused) and existing dependents of an inserted segment remain when a segment is inserted if:

*   The segment to be inserted already exists (same segment type and same key field value for both the physical and logical sequencing)
*   The delete bit is set on for that segment along the path of insertion

For HDAM and HIDAM databases, the logical twin chain is established as required, and existing dependents of the inserted segment remain.

For HISAM databases, if the root segment is physically and logically deleted before the insert is done, then the first logical record for that root in primary and secondary data set groups is reused. Remaining logical records on any OSAM chain are dropped.

Delete rules summary

The following list provides a summary of the delete rules.

The DLET Call

A DLET call issued against a concatenated segment (SOURCE=DATA/DATA, DATA/KEY, KEY/DATA) is a DLET call against the logical child only.

A DLET call against a logical child that has been accessed from its logical parent is a request that the logical child be logically deleted.

In all other cases, a DLET call issued against a segment is a request for that segment to be physically deleted.

Physical Deletion

A physically deleted segment cannot be accessed from its physical path, however, one exception exists: If one of the physical parents of the physically deleted segment is a logical child that can be accessed from its logical parent, then the physically deleted segment is accessible from that logical child. The physically deleted segments is accessible because the physical dependents of the logical child are variable intersection data.

Logical Deletion

By definition, a logically deleted logical child cannot be accessed from its logical parent. Unidirectional logical child segments are assumed to be logically deleted.

By definition, a logical parent is considered logically deleted when all its logical children are physically deleted and all its physical children that are part of a physically paired set are physically deleted.

Access Paths

Neither physical nor logical deletion of a segment prevents access to the segment from its physical or logical children, or from the segment to its physical or logical parents. A physically deleted root segment can be accessed only from its physical or logical children.

Propagation of Delete

In bidirectional physical pairing, physical deletion of one of the pair of logical children causes logical deletion of its paired segment. Likewise, logical deletion of one causes physical deletion of the other.

Physical deletion of a segment propagates logical deletion requests to its bidirectional logical children. Physical deletion of a segment propagates physical deletion requests to its physical children and to any index pointer segments for which it is the source segment.

Delete Rules

Further delete operations are governed by the following delete rules:

Logical Parent

When RULES=P is specified, if the segment is not already logically deleted, a DLET call against the segment or any of its physical parents results in a DX status code. No segments are deleted. If a request is made against the segment as a result of propagation across a logical relationship, then the P rule works like the L rule.

When RULES=L is specified, either physical or logical deletion can occur first, and neither causes the other to occur.

When RULES=V is specified, either physical or logical deletion can occur first. If the segment is logically deleted as the result of a DLET call, then it is physically deleted also.

Physical Parent of a Virtually Paired Logical Child
RULES=P, L, or V is meaningless.

When RULES=B is specified and all physical children that are virtually paired logical children are logically deleted, the physical parent segment is physically deleted.

Logical Child
When RULES=P is specified, if the segment is not already logically deleted, then a DLET call requesting physical deletion of the segment or any of its physical parents results in a DX status code. No segments are deleted. If a delete request is made against the segment as a result of propagation across a logical relationship or if the segment is one of a physically paired set, then the rule works like the L rule.

When RULES=L is specified, either physical or logical deletion can occur first, and neither causes the other to occur.

When RULES=V is specified, either physical or logical deletion can occur first and either causes the other to occur. If this rule is used on only one segment of a physically paired set, it works like the L rule.

Space Release
Depending on the database organization, DASD space can or cannot be reused when it is released. DASD space for a segment is released when the following conditions are met:

- Space has been released for all physical dependents of the segment.
- The segment is physically deleted.
- If the segment is a logical child or a logical parent, then it is physically and logically deleted.
- If the segment is a dependent of a logical child (variable intersection data) and the DLET call was issued against a physical parent of the logical child, then the logical child is both physically and logically deleted.
- If the segment is a primary index pointer segment, the space is released for its target segment.

Using the DLET call

The DLET call is a request to delete a path of segments, not a request to release the DASD space used by a segment.

Delete rules are needed when a segment is involved in a logical relationship, because that segment belongs to two paths: a physical and a logical path. The selection of the delete rules for the logical child and its logical and physical parent (or two logical parents if physical pairing is used) determines whether one or two DLET calls are necessary to delete the two access paths.

Physical and logical deletion
Physically deleting a segment prevents further access to that segment using its physical parents.

Physically deleting a segment also physically deletes its physical dependents, however one exception to this exists: If one of the physical parents of the physically deleted segment is a logical child that has been accessed from its logical parent, then the physically deleted segment is accessible from that logical child. The deleted segment is accessible from that logical child because the physical dependents of a logical child are variable intersection data.

Logically deleting a logical child prevents further access to the logical child using its logical parent. Unidirectional logical child segments are assumed to be logically deleted. A logical parent is considered logically deleted when all its logical children are physically deleted. For physically paired logical relationships, the physical child paired to the logical child must also be physically deleted before the logical parent is considered logically deleted.

Deleting concatenated segments

The following application program can be sensitive to either the concatenated segment—SOURCE=(DATA/DATA), (DATA/KEY), (KEY/DATA)—or the logical child, because it is the logical child that is either physically or logically deleted (depending on the path accessed) in all cases.

The concatenated segment relationships are shown in the following figure.

SOURCE=(DATA/DATA)   (DATA/KEY)   (KEY/DATA)
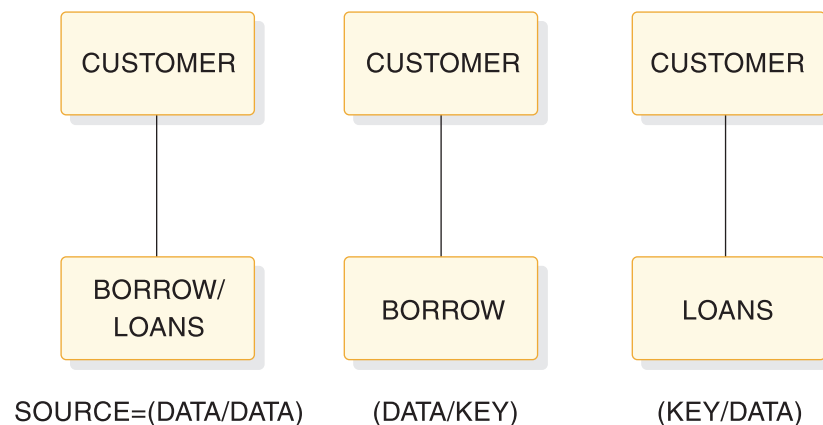
Figure 164. Concatenated segment relationships

Related reference:

DFSVSMxx member of the IMS PROCLIB data set (System Definition)

The third access path

In the figure below, three paths to the logical child segment SEG4 exist.

The three paths to the logical child segment SEG4 are:
- The physical path from its physical parent SEG3
- The logical path from its logical parent SEG7
- A third path from SEG4's physical dependents (SEG5 and SEG6) (because segment SEG6 is a logical parent accessible from its logical child SEG2)

These paths are called "full-duplex" paths, which means accessibility to segments in the paths is in two directions (up and down). Two delete bits that control access along the paths exist, but they are "half-duplex," which means they only block half of each respective path. No bit that blocks the third path exists. If SEG4 were both

physically and logically deleted (in which case the PD and LD bits in SEG4 would be set), SEG4 would still be accessible from the third path, and so would both of its parents.

Neither physical nor logical deletion prevents access to a segment from its physical or logical children. Logically deleting SEG4 prevents access to SEG4 from its logical parent SEG7, and it does not prevent access from SEG4 to SEG7. Physically deleting SEG4 prevents access to SEG4 from its physical parent SEG3, but it does not prevent access from SEG4 to SEG3.
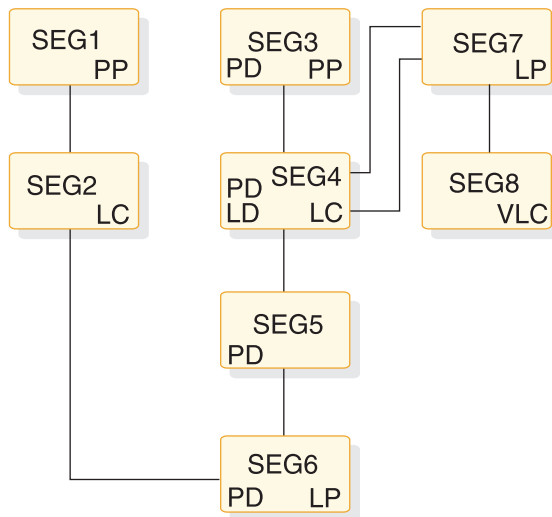
Figure 165. Third access path example

Related concepts:
"Possibility of abnormal termination" on page 317

Issuing the delete call

A DLET call can be issued against a segment defined in either a physical or logical DBD. The call can be issued against either a physical segment or a concatenated segment.

A DLET call issued against a concatenated segment requests deletion of the logical child in the path that is accessed. If a concatenated segment or a logical child is accessed from its logical parent, the DLET call requests logical deletion. In all other cases, a delete call requests physical deletion.

Physical deletion of a segment generates a request for logical deletion of all the segment's logical children and generates a request for physical deletion of all the segment's physical children. Physical deletion of a segment also generates a request to delete any index pointer segments for which the physically deleted segment is the source segment.

Delete sensitivity must be specified in the PCB for each segment against which a delete call can be issued. The call does not need to be specified for the physical dependents of those segments. Delete operations are not affected by KEY or DATA sensitivity as specified in either the PCB or logical DBD.

Status codes

The nonblank status codes that can be returned to an application program after a DLET call are as follows.

- DX—A delete rule was violated
- DA—The key was changed in the I/O area
- AM—The call function was not compatible with the processing option or segment sensitivity

DASD space release

The DLET call is not a request for release of DASD space. Depending on the database organization, DASD space can or cannot be reused when it is released.

DASD space for a segment is released when the following conditions are met:
- Space has been released for all physical dependents of the segment.
- The segment is physically deleted (PD bit is set or being set on).
- If the segment is a logical child or logical parent, then it must be physically and logically deleted (PD bit is set or being set on and LD bit is set or assumed set).
- If the segment is a dependent of a logical child (and is variable intersection data) and the DLET call was issued against a physical parent of the logical child, the logical child must be both physically and logically deleted.
- If the segment is a secondary index pointer segment, the space has been released for its target segment.

The segment delete byte

The delete byte is used by IMS to maintain the delete status of segments within a database.

The bits in the delete byte are only meaningful for logical child segments and their logical parents. For segments involved in a logical relationship, the PD and LD bits are set or assumed set as follows:
- If a segment is physically deleted (thereby preventing further access to it from its physical parent), then delete processing scans downward from the deleted segment through its dependents, turns upward, and either releases each segment's DASD space or sets the PD bit. HISAM is the one exception to this process. In HISAM, the delete bit is set in the segment specified by the DLET call and processing terminates.
- If the PD bit is set in a logical parent, the LD bit is set in all logical children that can be reached from that logical parent.
- When physical pairing is used, if the PD bit is set in one of a pair of logical children, the LD bit is set in its paired segment.
- When a virtually paired logical child is logically deleted (thereby preventing further access to it from its logical parent), the LD bit is set in the logical child.
- The LD bit is assumed set in all logical children in unidirectional logical relationships.
- If physical pairing is used, the LD bit is assumed set in a parent if all the paired segments that are physical children of the parent have the PD bit set on.

Bits in the delete byte

The meaning of the delete byte is determined by which bits within the byte are turned on.

This topic contains Diagnosis, Modification, and Tuning information.

The meaning of each bit in the delete byte, when turned on, is as follows:

Bit   Meaning When Delete Byte is Turned On

0	Segment has been marked for deletion. This bit is used for segments in a HISAM or secondary index database or segments in primary index.
1	Database record has been marked for deletion. This bit is used for segments in a HISAM or secondary index database or segments in a primary index.
2	Segment has been processed by the delete routine.
3	This bit is reserved.
4	Prefix and data portion of the segment are separated in storage. (The delete byte preceding the separated data portion of the segment has all bits turned on.)
5	Segment has been marked for deletion from a physical path. This bit is called the PD (physical delete) bit.
6	Segment has been marked for deletion from a logical path. This bit is called the LD (logical delete) bit.
7	Segment has been marked for removal from its logical twin chain. This bit should only be set on if bits 5 and 6 are also on).

Related concepts:

"Delete byte" on page 16

Bits in the prefix descriptor byte

The delete byte is also used for the root segment of a DEDB, only there it is called a prefix descriptor byte.

This topic contains Diagnosis, Modification, and Tuning information.

The meaning of each bit, when turned on, is as follows:

Bit	**Meaning When Root Segment Prefix Descriptor is Turned On**
0	Sequential dependent segment is defined.
1-3	These bits are reserved.
4-7	If the number of defined segments is 8 or less, bits 4 through 7 contain the highest defined segment code. Otherwise, the bits are set to 000.

Related concepts:

"Insert, delete, and replace rules for logical relationships" on page 287

Insert, delete, and replace rules summary

The following figure summarizes insert, delete, and replace rules by stating a desired result and then indicating the rule that can be used to obtain that result.
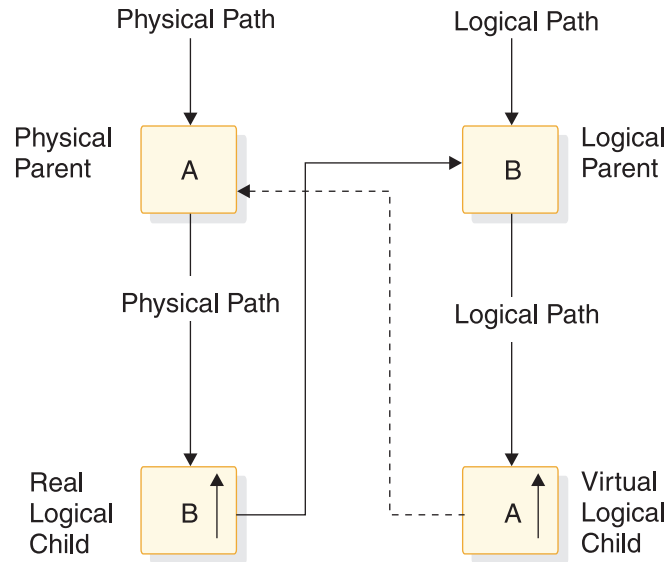
Figure 166. Insert, delete, and replace rules summary

The following table lists the insert, delete, and replace rules and how to specify them.

Table 60. Specifying insert, delete, and replace rules

Rule	RULES= specification
physical insert rule	RULES= (P,_,_)
logical insert rule	RULES= (L,_,_)
virtual insert rule	RULES= (V,_,_)
physical delete rule	RULES= (_,P,_)
logical delete rule	RULES= (_,L,_)
bidirectional virtual delete rule	RULES= (_,B,_)
virtual delete rule	RULES= (_,V,_)
physical replace rule	RULES= (_,_,P)
logical replace rule	RULES= (_,_,L)
virtual replace rule	RULES= (_,_,V)

Insert rules for physical parent segment A

The insert rules for physical parent (PP) segment A control the insert of PP A using the logical path to PP A.

The rules are as follows:
- To disallow the insert of PP A on its logical path, use the physical insert rule.
- To allow the insert of PP A on its logical path (concatenated with virtual logical child segment A), use either the logical or virtual rule.

  Where PP A is already present, a logical connection is established to the existing PP A segment. The existing PP A can either be replaced or remain unchanged:
  – If PP A is to remain unchanged by the insert call, use the logical insert rule.
  – If PP A is to be replaced by the insert call, use the virtual insert rule.

Delete rules for physical parent segment A

The delete rules for PP segment A control the deletion of PP A using the logical path to PP A.

The rules are as follows:
- To cause PP segment A to be deleted automatically when the last logical connection (through real logical child segment B to PP segment A) is broken, use the bidirectional virtual delete rule.
- The other delete rules for PP A are not meaningful.

Replace rules for physical parent segment A

The replace rules for PP segment A control the replacement of PP A using the logical path to PP A.

The rules are as follows:
- To disallow the replacement of PP A on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace PP A, use the physical replace rule.
- To disregard the replacement of PP A on its logical path, use the logical replace rule.
- To allow the replacement of PP A on its logical path, use the virtual replace rule.

Insert rules for logical parent segment B

The insert rules for logical parent (LP) segment B control the insert of LP B using the logical path to LP B.

Note: These rules are identical to the insert rules for PP segment A.

The rules are as follows:
- To disallow the insert of LP B on its logical path, use the physical insert rule.
- To allow the insert of LP B on its logical path (concatenated with virtual segment RLC B) use either the logical or virtual rule.

  Where LP B is already present, a logical connection is established to the existing LP B segment. The existing LP B can either be replaced or remain unchanged:
  – If LP B is to remain unchanged by the insert call, use the logical insert rule.
  – If LP B is to be replaced by the insert call, use the virtual insert rule.

Delete rules for logical parent segment B

The delete rules for segment LP B control the deletion of LP B on its physical path. A delete call for a concatenated segment is interpreted as a delete of the logical child only.

The rules are as follows:
- To ensure that LP B remains accessible until the last logical relationship path to that occurrence has been deleted, choose the physical delete rule. If an attempt to delete LP B is made while there are occurrences of real logical child (RLC) B pointing to LP B, a 'DX' status code is returned and no segment is deleted.
- To allow segment LP B to be deleted on its physical path, choose the logical delete rule. When LP B is deleted, it is no longer accessible on its physical path. It is still possible to access LP B from PP A through RLC B as long as RLC B exists.

- Use the virtual delete rule to physically delete LP B when it has been explicitly deleted by a delete call or implicitly deleted when all RLC Bs pointing to it have been physically deleted.

Replace rules for logical parent segment B

The replace rules for LP segment B control the replacement of LP B using the logical path to LP B.

Note: These rules are identical to the replace rules for PP segment A.

The rules are as follows:
- Use the physical replace rule to disallow the replacement of LP B on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace LP B.
- Use the logical replace rule to disregard the replacement of LP B on its logical path.
- Use the virtual replace rule to allow the replacement of LP B on its logical path.

Insert rules for real logical child segment B

The insert rules do not apply to a logical child.

Delete rules for real logical child segment B

The delete rules for RLC segment B apply to delete calls using its logical or physical path.

The rules are as follows:
- Use the physical delete rule to control the sequence in which RLC B is deleted on its logical and physical paths. The physical delete rule requires that it be logically deleted before it is physically deleted. A violation results in a 'DX' status code.
- Use the logical delete rule to allow either physical or logical deletes to be first.
- Use the virtual delete rule to use a single delete call from either the logical or physical path to both logically and physically delete RLC B.

Replace rules for real logical child segment B

The replace rules for LP B control the replacement of RLC B using the logical path to RLC B.

Note: These rules are identical to the replace rules for PP segment A.

The rules are as follows:
- Use the physical replace rule to disallow the replacement of RLC B on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace RLC B.
- To disregard an attempt to replace RLC B on its logical path, use the logical replace rule.
- To allow the replacement of RLC B on its logical path, use the virtual replace rule.

Logical relationships and HALDB databases

HALDB databases support logical relationships in the same manner that non-HALDB DL/I databases do, with a single exception: bidirectional logical relationships in a HALDB database must be implemented with physical pairing.

When you load a new partitioned database that contains logical relationships, the logical child segments cannot be loaded as part of the load step. IMS adds the logical children by normal update processing after the database has been loaded.

HALDB databases use an indirect list data set (ILDS) to maintain logical relationship pointers when logically related databases are reorganized.

Related concepts:

"The HALDB self-healing pointer process" on page 691

Performance considerations for logical relationships

If you are implementing a logical relationship, you make several choices that affect the resources needed to process logically related segments.

Logical parent pointers

The logical child segment on DASD has a pointer to its logical parent. You choose how this pointer is physically stored on external storage. Your choices are:

- Direct pointing (specified by coding POINTER=LPARNT in the SEGM statement for the logical child)
- Symbolic pointing (specified by coding the PHYSICAL operand for the PARENT= keyword in the SEGM statement for the logical child)
- Both direct and symbolic pointing

The advantages of direct pointers are:

- Because direct pointers are only 4 bytes long, they are usually shorter than symbolic pointers. Therefore, less DASD space is generally required to store direct pointers.
- Direct pointers usually give faster access to logical parent segments, except possibly HDAM or PHDAM logical parent segments, which are roots. Symbolic pointers require extra resources to search an index for a HIDAM database. Also, with symbolic pointers, DL/I has to navigate from the root to the logical parent if the logical parent is not a root segment.

The advantages of symbolic pointers are:

- Symbolic pointers are stored as part of the logical child segment on DASD. Having the symbolic key stored on DASD can save the resources required to format a logical child segment in the user's I/O area. Remember, the symbolic key always appears in the I/O area as part of the logical child. When retrieving a logical child, IMS has to construct the symbolic key if it is not stored on DASD.
- Logical parent databases can be reorganized without the logical child database having to be reorganized. This applies to unidirectional and bidirectional physically paired relationships (when symbolic pointing is used).

Symbolic pointing must be used:

- When pointing to a HISAM logical parent database
- If you need to sequence logical child segments (except virtual logical children) on any part of the symbolic key

KEY/DATA considerations

When you include a concatenated segment as part of a logical DBD, you control how the concatenated segment appears in the user's I/O area. You do this by specifying either KEY or DATA on the SOURCE= keyword of the SEGM statement for the concatenated segment. A concatenated segment consists of a logical child followed by a logical (or destination) parent. You specify KEY or DATA for both parts. For example, you can access a concatenated segment and ask to see the following segment parts in the I/O area:

- The logical child part only
- The logical (or destination) parent part only
- Both parts

By carefully choosing KEY or DATA, you can retrieve a concatenated segment with fewer processing and I/O resources. For example:

- Assume you have the unidirectional logical relationship shown in the following figure.
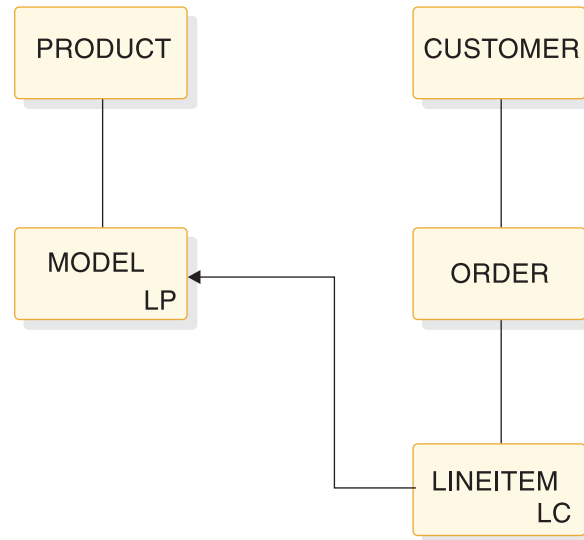
Figure 167. Example of a unidirectional logical relationship

- Assume you have the logical structure shown in the following figure.
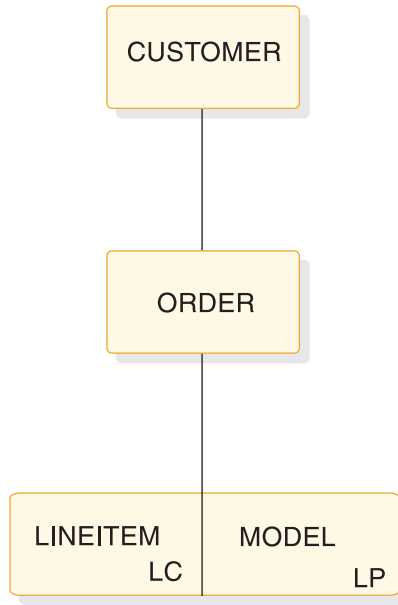
Figure 168. Example of a logical structure

- Finally, assume you only need to see the data for the LINEITEM part of the concatenated segment.

You can avoid the extra processing and I/O required to access the MODEL part of the concatenated segment if you:

- Code the SOURCE keyword of the concatenated segment's SEGM statement as:

  `SOURCE=(lcsegname,DATA,lcdbname),(lpsegname,KEY,lpdbname)`

- Store a symbolic logical parent pointer in LINEITEM. If you do not store the symbolic pointer, DL/I must access MODEL and PRODUCT to construct it.

To summarize, do not automatically choose DATA sensitivity for both the logical child and logical parent parts of a concatenated segment. If you do not need to see the logical parent part, code KEY sensitivity for the logical parent and store the symbolic logical parent pointer on DASD.

Sequencing logical twin chains

With virtual pairing, it is possible to sequence the real logical child on physical twin chains and the virtual logical child on logical twin chains. If possible, avoid operations requiring that you sequence logical twins. When a logical twin chain is followed, DL/I usually has to access multiple database records. Accessing multiple database records increases the resources required to process the call.

This problem of increased resource requirements to process calls is especially severe when you sequence the logical twin chain on all or part of the symbolic logical parent pointer. Because a virtual logical child is not stored, it is necessary to construct the symbolic logical parent pointer to determine if a virtual logical child satisfies the sequencing operation. DL/I must follow physical parent pointers to construct the symbolic pointers. This process takes place for each virtual logical child in the logical twin chain until the correct position is found for the sequencing operation.

Placement of the real logical child in a virtually paired relationship

In placing the real logical child in a virtually paired relationship, here are some considerations:

- If you need the logical child sequenced in only one of the logically related databases, put the real logical child in that database.
- If you must sequence the logical child in both logically related databases, put the real logical child in the database from which it is most often retrieved.
- Try to place the real logical child so logical twin chains are as short as possible. This placement decreases the number of database records that must be examined to follow a logical twin chain.

Note: You cannot store a real logical child in a HISAM database, because you cannot have logical child pointers (which are direct pointers) in a HISAM database.

Chapter 18. Creating secondary indexes

Secondary indexes are indexes that process a segment type in a sequence other than the one that is defined by the segment's key. A secondary index can also process a segment type based on a qualification in a dependent segment.

The following database types support secondary indexes:
- HISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM
- DEDB

Related concepts:

Chapter 12, "IMS catalog secondary index," on page 109

Related tasks:

"Altering IMS indexes" on page 510

"Dropping an index" on page 749

Related reference:

"XDFLD segment type format" on page 94

The purpose of secondary indexes

Secondary indexing provides a way to meet the different processing requirements of various applications. Secondary indexing allows you to have an index based on any field in the database, not just the key field in the root segment.

When you design your database records, you design them to meet the processing requirements of many applications. You decide what segments will be in a database record and what fields will be in a segment. You decide the order of segments in a database record and fields within a segment. You also decide which field in the root segment will be the key field, and whether the key field will be unique. All these decisions are based on what works best for all of the processing requirements of your applications. However, the choices you make might suit the processing requirements of some applications better than others.

Example: A database record in an educational database is shown in the following figure.
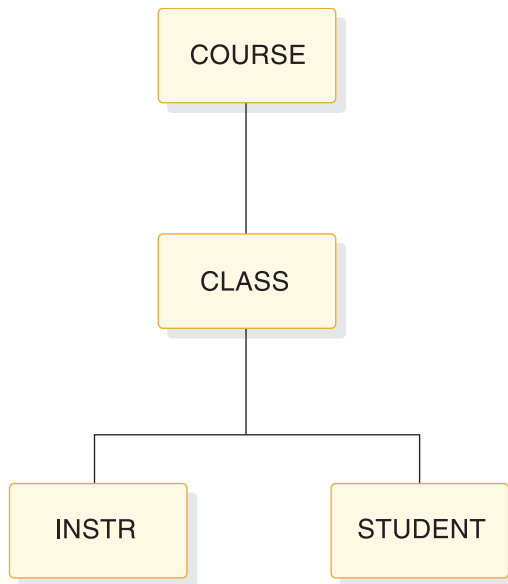
Figure 169. Database record in educational database

The following figure shows the root segment, COURSE, and the fields it contains. The course number field is a unique key field.

Figure 170. Example of a database record unique key field

You chose COURSE as the root and course number as a unique key field partly because most applications requested information based on course numbers. For these applications, access to the information needed from the database record is fast. For a few of your applications, however, the organization of the database record does not provide such fast access. One application, for example, would be to access the database by student name and then get a list of courses a student is taking. Given the order in which the database record is now organized, access to the courses a student is taking requires a sequential scan of the entire database. Each database record has to be checked for an occurrence of the STUDENT segment. When a database record for the specific student is found, then the COURSE segment has to be referenced to get the name of the course the student is taking. This type of access is relatively slow. In this situation, you can use a secondary index that has a set of pointer segments for each student to all COURSE segments for that student.

Another application would be to access COURSE segments by course name. In this situation, you can use a secondary index that allows access to the database in course name sequence (rather than by course number, which is the key field).

Characteristics of secondary indexes

Secondary indexes can be used with HISAM, HDAM, PHDAM, HIDAM, DEDB, and PHIDAM databases.

A secondary index is in its own a separate database and must use VSAM as its access method. Because a secondary index is in its own a database, it can be processed as a separate database.

Secondary indexes for full-function databases are invisible to the application program. When an application program needs to access a full-function database using the secondary index, this fact is communicated to IMS by coding the PROCSEQ= parameter in the PCB. If an application program needs to do processing using the regular processing sequence, PROCSEQ= is not coded. If the application program needs to do processing using both the regular processing sequence and the secondary index, the PSB for the application program must contain two PCBs, one with PROCSEQ= coded and one without.

Secondary indexes for Fast Path databases are also invisible to the application program. When a DEDB database needs to be accessed using its Fast Path secondary index, the PROCSEQD= parameter in the PCB is used to specify the name of the Fast Path secondary index database to use to access the primary DEDB database. The PROCSEQD= parameter has the same function as the full-function PROCSEQ= parameter. The PROCSEQD= parameter stands for PROCSEQ for DEDB databases.

When two PCBs are used, it enables an application program to use two paths into the database and two sequence fields. One path and sequence field is provided by the regular processing sequence, and one is provided by the secondary index. The secondary index gives an application program both an alternative way to enter the database and an alternative way to sequentially process database records.

If a PSB contains only a Fast Path secondary index PCB to access the Fast Path secondary index database as a separate database, the associated DEDB PCB must be included in the PSB. The minimal DEDB PCB requires a SENSEG statement for the root segment of the associated DEDB database.

A final characteristic of full-function secondary indexes is that there can be 32 secondary indexes for a segment type and a total of 1000 secondary indexes for a single full-function database.

Fast Path secondary index databases can be HISAM or SHISAM databases.

Fast Path secondary indexes have the following capabilities that are not available with full-function secondary indexes:

User data partitioning
> A single Fast Path secondary index can span multiple physical databases, each of which is considered a partition. Each partition can contain a range of keys. Index keys are assigned to a partition by a user partition selection exit routine. The index databases can be accessed individually or as one logical separate database.
>
> The partitions for a Fast Path secondary index are created using the DBD generation utility. When user partitioning for a Fast Path HISAM secondary index database or a Fast Path SHISAM secondary index database is requested:

- The PROCSEQD= parameter specifies the name of the first partition database in the user partition group.
- The PSELRTN= parameter specifies the DEDB partition selection exit routine that is used to determine the actual partition. The sample that is shipped with IMS is the Data Entry Database Partition Selection exit routine (DBFPSE00).

When the DEDB Partition Selection exit routine is not called, the PSELOPT=MULT | SNGL parameter can be used to indicate how many partition databases are processed before a GB status code is returned to an application to indicate the end of the database. PSELOPT= can be specified in both XDFLD statement or PCB with PROCSEQD= statement. If both are specified, the one in the PCB statement takes precedence.

Each Fast Path secondary index database can have a maximum of 101 user partition databases. Two or more user partition databases, separated by commas and enclosed in parentheses can be specified after the secondary index segment name in the NAME= parameter on the LCHILD statement in the primary DEDB database.

User data partitioning can be used with multiple secondary index segments. The following figure illustrates the concept of partitioning Fast Path secondary indexes.
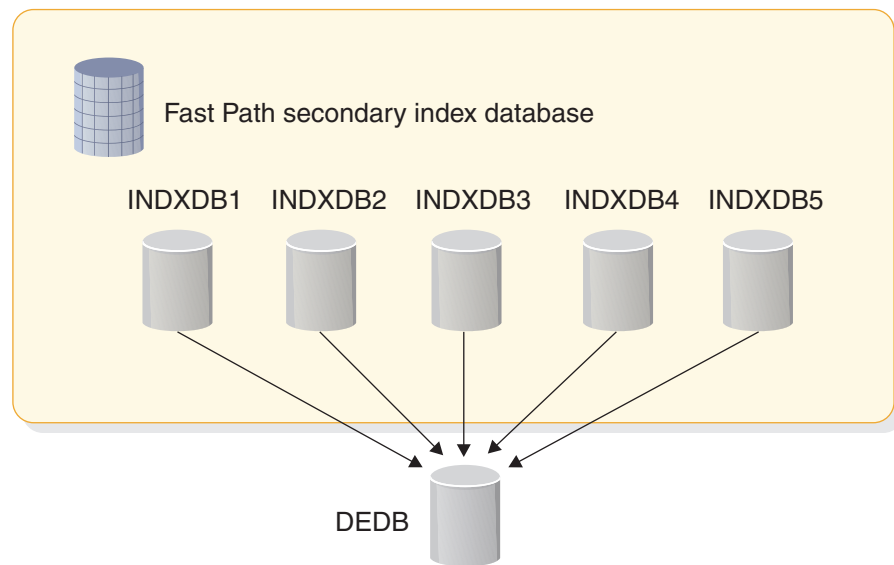
Multiple secondary index segments
> You can create multiple index entries from different fields in the same source segment.

> This is done by defining two or more LCHILD/XDFLD statement pairs under the SEGM statement of a target segment and specifying the MULTISEG=YES on the LCHILD statement.

A final characteristic of Fast Path secondary indexes is that there can be a maximum of 32 secondary indexes per segment and 255 secondary indexes per DEDB. Each multiple LCHILD/XDFLD statement pair counts towards the 32 secondary indexes per segment limit. When a Fast Path secondary index consists of partition databases, only the Fast Path secondary index database itself (not the partitions) is counted toward the 255 secondary indexes per DEDB limit.

Segments used for secondary indexes

To set up a secondary index, three types of segments must be defined to IMS: pointer, target, and source segments.

The following figure illustrates the segments used for a secondary index.
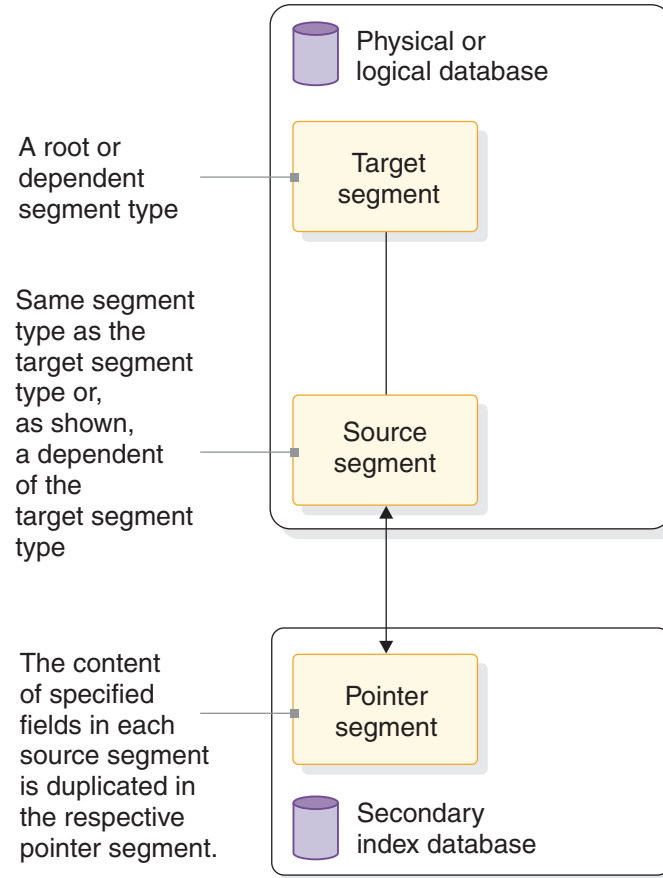
Figure 171. Segments used for secondary indexes

Pointer Segment

The pointer segment is contained in the secondary index database and is the only type of segment in the secondary index database. Its format is shown in the following figure.

Prefix		Data			
Delete byte	RBA of the segment to be retrieved	Key field	Optional fields		Symbolic pointer to the segment to be retrieved
Bytes 1	4		Varies		Varies
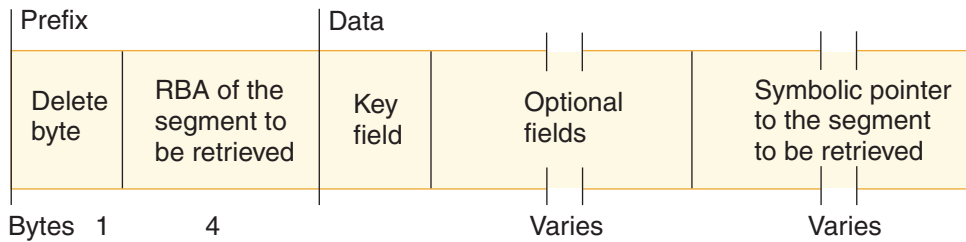
Figure 172. Format of pointer segments contained in the secondary index database

The first field in the prefix is the delete byte. The second field is the address of the segment the application program retrieves from the regular database. This field is not present if the secondary index uses symbolic pointing. Symbolic pointing is pointing to a segment using its concatenated key. HIDAM and HDAM can use symbolic pointing; however, HISAM must use symbolic pointing. Symbolic pointing is not supported for PHDAM and PHIDAM databases.

For a HALDB PSINDEX database, the segment prefix of pointer segments is slightly different. The "RBA of the segment to be retrieved field" is part of an Extended Pointer Set (EPS), which is longer than 4 bytes. Within the prefix, the EPS is followed by the key of the target's root.

For a DEDB database, the pointer segments must be symbolic.

Target Segment

The target segment is in the regular database, and it is the segment the application program needs to retrieve. A target segment is the segment to which the pointer segment points. The target segment can be at any one of the 15 levels in the database, and it is accessed directly using the RBA or symbolic pointer stored in the pointer segment. Physical parents of the target segment are not examined to retrieve the target segment, except in one special case discussed in "Symbolic pointer field" on page 349.

Source Segment

The source segment is also in the regular database. The source segment contains the field (or fields) that the pointer segment has as its key field. Data is copied from the source segment and put in the pointer segment's key field. The source and the target segment can be the same segment, or the source segment can be a dependent of the target segment. The optional fields are also copied from the source segment with one exception, which is discussed later in this topic.

Restriction: A DEDB database with sequential dependent (SDEP) segments can have a secondary index database. However, SDEP segments cannot be used as target or source segments for a Fast Path secondary index.

In the full-function education database shown in the following figure, three segments work together. The education database is a HIDAM database that uses RBAs rather than symbolic pointers. Suppose an application program needs to access the education database by student name and then list all courses the student is taking:

• The segment the application is trying to retrieve is the COURSE segment, because the segment contains the names of courses (COURSENM field). Therefore, COURSE is the target segment, and needs retrieval.

• In this example, the application program is going to use the student's name in its DL/I call to retrieve the COURSE segment. The DL/I call is qualified using

student name as its qualifier. The source segment contains the fields used to sequence the pointer segments in the secondary index. In this example, the pointer segments must be sequenced by student name. The STUDENT segment becomes the source segment. It is the fields in this segment that are copied into the data portion of the pointer segment as the key field.

- The call from the application program starts a search for a pointer segment with a key field that matches the student name. After the correct pointer segment in the index is found, it contains the address of the COURSE segment the application program is trying to retrieve.
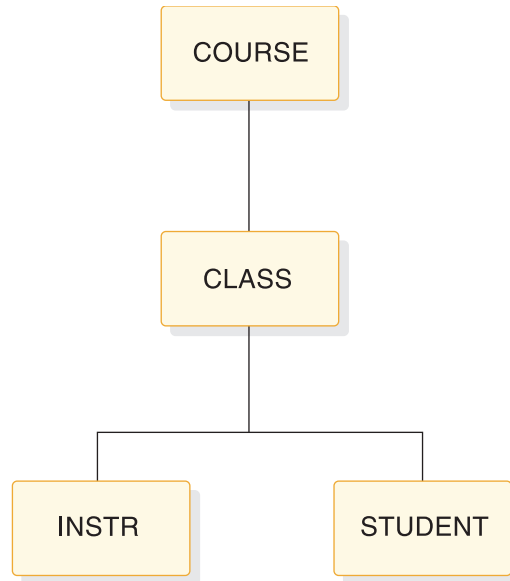
Figure 173. Education database record

The following figure shows how the pointer, target, and source segments work together.
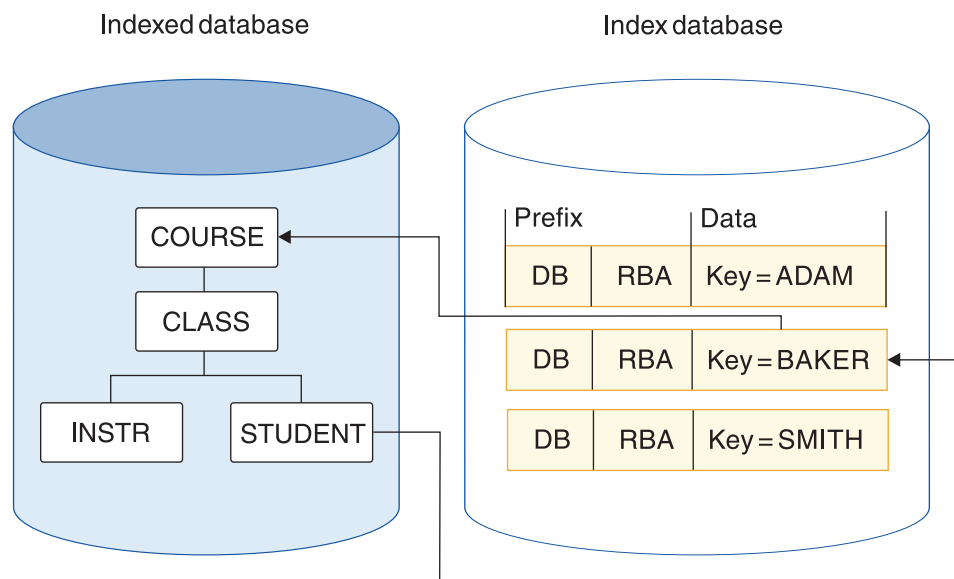
Figure 174. How a segment is accessed using a secondary index

The call that the application program issues when a secondary index is used is GU
COURSE (XNAME   = BAKER ...).

XNAME is from the NAME parameter in the XDFLD statement.

COURSE is the target segment that the application program is trying to retrieve.

STUDENT is the source segment containing one or more fields that the application program uses as a qualifier in its call and that the data portion of a pointer segment contains as a key.

The BAKER segment in the secondary index is the pointer segment, whose prefix contains the address of the segment to be retrieved and whose data fields contain the key the application program uses as a qualifier in its call.

How secondary indexes restructure the hierarchy of databases

When an application program accesses a database through a secondary index, the database records are processed in an alternative sequence.

These topics provide the details of how full-function database and DEDB database processing is impacted by secondary indexing

Related tasks:

"Altering IMS indexes" on page 510

How secondary indexes restructure the hierarchy of full-function databases

When the PROCSEQ= parameter in the PCB is coded to specify that the application program needs to do processing using the secondary index, the way in which the application program perceives the database record changes.

If the target segment is the root segment in the database record, the structure that the application program perceives does not differ from the one it can access using the regular processing sequence. However, if the target segment is not the root segment, the hierarchy in the database record is conceptually restructured. The following figures illustrate this concept.

The target segment (as shown in Figure 175 on page 343) is segment G. Target segment G becomes the root segment in the restructured hierarchy (as shown in Figure 176 on page 343). All dependents of the target segment (segments H, J, and I) remain dependents of the target segment. However, all segments on which the target is dependent (segments D and A) and their subordinates become dependents of the target and are put in the leftmost positions of the restructured hierarchy. Their position in the restructured hierarchy is the order of immediate dependency. D becomes an immediate dependent of G, and A becomes an immediate dependent of D.
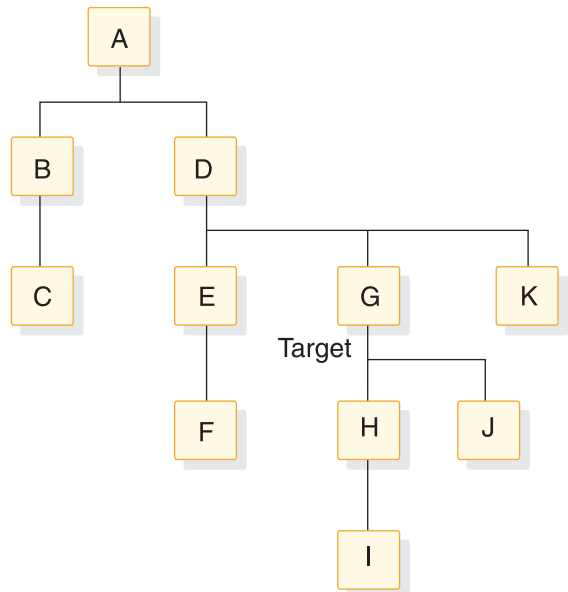
Figure 175. Physical database structure with target segment G
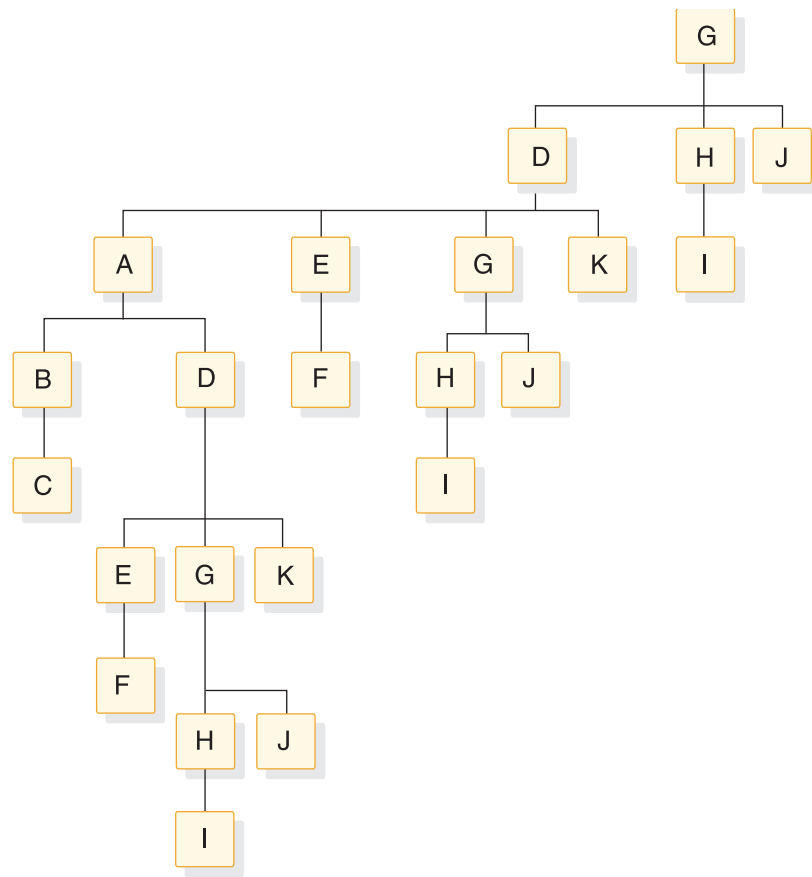
Figure 176. Secondary index structure indexed in secondary index on segment G

This new structure is called a *secondary data structure*. A processing restriction exists when using a secondary data structure, and the target segment and the segments on which it was dependent (its physical parents, segments D and A) cannot be inserted or deleted.

Secondary processing sequence

The restructuring of the hierarchy in the database record changes the way in which the application program accesses segments. The new sequence in which segments are accessed is called the *secondary processing sequence*. Figure 176 on page 343 shows how the application program perceives the database record.

If the same segment is referenced more than once (as shown in Figure 176 on page 343), you must use the DBDGEN utility to generate a logical DBD that assigns alternate names to the additional segment references. If you do not generate the logical DBD, the PSBGEN utility issues the message SEG150 for the duplicate SENSEG names.

How secondary indexes restructure the hierarchy of DEDB databases

When a primary DEDB database is accessed through its secondary index using the PCB with the PROCSEQD= parameter, the primary DEDB database is processed in an alternate sequence.

When the PROCSEQD= parameter in the PCB is coded, the way in which the application program perceives the database record changes. If the target segment is a root segment in the primary DEDB database, the inverted structure in the primary DEDB database that is using the secondary index is the same as the physical structure of the primary DEDB database. Subsequent unqualified DL/I GNP or GN calls after the GU of the target segment return segments in the primary DEDB database in the physical structure order.

If the target segment is not a root segment in the primary DEDB database, the hierarchy in the database record is conceptually restructured as an inverted structure. The DEDB inverted structure access is limited to a subset of segments. For DEDB inverted structure access, the target segment, its direct parent segments from the target segment to the root segment, and all its child segments from the target segments are accessible.

For a target segment that is not a root segment, the direct parent segments from the target segment to the root segment must have a unique key FIELD statement defined for each direct parent segment and the target segment. If there is no unique key in a FIELD statement for a direct parent segment, the DBDGEN utility terminates with an MNOTE 8 and message DGEN332.

Subsequent unqualified DL/I GNP or GN calls after the GU of the target segment return segments in the primary DEDB database in the conceptually reconstructed DEDB inverted structure order:

1. Transverse vertically up from the target segment through its direct parent segments to the root segment.
2. The target's direct parent segments to the root segment are returned.
3. Transverse vertically down from the target segment to all its child segments.
4. One or more of the target's child segments are returned if requested; specify SENSEG statements for the target's child segments.

The following two figures illustrate how the inverted structure is conceptually restructured when the PROCSEQD= parameter is coded in the PCB to indicate alternate sequence processing using the secondary index database for a primary DEDB database.

The following figure illustrates a physical structure of a database record with root A and target segment of G.
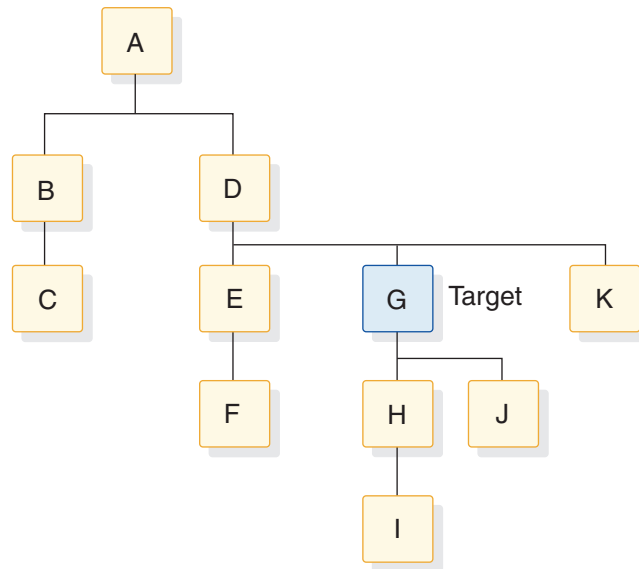
Figure 177. Physical database structure with target segment of G

The following figure illustrates a DEDB inverted database structure with a target segment of G.
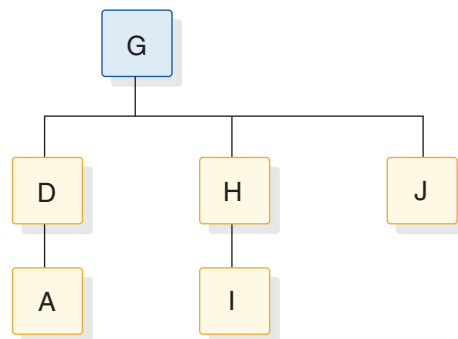
Figure 178. DEDB inverted database structure from target segment of G

The target segment (as shown in the figures) is segment G. Target segment G becomes the root segment in the restructured hierarchy. All dependents of the target segment (segments H, I, J) remain dependents of the target segment. However, all segments on which the target is a direct dependent (segments D and A) are put in the leftmost positions of the restructured hierarchy. Their position in the restructured hierarchy is the order of immediate dependency. D becomes an immediate dependent of G, and A becomes an immediate dependent of D.

The new inverted structure is called a *DEDB secondary data structure*. A processing restriction exists when using a secondary data structure, and the target segment and the segments on which it was dependent (its physical parents, segments D and A) cannot be inserted or deleted.

The following steps describe the DL/I call sequence using the PCB with the PROCSEQD= parameter:

1. A qualified GU call returns target segment G.
2. A subsequent unqualified GNP or GN calls return segments D, A, H, I, J.
3. For unqualified GNP calls, a GE status code is returned after segment J.
4. For unqualified GN calls, a GN call returns the next nth segment after segment G in the secondary index database.
5. For unqualified GN calls, repeat steps 2 and 4. A GB status code is returned when there are no more segments in the secondary index database.

Related tasks:

"Adding a secondary index to a DEDB" on page 748

How a secondary index is stored

Secondary index databases contain root segments only.

They are stored in a single VSAM KSDS if the key in the pointer segment is unique. If keys are not unique, an additional data set must be used (an ESDS) to store segments containing duplicate keys. (KSDS data sets do not allow duplicate keys.) Duplicate keys exist when, for example, a secondary index is used to retrieve courses based on student name. As shown in the following figure, several source segments could exist for each student.

Figure 179. Examples of source segments for each student

Each pointer segment in a secondary index is stored in one logical record. A logical record containing a pointer segment is shown in the following figure.
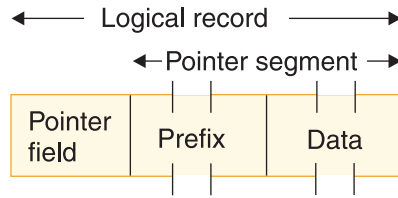
Figure 180. Example of a logical record containing a pointer segment

A HALDB secondary index record is shown in the following figure.
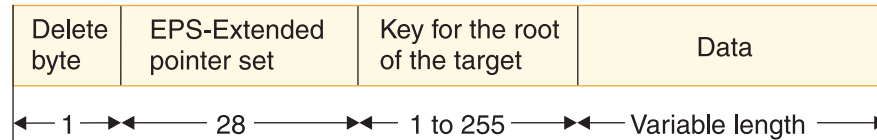
Figure 181. Secondary index entry for HALDB

The format of the logical record is the same in both a KSDS and ESDS data set. The pointer field at the beginning of the logical record exists only when the key in the data portion of the segment is not unique. If keys are not unique, some pointer segments will contain duplicate keys. These pointer segments must be chained together, and this is done using the pointer field at the beginning of the logical record.

Pointer segments containing duplicate keys are stored in the ESDS in LIFO (last in, first out) sequence. When the first duplicate key segment is inserted, it is written to the ESDS, and the KSDS logical record containing the segment it is a duplicate of points to it. When the second duplicate is inserted, it is inserted into the ESDS in the next available location. The KSDS logical record is updated to point to the second duplicate. The effect of inserting duplicate pointer segments into the ESDS in LIFO sequence is that the original pointer segment (the one in the KSDS) is retrieved last. This retrieval sequence should not be a problem, because duplicates, by definition, have no special sequence.

Format and use of fields in a pointer segment

Like all segments, the pointer segment has a prefix and data portion.

This topic contains Diagnosis, Modification, and Tuning information.

The prefix portion has a delete byte, and when direct rather than symbolic pointing is used, it has the address of the target segment (4 bytes). The data portion has a series of fields, and some of them are optional. All fields in the data portion of a pointer segment contain data taken from the source segment (with the exception of user data). These fields are the constant field (optional), the search field, the subsequence field (optional), the duplicate data field (optional), the concatenated key field (optional except for HISAM), and then the data (optional).

The following figure shows the fields in a pointer segment.
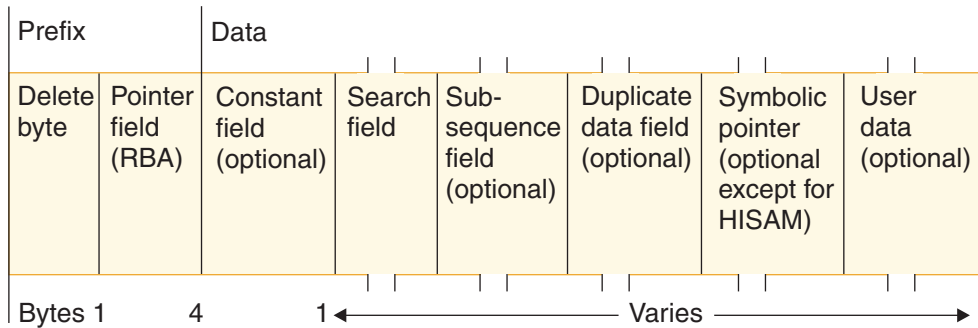
Figure 182. Fields of a pointer segment in a secondary index

Delete byte

The delete byte is used by IMS to determine whether a segment has been deleted from the database.

Pointer field

This field, when present, contains the RBA of the target segment. The pointer field exists when direct pointing is specified for an index pointing to an HD database. Direct pointing is simply pointing to a segment using its actual address.

The other type of pointing that can be specified is symbolic pointing. Symbolic pointing, which is explained under "Symbolic pointer field," can be used to point to HD databases and must be used to point to HISAM databases. If symbolic pointing is used, this field does not exist.

Constant field

This field, when present, contains a 1-byte constant. The constant is used when more than one index is put in an index database. The constant identifies all pointer segments for a specific index in the shared index database. The value in the constant field becomes part of the key.

The Constant field is not present in Fast Path secondary index pointer segments.

Search field

The data in the search field is the key of the pointer segment. All data in the search field comes from data in the source segment. As many as five fields from the source segment can be put in the search field. These fields do not need to be contiguous fields in the source segment. When the fields are stored in the pointer segment, they can be stored in any order. When stored, the fields are concatenated. The data in the search field (the key) can be unique or non-unique.

IMS automatically maintains the search field in the pointer segment whenever a source segment is modified.

Subsequence field

The subsequence field, like the search field, contains from one to five fields of data from the source segment. Subsequence fields are optional, and can be used if you have non-unique keys. The subsequence field can make non-unique keys unique. Making non-unique keys unique is desirable because of the many disadvantages of

non-unique keys. For example, non-unique keys require you to use an additional data set, an ESDS, to store all index segments with duplicate keys. An ESDS requires additional space. More important, the search for specific occurrences of duplicates requires additional I/O operations that can decrease performance.

When a subsequence field is used, the subsequence data is concatenated with the data in the search field. These concatenated fields become the key of the pointer segment. If properly chosen, the concatenated fields form a unique key. It is not always possible to form a unique key using source data in the subsequence field. Therefore, you can use system related fields to form unique keys.

One important thing to note about using subsequence fields is that if you use them, the way in which an SSA is coded does not need to change. The SSA can still specify what is in the search field, but it cannot specify what is in the search plus the subsequence field. Subsequence fields are not seen by the application program unless it is processing the secondary index as a separate database.

Up to five fields from the source segment can be put in the subsequence field. These fields do not need to be contiguous fields in the source segment. When the fields are stored in the pointer segment, they can be stored in any order. When stored, they are concatenated.

IMS automatically maintains the subsequence field in the pointer segment whenever a source segment is modified.

Duplicate data field

The duplicate data field, like the search field, contains from one to five fields of data from the source segment. Duplicate data fields are optional. Use duplicate data fields when you have applications that process the secondary index as a separate database. Like the subsequence field, the duplicate data field is not seen by an application program unless it is processing the secondary index as a separate database.

As many as five fields from the source segment can be put in the duplicate data field. These fields do not need to be contiguous fields in the source segment. When the fields are stored in the pointer segment, they can be stored in any order. When stored, they are concatenated.

IMS automatically maintains the duplicate data field in the pointer segment whenever a source segment is modified.

Symbolic pointer field

This field, when present, contains the concatenated key of the target segment. This field exists when the pointer segment points to the target segment symbolically, rather than directly. Direct pointing is simply pointing to a segment using its actual address. Symbolic pointing is pointing to a segment by a means other than its actual address. In a secondary index, the concatenated key of the target segment is used as a symbolic pointer.

Segments in an HDAM or a HIDAM database being accessed using a secondary index can be accessed using a symbolic pointer. Segments in a HISAM database must be accessed using a symbolic pointer because segments in a HISAM database can "move around," and the maintenance of direct-address pointers could be a large task. One of the implications of using symbolic pointers is that the physical

parents of the target segment must be accessed to get to the target segment. Because of this extra access, retrieval of target segments using symbolic pointing is not as fast as retrieval using direct pointing. Also, symbolic pointers generally require more space in the pointer segment. When symbolic pointers are used, the pointer field (4 bytes long) in the prefix is not present, but the fully concatenated key of the target segment is generally more than 4 bytes long.

IMS automatically generates the concatenated key field when symbolic pointing is specified.

One situation exists in which symbolic pointing is specified and IMS does not automatically generate the concatenated key field. This situation is caused by specifying the system-related field /CK as a subsequence or duplicate data field in such a way that the concatenated key is fully contained. In this situation, the symbolic pointer portion of either the subsequence field or the duplicate data field is used.

User data in pointer segments

You can include any user data in the data portion of a pointer segment by specifying a segment length long enough to hold it. You need user data when applications process the secondary index as a separate database. Like data in the subsequence and duplicate data fields, user data is never seen by an application program unless it is processing the secondary index as a separate database.

You must initially load user data. You must also maintain it. During reorganization of a database that uses secondary indexes, the secondary index database is rebuilt by IMS. During this process, all user data in the pointer segment is lost.

Related concepts:

"Sharing secondary index databases" on page 359

Related tasks:

"Processing a secondary index as a separate database" on page 358

Fields in the HISAM secondary index pointer

Fields in a HISAM secondary index database support both unique and non-unique keys.

This topic contains Diagnosis, Modification, and Tuning information.

A HISAM secondary index database supports both unique and non-unique keys. Non-unique keys are stored and retrieved in last-in first-out (LIFO) order. Both KSDS and ESDS data sets are required when the secondary index database supports non-unique keys. The first inserted non-unique key is stored in the KSDS data set and the remaining non-unique keys are stored in the ESDS data set in LIFO order.

A HISAM secondary index database supports subsequence field, duplicate data field, user data field, and the /CK operand. The subsequence field, and the /CK operand can be used to make the secondary index key unique.

A HISAM secondary index database contains fixed-length segments, provides data partitioning using a partition selection exit routine, and supports the Segment edit/compression exit routine (DFSCMPX0).

Fields in the HISAM secondary index pointer with a unique key

The following figure shows the fields in a HISAM secondary index pointer with a unique key.

Delete byte	Search field	Subsequence	Duplicate data	Symbolic pointer	User data
		(optional)	(optional)		(optional)

Bytes1 |◄----------------------------- Variable ------------------------------------►|

|◄----►|◄----------------------------- Data -------------------------------------►|
Prefix

|◄---------------------------------- Pointer segment ----------------------------►|

|◄---------------------------------- Logical record -----------------------------►|
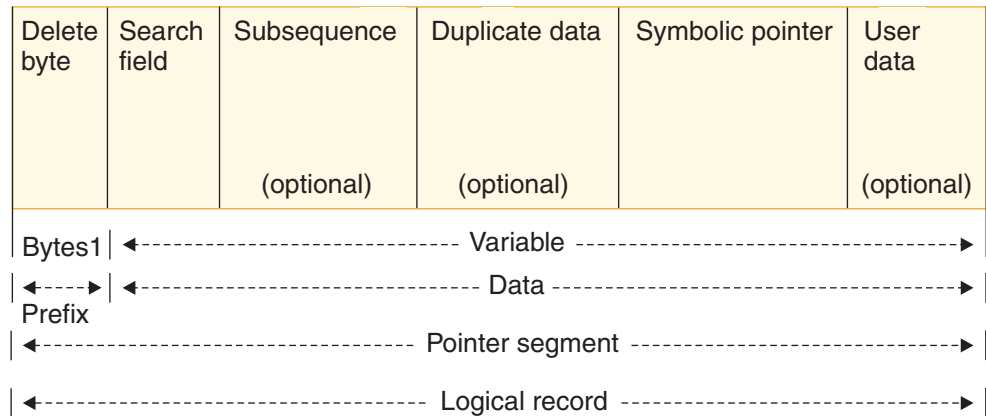
Figure 183. Example of a HISAM secondary index pointer with a unique key

Logical record
    A secondary index pointer segment is stored in a logical record.

Pointer segment
    A secondary index pointer segment contains prefix and data fields.

Prefix Delete byte: one byte

Data fields

Search field
    Variable-length bytes, made up of up to 5 fields from the source.

Subsequence field
    Variable-length bytes, made up of up to 5 fields from the source or IMS-generated values (optional). It is used to make the secondary index key unique. It can be used to order segments in a secondary index database. The search field and the subsequence field together make up the key of the secondary index.

Duplicate data field
    Variable-length bytes, made up of up to 5 fields from the source (optional). It is only used when processing the secondary index as a database.

Symbolic pointer field
    Variable-length bytes. It is the concatenated key to the target.

User data
    Variable-length bytes, made up of any user data fields (optional). It is only used when processing the secondary index as a database.

Fields in the HISAM secondary index pointer with a non-unique key

Duplicate key pointer	Delete byte	Search	Subsequence	Duplicate data	Symbolic pointer	User data
			(optional)	(optional)		(optional)

Bytes 4   1 |◄---------------- Variable -----------------------------------►|

|◄----►| |◄------------------ Data --------------------------------------►|
 Prefix

|◄--------------------- Pointer segment -----------------------------►|

|◄----------------------------------- Logical record ----------------------------------►|
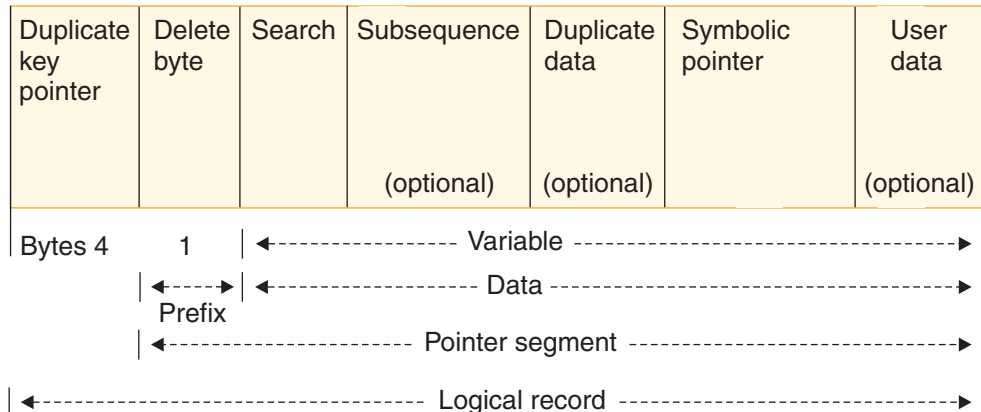
Figure 184. Example of a HISAM secondary index pointer with a non-unique key

Logical record
> A secondary index pointer segment is stored in a logical record.

> **Duplicate key pointer**
>> Four byte pointer for HISAM secondary index when the secondary index key is non-unique. If the keys are not unique, some pointer segments will contain duplicate keys. These pointer segments must be chained together, and this is done using the duplicate key pointer field at the beginning of the logical record. The duplicate key pointer exists when the OVFLW= operand on the DATASET statement is defined.

Pointer segment
> A secondary index pointer segment contains prefix and data fields.

Prefix   Delete byte: one byte

Data fields

> **Search field**
>> Variable-length bytes, made up of up to 5 fields from the source.

> **Subsequence field**
>> Variable-length bytes, made up of up to 5 fields from the source or IMS-generated values (optional). It is used to make the secondary index key unique. It can be used to order segments in a secondary index database. The search field and the subsequence field together make up the key of the secondary index.

> **Duplicate data field**
>> Variable-length bytes, made up of up to 5 fields from the source (optional). It is only used when processing the secondary index as a database.

> **Symbolic pointer field**
>> Variable-length bytes. It is the concatenated key to the target.

> **User data field**
>> Variable-length bytes, made up of any user data fields (optional). It is only used when processing the secondary index as a database.

Related tasks:

Fields in the SHISAM secondary index pointer

SHISAM secondary index databases are not required to register to DBRC. A SHISAM secondary index database supports only unique keys because a SHISAM database supports only KSDSs, not ESDSs.

A SHISAM secondary index database supports the subsequence field, the duplicate data field, the user data field, and the /CK operand. The subsequence field and the /CK operand can be used to make the secondary index key unique.

A SHISAM secondary index database contains fixed-length segments, provides data partitioning using the DEDB Partition Selection exit routine, and supports the Segment edit/compression exit routine.
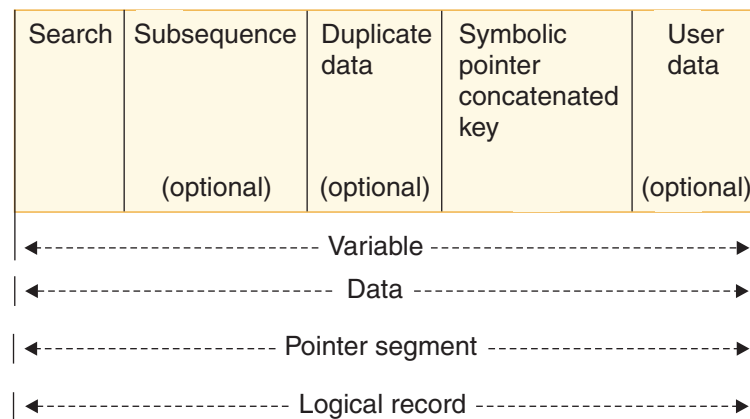
Figure 185. Example of a SHISAM secondary index pointer

Logical record
> A secondary index pointer segment is stored in a logical record.

Pointer segment
> A secondary index pointer segment contains prefix and data fields. Because SHISAM secondary index segments do not have a prefix field, a secondary index pointer segment contains only data fields.

Data fields

> **Search field**
>> Variable-length bytes, including up to 5 fields from the source. The search field is the key of the secondary index.

> **Subsequence field (optional)**
>> Variable-length bytes, including up to 5 fields from the source or IMS-generated values. The subsequence field makes the secondary index key unique, and can be used to order segments in a secondary index database. Subsequence length is used to determine the concatenated key length.

> **Duplicate data field (optional)**
>> Variable-length bytes, including up to 5 fields from the source. The field is used only when processing the secondary index as a database.

Symbolic pointer concatenated key field
> Variable-length bytes. The field is the concatenated key to the target.

User data field (optional)
> Variable-length bytes, including any user data fields. The field is used only when processing the secondary index as a database.

Related tasks:

"Processing a secondary index as a separate database" on page 358

Making keys unique using system related fields

If creating unique keys by keeping additional information from the source segment in the subsequence field of the pointer segment does not work for you, there are two other ways to force unique keys, both of which use an operand in the FIELD statement of the source segment in the DBD.

The FIELD statement defines fields within a segment type.

Using the /SX operand

For HD databases, you can code a FIELD statement with a NAME field that starts with /SX. The /SX can be followed by any additional characters (up to five) that you need. When you use this operand, the system generates (during segment insertion) the RBA, or an 8-byte ILK for PHDAM or PHIDAM, of the source segment. The system also puts the RBA or ILK in the subsequent field in the pointer segment, thus ensuring that the key is unique. The FIELD statement in which /SX is coded is the FIELD statement defining fields in the source segment. The /SX value is not, however, put in the source segment. It is put in the pointer segment.

When you use the /SX operand, the XDFLD statement in the DBD must also specify /SX (plus any of the additional characters added to the /SX operand). The XDFLD statement, among other things, identifies fields from the source segment that are to be put in the pointer segment. The /SX operand is specified in the SUBSEQ= operand in the XDFLD statement.

Using the /CK operand

The other way to force unique keys is to code a FIELD statement with a NAME parameter that starts with /CK. When used as a subsequence field, /CK ensures unique keys for pointer segments. You can use this operand for HISAM, HDAM, PHDAM, HIDAM, or PHIDAM databases. The /CK can be followed by up to five additional characters. The /CK operand works like the /SX operand except that the concatenated key, rather than the RBA, of the source segment is used. Another difference is that the concatenated key is put in the subsequence or duplicate data field in the pointer segment. Where the concatenated key is put depends on where you specify the /CK.

When using /CK, you can use a portion of the concatenated key of the source segment (if some portion will make the key unique) or all of the concatenated key. You use the BYTES= and START= operands in the FIELD statement to specify what you need.

For example, suppose you are using the database record shown in the following figure.
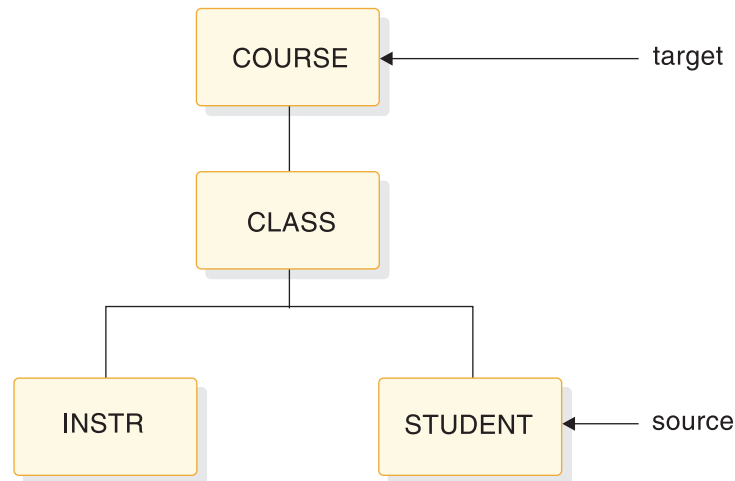
Figure 186. Database record showing the source and target for secondary indexes

The concatenated key of the STUDENT segment is shown in the following figure.

Figure 187. Concatenated key of the STUDENT segment

If you specify on the FIELD statement whose name begins with /CK BYTES=21, START=1, the entire concatenated key of the source segment will be put in the pointer segment. If you specify BYTES=6, START=16, only the last six bytes of the concatenated key (CLASSNO and SEQ) will be put in the pointer segment. The BYTES= operand tells the system how many bytes are to be taken from the concatenated key of the source segment in the PCB key feedback area. The START= operand tells the system the beginning position (relative to the beginning of the concatenated key) of the information that needs to be taken. As with the /SX operand, the XDFLD statement in the DBD must also specify /CK.

To summarize: /SX and /CK fields can be included on the SUBSEQ= parameter of the XDFLD statement to make key fields unique. Making key fields unique avoids the overhead of using an ESDS to hold duplicate keys. The /CK field can also be specified on the DDATA= parameter of the XDFLD statement but the field will not become part of the key field.

When making keys unique, unique sequence fields must be defined in the target segment type, if symbolic pointing is used. Also, unique sequence fields must be defined in all segment types on which the target segment type is dependent (in the physical rather than restructured hierarchy in the database).

How sparse indexing suppresses index entries

When a source segment is loaded, inserted, or replaced in the database, DL/I automatically creates or maintains the pointer segment in the index. This happens automatically unless you have specified that you do not need certain pointer segments built.

For example, suppose you have a secondary index for an education database. STUDENT is the source segment, and COURSE is the target segment. You might need to create pointer segments for students only if they are associated with a certain customer number. This could be done using sparse indexing, a performance enhancement of secondary indexing.

HALDB partitioned secondary indexes (PSINDEXes) support sparse indexing; however, the sparse indexing of the segments in a PSINDEX does not reduce the number of indirect list entries (ILEs) in the indirect list data set (ILDS) that manages pointers between the indexed database and the PSINDEX. The number of ILEs in an ILDS matches the number of target segments in the indexed database rather than the number of segments in the PSINDEX.

Advantages of sparse indexing

Sparse indexing allows you to specify the conditions under which a pointer segment is suppressed, not generated, and put in the index database. Sparse indexing has two advantages. The primary one is that it reduces the size of the index, saving space and decreasing maintenance of the index. By decreasing the size of the index, performance is improved. The second advantage is that you do not need to generate unnecessary index entries.

Suppressing index maintenance for BMP regions

When a DEDB database has secondary index defined, IMS automatically performs index maintenance when the source statement is inserted, updated, or deleted. The index suppression option provides the capability for updating a DEDB database with one or more secondary index defined without index maintenance. If an application has many updates to the primary DEDB database that would result in significant index maintenance to its associated secondary index database, you can suppress the index maintenance for the application. Then synchronize your primary DEDB database and its secondary index databases at a later time using an in-house application or vendor tool product.

To suppress index maintenance, specify the //DFSCTL DD statement in the JCL of the IMS BMP region.

```
 //DFSCTL DD *
SETI PSB=psbname
```

The SETI PSB=psbname parameter suppresses index maintenance for any DEDB database with secondary index defined for the BMP application for PSB of psbname.

If *psbname* in the PSB=*psbname* parameter in the SETI statement does not match the PSB name for the BMP application, or the PSB= parameter is not specified in the SETI statement, message DFS0510E is issued and the application is terminated with an ABENDU1060. You will need to correct the SETI statement and rerun the BMP application.

Specifying a sparse index

Sparse indexing can be specified in two ways.

First, you can code a value in the NULLVAL= operand on the XDFLD statement in the DBD that equals the condition under which you do not need a pointer segment put in the index. You can put BLANK, ZERO, or any 1-byte value (for example, X'10', C'Z', 5, or B'00101101') in the NULLVAL= operand.

- BLANK is the same as C ' ' or X'40'
- ZERO is the same as X'00' but *not* C'0'

When using the NULLVAL= operand, a pointer segment is suppressed if every byte of the source field has the value you used in the operand.

Second, if the values you are allowed to code in the NULLVAL= operand do not work for you, you can create an index maintenance exit routine that determines the condition under which you do not need a pointer segment put in the index. If you create your own index maintenance exit routine, you code its name in the EXTRTN= operand on the XDFLD statement in the DBD. You can only have one index maintenance exit routine for each secondary index. This exit routine, however, can be a general purpose one that is used by more than one secondary index.

The exit routine must be consistent in determining whether a particular pointer segment needs to be put in the index. The exit routine cannot examine the same pointer segment at two different times but only mark it for suppression once. Also, user data cannot be used by your exit routine to determine whether a pointer segment is to be put in the index. When a pointer segment needs to be inserted into the index, your exit routine only sees the actual pointer segment just before insertion. When a pointer segment is being replaced or deleted, only a prototype of the pointer segment is seen by your exit routine. The prototype contains the contents of the constant, search, subsequence, and duplicate data fields, plus the symbolic pointer if there is one.

The information needed to code a secondary index maintenance exit routine is in *IMS Version 15 Exit Routines*.

How the secondary index is maintained

When a source segment is inserted, deleted, or replaced in the database, IMS keeps the index current regardless whether the application program performing the update uses the secondary index.

The way in which IMS maintains the index depends on the operation being performed. Regardless of the operation, IMS always begins index maintenance by building a pointer segment from information in the source segment that is being inserted, deleted, or replaced. (This pointer segment is built but not yet put in the secondary index database.)

Inserting a source segment

When a source segment is inserted, DL/I determines whether the pointer segment needs to be suppressed. If the pointer segment needs to be suppressed, it is not put in the secondary index. If the pointer segment does not need to be suppressed, it is put in the secondary index.

Deleting a source segment

When a source segment is deleted, IMS determines whether the pointer segment is one that was suppressed. If so, IMS does not do any index maintenance. If the segment is one that was suppressed, there should not be a corresponding pointer segment in the index to delete. If the pointer segment is not one that was suppressed, IMS finds the matching pointer segment in the index and deletes it. Unless the segment contains a pointer to the ESDS data set, which can occur with a non-unique secondary index, the logical record containing the deleted pointer segment in a KSDS data set is erased.

Replacing a source segment

When a source segment is replaced, the pointer segment in the index might or might not be affected. The pointer segment in the index might need to be replaced or deleted, or the pointer segment might need no changes. After replacement or deletion, a new pointer segment is inserted. IMS determines what needs to be done by comparing the pointer segment it built (the new one) with the matching pointer segment in the secondary index (the old one).

- If both the new and the old pointer segments need to be suppressed, IMS does not do anything (no pointer segment exists in the index).
- If the new pointer segment needs to be suppressed but the old one does not, then the old pointer segment is deleted from the index.
- If the new pointer segment does not need to be suppressed but the old pointer segment is suppressed, then the new pointer segment is inserted into the secondary index.
- If neither the new or the old segment needs to be suppressed and:
  - If there is no change to the old pointer segment, IMS does not do anything.
  - If the non-key data portion in the new pointer segment is different from the old one, the old pointer segment is replaced. User data in the index pointer segment is preserved when the pointer segment is replaced.
  - If the key portion in the new pointer segment is different from the old one, the old pointer segment is deleted and the new pointer segment is inserted. User data is *not* preserved when the index pointer segment is deleted and a new one inserted.

If you reorganize your secondary index and it contains non-unique keys, the resulting pointer segment order can be unpredictable.

Processing a secondary index as a separate database

Because they are actual databases, secondary indexes can be processed independently.

A number of reasons exist why an application program might process a secondary index as an independent database. For example, an application program can use the secondary index to retrieve a small piece of data from the database. If you put this piece of data in the pointer segment, the application program can retrieve it without an I/O operation to the regular database. You could put the piece of data in the duplicate data field in the pointer segment if the data was in the source segment. Otherwise, you must carry the data as user data in the pointer segment. (If you carry the data as user data, it is lost when the primary database is reorganized and the secondary index is recreated.)

Another reason for processing a secondary index as a separate database is to maintain it. You could, for example, scan the subsequence or duplicate data fields to do logical comparisons or data reduction between two or more indexes. Or you can add to or change the user data portion of the pointer segment. The only way an application program can see user data or the contents of the duplicate data field is by processing the secondary index as a separate database.

In processing a secondary index as a separate database, several processing restrictions designed primarily to protect the secondary index database exist. The restrictions are as follows:

- Segments cannot be inserted.
- Segments can be deleted. Note, however, that deleted segments can make your secondary index invalid for use as an index.
- The key field in the pointer segment (which consists of the search field, and if they exist, the constant and subsequence fields) cannot be replaced.

In addition to the restrictions imposed by the system to protect the secondary index database, you can further protect it using the PROT operand in the DBD statement. When PROT is specified, an application program can only replace user data in a pointer segment. However, pointer segments can still be deleted when PROT is specified. When a pointer segment is deleted, the source segment that caused the pointer segment to be created is not deleted. Note the implication of this: IMS might try to do maintenance on a pointer segment that has been deleted. When it finds no pointer segment for an existing source segment, it will return an NE status code. When NOPROT is specified, an application program can replace all fields in a pointer segment except the constant, search, and subsequence fields. PROT is the default for this parameter.

For an application program to process a secondary index as a separate database, you merely code a PCB for the application program. This PCB must reference the DBD for the secondary index. When an application program uses qualified SSAs to process a secondary index database, the SSAs must use the complete key of the pointer segment as the qualifier. The complete key consists of the search field and the subsequence and constant fields (if these last two fields exist). The PCB key feedback area in the application program will contain the entire key field.

If you are using a shared secondary index, calls issued by an application program (for example, a series of GN calls) will not violate the boundaries of the secondary index they are against. Each secondary index in a shared database has a unique DBD name and root segment name.

Related concepts:

"Format and use of fields in a pointer segment" on page 347

Sharing secondary index databases

An index database can contain up to 16 secondary indexes. When a database contains more than one secondary index, the database is called a *shared index database*. HALDBs and DEDBs do not support shared secondary indexes.

Although using a shared index database can save some main storage, the disadvantages of using a shared index database generally outweigh the small amount of space that is saved by its use.

The original advantage of a shared index database was that it saved a significant amount of main storage for buffers and some control blocks. However, when VSAM was enhanced with shared resources, the savings in storage became less significant.

For example, performance can decrease when more than one application program simultaneously uses the shared index database. (Search time is increased because the arm must move back and forth between more than one secondary index.) In addition, maintenance, recovery, and reorganization of the shared index database can decrease performance because all secondary indexes are, to some extent, affected if one is. For example, when a database that is accessed using a secondary index is reorganized, IMS automatically builds a new secondary index. This means all other indexes in the shared database must be copied to the new shared index.

If you are using a shared index database, you need to know the following information:
- A shared index database is created, accessed, and maintained just like an index database with a single secondary index.
- The various secondary indexes in the shared index database do not need to index the same database.
- One shared index database could contain all secondary indexes for your installation (if the number of secondary indexes does not exceed 16).

In a shared index database:
- All index segments must be the same length.
- All keys must be the same length.
- The offset from the beginning of all segments to the search field must be the same. This means all keys must be either unique or non-unique. With non-unique keys, a pointer field exists in the target segment. With unique keys, it does not. So the offset to the key field, if unique and non-unique keys were mixed, would differ by 4 bytes.

  If the search fields in your secondary indexes are not the same length, you might be able to force key fields of equal length by using the subsequence field. You can put the number of bytes you need to make each key field an equal length in the subsequence field.
- Each shared secondary index requires a constant specified for it, a constant that uniquely identifies it from other indexes in the secondary index database. IMS puts this identifier in the constant field of each pointer segment in the secondary index database. For shared indexes, the key is the constant, search, and (if used) the subsequence field.

Shared secondary index database commands

Commands sometimes operate differently depending on whether they are issued for the first of the secondary indexes or for subsequent secondary indexes. The first secondary index is the first database name specified in the DBDUMP statement of the shared secondary index DBDGEN. This first database is the **real** database. Other secondary index databases are physically part of the **real** database but they are logically distinct.

The first column in the following table lists the issuing command, the second column lists where the command is issued, the third column lists the affects of the command that was issued, and the fourth column provides additional comments.

Table 61. The effects of issuing shared secondary index database commands

Issuing the Commands...	On the...	Affects...	Comments
/STOP /LOCK UPDATE DB STOP(SCHD) UPDATE DB SET(LOCK(ON))	First secondary index	Only the named database	If no applications are scheduled on any shared secondary indexes that cause the authorization of the real database by DBRC, the commands have the same effect as the /DBRECOVERY or UPD DB STOP(ACCESS)command on the first secondary index. When a /DISPLAY DB or QUERY DB command is issued on the shared secondary index database, the subsequent secondary indexes are shown as stopped or locked only if the /STOP, UPD DB STOP(SCHD), /LOCK, UPD DB SET(LOCK(ON)), UPD DB STOP(ACCESS), or /DBRECOVERY command was issued. To undo the /STOP, UPD DB STOP(SCHD), UPD DB SET(LOCK(ON)), or /LOCK command, issue a /START, UPD DB START(ACCESS), UPD DB SET 1OCK(OFF)), or /UNLOCK command on the first secondary index.
/STOP UPD DB STOP(SCHD) /LOCK UPD DB SET(LOCK(ON))	Subsequent secondary indexes	Only the named database	To undo the /STOP, UPD DB STOP(SCHD), UPD DB SET(LOCK(ON)), or /LOCK command, issue a /START, UPD DB START(ACCESS), UPD DB SET(LOCK(OFF)), or /UNLOCK command on the named database.
/DBDUMP UPD DB STOP(UPDATES)	First secondary index	All databases sharing the secondary index data set	The /DBDUMP or UPD DB STOP(UPDATES) command quiesces activity on all the indexes in the shared database. The database is then closed and reopened for input only. To undo the /DBDUMP or UPD DB STOP(UPDATES) command, issue a /START or UPD DB START(ACCESS) command on the first secondary index.

Table 61. The effects of issuing shared secondary index database commands (continued)

Issuing the Commands...	On the...	Affects...	Comments
/DBDUMP UPD DB STOP(UPDATES)	Subsequent secondary indexes	Only the named database	The secondary index is available for read only. To undo the /DBDUMP or UPD DB STOP(UPDATES) command, issue a /START or UPD DB START(ACCESS)command on the named database.
/DBRECOVERY UPD DB STOP(ACCESS)	First secondary index	All databases sharing the secondary index data set	The /DBRECOVERY and UPD DB STOP(ACCESS) command quiesces activity on all the indexes in the shared database. The database is then closed and stopped. When the /DISPLAY command is issued on the shared secondary index database, the subsequent secondary indexes are shown as stopped or locked only if the /STOP, UPD DB STOP(SCHD), /LOCK, UPD DB SET(LOCK(ON)), /DBRECOVERY, or UPD DB STOP(ACCESS) command was issued. To undo the /DBRECOVERY or UPD DB STOP(ACCESS) command, issue a /START or UPD DB START(ACCESS) command on the first secondary index.
/DBRECOVERY UPD DB STOP(ACCESS)	Subsequent secondary indexes	Only the named database	This command is the same as the /STOP and UPD DB STOP(SCHD) command for the named database. However, the /DBRECOVERY and UPD DB STOP(ACCESS) command works immediately, but the /STOP and UPD DB STOP(SCHD) command allows current work to quiesce. To undo the /DBRECOVERY or UPD DB STOP(ACCESS) command, issue a /START or UPD DB START(ACCESS) command on the named database.

Related concepts:

"Format and use of fields in a pointer segment" on page 347

INDICES= parameter

You can specify an INDICES= parameter on the PCB in the SENSEG statement, to specify a secondary index that contains search fields used to qualify SSAs for an indexed segment type.

The use of the INDICES= parameter does not alter the processing sequence selected for the PCB by the presence or absence of the PROCSEQ= parameter.

The INDICES= parameter is not supported if the primary database is a DEDB. The INDICES= parameter is not supported by Fast Path secondary index.

The following figure and code examples illustrate the use of the INDICES=parameter.
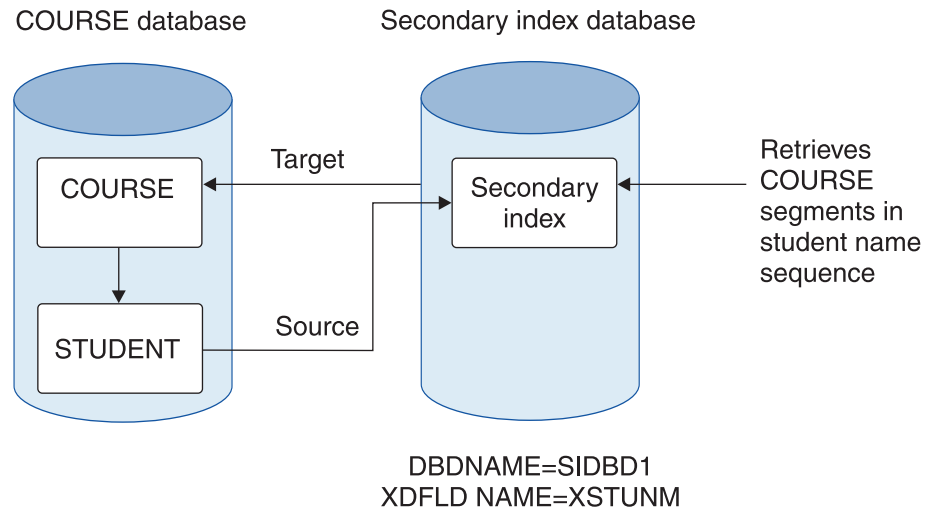
Figure 188. Databases for first example of the INDICES= parameter

```
PCB
SENSEG NAME=COURSE, INDICES=SIDBD1
SENSEG NAME=STUDENT
```

Figure 189. PCB for the first example of the INDICES= parameter

```
GU COURSE COURSENM=12345&.XSTUNM=JONES
```

Figure 190. Application program call issued for the first example of the INDICES= parameter

When the preceding GU call is used, IMS gets the COURSE segment with a number 12345. Then IMS gets a secondary index entry, one in which XSTUNM is equal to JONES. IMS checks to see if the pointer in the secondary index points to the COURSE segment with course number 12345. If it does, IMS returns the COURSE segment to the application program's I/O area. If the secondary index pointer does not point to the COURSE segment with course number equal to 12345, IMS checks for other secondary index entries with XSTUNM equal to JONES and repeats the compare.

If all secondary index entries with XSTUNM equal to JONES result in invalid compares, no segment is returned to the application program. By doing this, IMS need not search the STUDENT segments for a student with NAME equal to JONES. This technique involving use of the INDICES= parameter is useful when source and target segments are different.

The following figure shows the databases for the second example of the INDICES parameter.
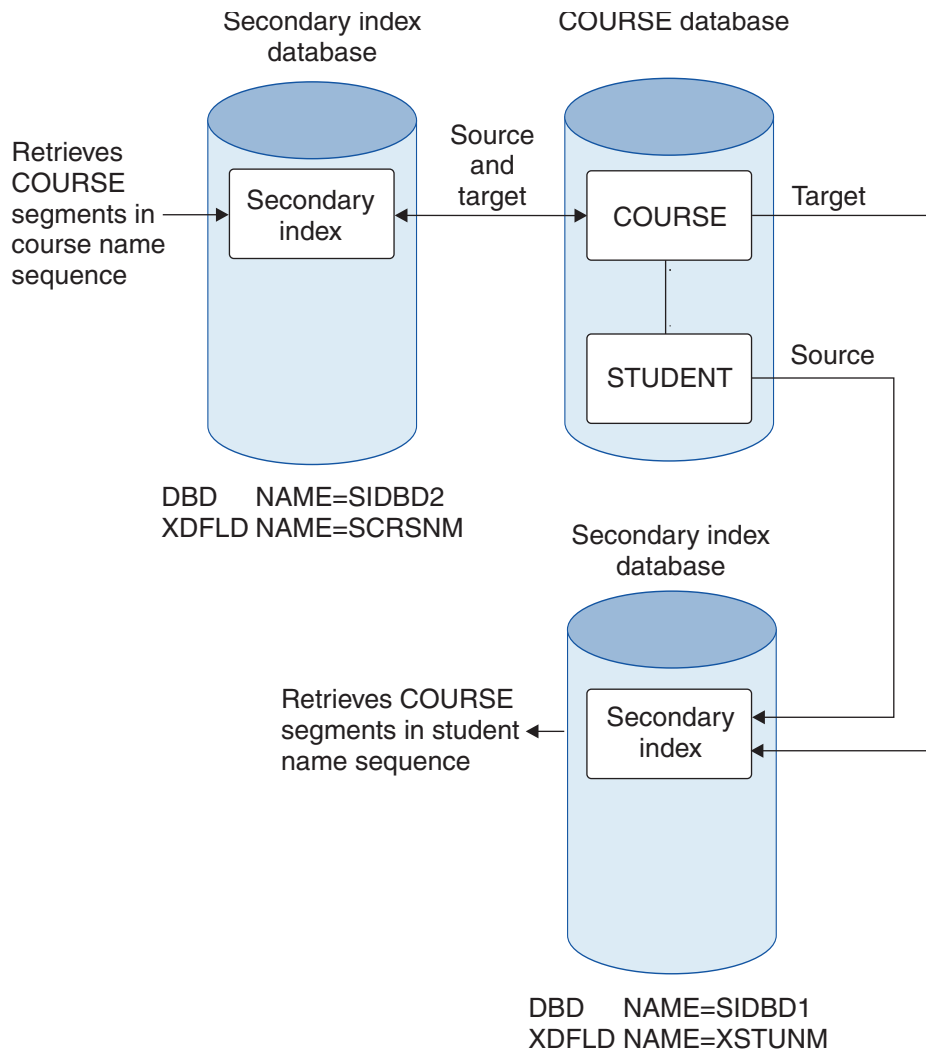
Figure 191. Databases for second example of the INDICES= parameter

The following code shows the example PCB.

```
PCB      PROCSEQ=SIDBD2
SENSEG NAME=COURSE, INDICES=SIDBD1
SENSEG NAME=STUDENT
```

Figure 192. PCB for the second example of the INDICES= parameter

The following code shows the example application programming call.

```
GU COURSE SCRSNM=MATH&XSTUNM=JONES
```

Figure 193. Application program call issued for the second example of the INDICES= parameter

Compare process and performance

Excluding COURSENM=12345 in the preceding GU call would impact performance. IMS retrieves the first COURSE segment in the COURSE database, and then a secondary index entry in which XSTUNM is equal to JONES. IMS checks to see if the pointer in the secondary index points to the COURSE segment just retrieved. If it does, IMS returns the COURSE segment to the application

program's I/O area. If the secondary index pointer does not point to this COURSE segment, IMS checks for other secondary index entries with XSTUNM equal to JONES and repeats the compare. If all secondary index entries with XSTUNM equal to JONES result in invalid compares, IMS retrieves the next COURSE segment and the secondary index entries as before, then repeats the compare. If all the COURSE segments result in invalid compares, no segment is returned to the application program.

The INDICES= parameter can also be used to reference more than one secondary index in the source call. The GU call shown in Figure 191 on page 364 shows another example of the INDICES=parameter.

In the figure shown in Figure 193 on page 364, IMS uses the SIDBD2 secondary index to get the COURSE segment for MATH. IMS then gets a COURSE segment using the SIDBD1. IMS can then compare to see if the two course segments are the same. If they are, IMS returns the COURSE segment to the application program's I/O area. If the compare is not equal, IMS looks for other SIDBD1 pointers to COURSE segments and repeats the compare operations. If there are still no equal compares, IMS checks for other SIDBD2 pointers to COURSE segments and looks for equal compares to SIDBD1 pointers. If all possible compares result in unequal compares, no segment is returned to the application program.

Note: This compare process can severely degrade performance.

Related concepts:

"Using secondary indexes with logical relationships"

Using secondary indexes with logical relationships

You can use secondary indexes, except for Fast Path secondary indexes, with logical relationships.

When creating or using a secondary index for a database that has logical relationships, the following restrictions exist:

- A logical child segment or a dependent of a logical child cannot be a target segment.
- A logical child cannot be used as a source segment; however, a dependent of a logical child can.
- A concatenated segment or a dependent of a concatenated segment in a logical database cannot be a target segment.
- When using logical relationships, no qualification on indexed fields is allowed in the SSA for a concatenated segment. However, an SSA for any dependent of a concatenated segment can be qualified on an indexed field.

Related concepts:

"INDICES= parameter" on page 362

Using secondary indexes with variable-length segments

If a variable-length segment is a source segment, when an occurrence of it is inserted that does not have fields specified for use in the search, subsequence, or duplicate data fields of the pointer segment, the following events can occur.

- If the missing source segment data is used in the search field of the pointer segment, no pointer segment is put in the index.

- If the missing source segment data is used in the subsequence or duplicate data fields of the pointer segment, the pointer segment is put in the index. However, the subsequence or duplicate data field will contain one of the three following representations of zero:

    P = X'0F'

    X = X'00'

    C = C'0'

  Which of these is used is determined by what is specified on the FIELD statements in the DBD that defined the source segment field.

Considerations when using secondary indexing

When you use secondary indexes with your databases, you should be aware of a number of considerations.

The secondary index considerations that you should be aware of include:

- When a source segment is inserted into or deleted from a database, an index pointer segment is inserted into or deleted from the secondary index. This maintenance always occurs regardless of whether the application program doing the updating is using the secondary index.
- When an index pointer segment is deleted by a REPL or DLET call, position is lost for all calls within the database record for which a PCB position was established using the deleted index pointer segment.
- When replacing data in a source segment, if the data is used in the search, subsequence, or duplicate data fields of a secondary index, the index is updated to reflect the change as follows:
  - If data used in the duplicate data field of the pointer segment is replaced in the source segment, the pointer segment is updated with the new data.
  - If data used in the search or subsequence fields of the pointer segment is replaced in the source segment, the pointer segment is updated with the new data. In addition, the position of the pointer segment in the index is changed, because a change to the search or subsequence field of a pointer segment changes the key of the pointer segment. The index is updated by deleting the pointer segment from the position that was determined by the old key. The pointer segment is then inserted in the position determined by the new key.
- The use of secondary indexes increases storage requirements for all calls made within a specific PCB when the processing option allows the source segment to be updated. Additional storage requirements for each secondary index database range from 6K to 10K bytes. Part of this additional storage is fixed in real storage by VSAM.
- You should always compare the use of secondary indexing with other ways of achieving the same result. For example, to produce a report from an HDAM or PHDAM database in root key sequence, you can use a secondary index. However, in many cases, access to each root sequentially is a random operation. It would be very time-consuming to fully scan a large database when access to each root is random. It might be more efficient to scan the database in physical sequence (using GN calls and no secondary index) and then sort the results by root key to produce a final report in root key sequence.
- When calls for a target segment are qualified on the search field of a secondary index, and the indexed database is not being processed using the secondary index, additional I/O operations are required. Additional I/O operations are required because the index must be accessed each time an occurrence of the target segment is inspected. Because the data in the search field of a secondary

index is a duplication of data in a source segment, you should decide whether an inspection of source segments might yield the same result faster.
- When using a secondary data structure, the target segment and the segments on which it was dependent (its physical parents) cannot be inserted or deleted.

Example of defining secondary indexes

The secondary index in this example is used to retrieve COURSE segments based on student names.

The example is for a full-function database and uses direct, rather than symbolic, pointers.

For additional examples, including examples of defining secondary indexes for Fast Path DEDB databases, see Examples with secondary indexes (System Utilities).

Figure 194 on page 368 shows the EDUC database and its secondary index. The code samples that follow show the two DBDs required for the databases.

The pointer segment in the secondary index contains a student name in the search field and a system related field in the subsequence field. Both of these fields are defined in the STUDENT segment. The STUDENT segment is the source segment. The COURSE segment is the target segment.

The DBDs in the code samples highlight the statements and parameters coded when a secondary index is used. (Wherever statements or parameters are omitted the parameter in the DBD is coded the same regardless of whether secondary indexing is used.)

DBD for the EDUC database

An LCHILD and XDFLD statement are used to define the secondary index. These statements are coded after the SEGM statement for the target segment.
- LCHILD statement. The LCHILD statement specifies the name of the secondary index SEGM statement and the name of the secondary index database in the NAME= parameter. The PTR= parameter is always PTR=INDX when a secondary index is used.
- XDFLD statement. The XDFLD statement defines the contents of the pointer segment and the options used in the secondary index. It must appear in the DBD input deck after the LCHILD statement that references the pointer segment.

In the example, shown in Figure 194 on page 368, a system-related field (/SX1) is used on the SUBSEQ parameter. System-related fields must also be coded on FIELD statements after the SEGM for the source segment.
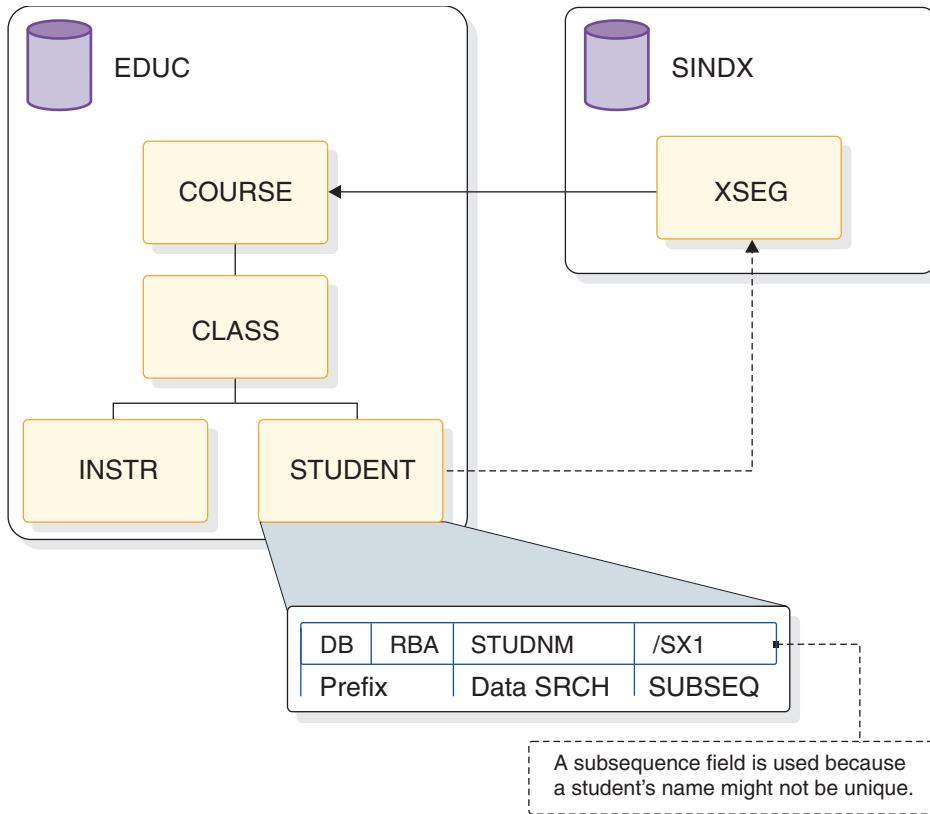
Figure 194. Databases for secondary indexing example

The following code shows the EDUC DBD for the example in Figure 194.

```
DBD    NAME=EDUC,ACCESS=HDAM,...
SEGM   NAME=COURSE,...
FIELD  NAME=(COURSECD,...
LCHILD NAME=(XSEG,SINDX),PTR=INDX
XDFLD  NAME=XSTUDENT,SEGMENT=STUDENT,SRCH=STUDNM,SUBSEQ=/SX1
SEGM   NAME=CLASS,...
FIELD  NAME=(EDCTR,...
SEGM   NAME=INSTR,...
FILED  NAME=(INSTNO,...
SEGM   NAME=STUDENT,...
FIELD  NAME=SEQ,...
FIELD  NAME=STUDNM,BYTES=20,START=1
FIELD  NAME=/SX1

DBDGEN
FINISH
END
```

The following code shows the SINDX DBD for the example in Figure 194.

```
DBD    NAME=SINDX,ACCESS=INDEX
SEGM   NAME=XSEG,...
FIELD  NAME=(XSEG,SEQ,U),BYTES=24,START=1
LCHILD NAME=(COURSE,EDUC),INDEX=XSTUDENT,PTR=SNGL

DBDGEN
FINISH
END
```

DBD for the SINDX database

DBD statement

The DBD statement specifies the name of the secondary index database in the NAME= parameter. The ACCESS= parameter is always ACCESS=INDEX for the secondary index DBD.

SEGM statement

You choose what is used in the NAME= parameter. This value is used when processing the secondary index as a separate database.

FIELD statement

The NAME= parameter specifies the sequence field of the secondary index. In this case, the sequence field is composed of both the search and subsequence field data, the student name, and the system-related field /SX1. You specify what is chosen by NAME=parameter.

LCHILD statement

The LCHILD statement specifies the name of the target, SEGM, and the name of the target database in the NAME= parameter. The INDEX= parameter has the name on the XDFLD statement in the target database. If the pointer segment contains a direct-address pointer to the target segment, the PTR= parameter is PTR=SNGL. The PTR= parameter is PTR=SYMB if the pointer segment contains a symbolic pointer to the target segment.

Related tasks:

"Adding a secondary index to a full-function database" on page 746

"Making keys unique using system related fields" on page 354

Related reference:

➥ Examples with secondary indexes (System Utilities)

➥ DBDGEN statements (System Utilities)

DEDB partitioned secondary indexes

DEDB partitioned secondary indexes allow a DEDB secondary index to be spread across multiple physical databases. A maximum of 101 user partition databases per Fast Path secondary index database are supported.

A very large secondary index can be partitioned into multiple physical HISAM or SHISAM databases. A user partition selection routine determines which partition an index key is assigned to. The first partition name is used in the PCB definition to represent the whole user partition group.

The following figure illustrates five index databases, each with a user-specified range of pointers that all point to one DEDB.
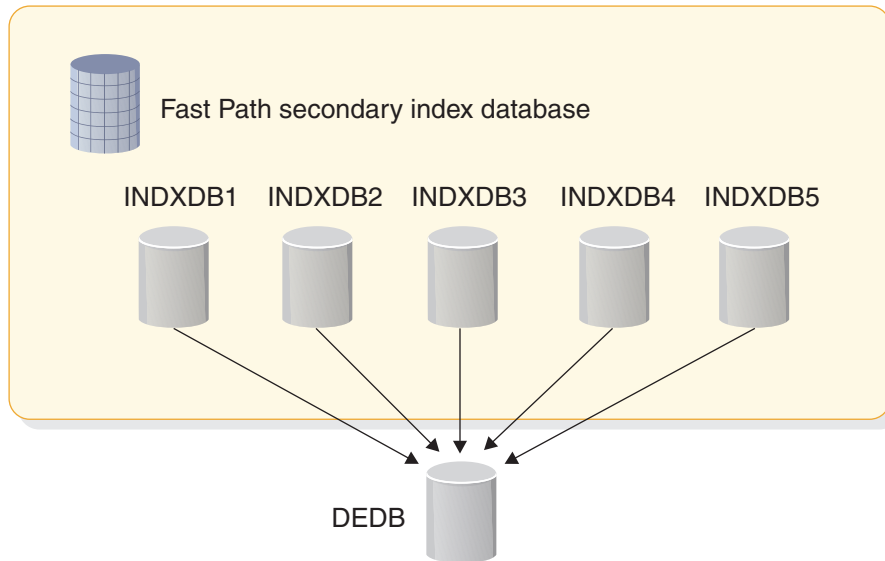
Figure 195. Secondary index that spans multiple physical databases

Each index database contains a range of keys. Index keys are assigned to an index database by a user partition selection exit routine.

Either HISAM or SHISAM can be used, but all partition databases in a user partition group must be defined as either all HISAM or all SHISAM. Each of the databases in an index must have the same structure and attributes. They can have different sizes to accommodate the number of entries that could exist in the different key ranges.

If user partitioning is requested for a HISAM secondary index or a SHISAM secondary index, two or more HISAM or SHISAM secondary index databases, respectively, can be specified in the NAME= parameter on the LCHILD statement of a primary DEDB database DBD.

Because a HISAM secondary index supports unique or non-unique keys with fixed length secondary index segments, and a SHISAM secondary index only supports unique keys with fixed length secondary index segments, user partitioning for HISAM secondary indexes cannot to have any SHISAM secondary index databases in the same user partition group. User partitioning for SHISAM secondary indexes cannot have any HISAM secondary index databases in the same user partition group. All partition databases in a user partition group must have the same access type defined in the secondary index DBD definitions for the partition databases:

- ACCESS=(INDEX,VSAM) for a HISAM secondary index database
- ACCESS=(INDEX,SHISAM) for a SHISAM secondary index database

When HISAM and SHISAM secondary index databases are included in the same user partition group in the NAME= parameter on a LCHILD statement, the ACBGEN utility detects the inconsistency and issues message DFS2293E. The primary DEDB database and its secondary index databases are deleted in the ACBLIB.

All HISAM secondary index partition databases in a user partition group must be defined as KSDS data set only for unique key support, or must be defined as both KSDS and ESDS data sets for non-unique key support.

When a HISAM secondary index user partition group includes both unique key and non-unique key HISAM secondary index databases, the ACBGEN utility detects the inconsistency and issues message DFS2294E. The primary DEDB database and its secondary index databases are deleted in the ACBLIB.

Single or multiple partition databases for a HISAM or SHISAM index

When you specify the PSELRTN= parameter on a XDFLD statement to request user partitioning, you can also control the number of partition databases to be processed using the PCB with the PROCSEQD= parameter before IMS returns a GB status code to indicate the end of the database.

In addition to the PSELRTN= parameter, the PSELOPT=MULT|SNGL parameter can be used to indicate how many partition databases in a logical HISAM secondary index database or a logical SHISAM secondary index database are processed before a GB status code is returned to an application to indicate the end of the database when the DEDB Partition Selection exit routine is not called.

For qualified GU calls with SSA that has equal-to or greater-than-or-equal-to relational operator, the DEDB Partition Selection exit routine is called to determine the partition database to be selected based on the search key in the SSA or other user defined partition selection criteria.

For qualified GU/GN calls with an SSA that has equal-to or greater-than-or-equal-to relational operator, the DEDB Partition Selection exit routine is not called. Instead, the segment is searched from the beginning of the first user partition database in the user partition group.

For unqualified GN calls or qualified GN calls with no SSA, the DEDB Partition Selection exit routine is not called.

For qualified GN calls with an SSA that has an equal-to or greater-than-or-equal-to relational operator, the DEDB Partition Selection exit routine is called when the position is not established yet. If the position is already established, the DEDB Partition Selection exit routine is not called.

The following table summarizes the conditions of DL/I GU and GN calls on whether or not the DEDB Partition Selection exit routine is called:

Table 62. Conditions of DL/I GU and GN calls

Call type	Action
Qualified GU with equal-to relational operator GU COURSE(NAMEINDX =CHEMISTRY)	Call the DEDB Partition Selection exit routine to select the user partition database.
Qualified GU with greater-than-or-equal-to relational operator GU COURSE(NAMEINDX>=CHEMISTRY)	Call the DEDB Partition Selection exit routine to select the user partition database.
Qualified GU with less-than-or-equal-to relational operator GU COURSE(NAMEINDX<=CHEMISTRY)	Do not call the DEDB Partition Selection exit routine. Search the segment from the beginning of the first user partition database.
Unqualified GN GN	Do not call the DEDB Partition Selection exit routine.
Qualified GN with no SSA GN COURSE	Do not call the DEDB Partition Selection exit routine.

Table 62. Conditions of DL/I GU and GN calls (continued)

Call type	Action
Qualified GN with equal-to relational operator GN COURSE(NAMEINDX=CHEMISTRY)	If the current position is not established yet, call the DEDB Partition Selection exit routine to select the user partition database. If the current position is already established, search the segment after the position.
Qualified GN with greater-than-or-equal-to relational operator GN COURSE(NAMEINDX >=CHEMISTRY)	If the current position is not established yet, call the DEDB Partition Selection exit routine to select the user partition database. If the current position is already established, search the segment after the position.
Qualified GN with less-than-or-equal-to relational operator GN COURSE(NAMEINDX<=CHEMISTRY	Do not call the DEDB Partition Selection exit routine. Search the segment from the beginning of the first user partition database.

The PSELOPT=MULT|SNGL is defaulted on an XDFLD statement when user partitioning is requested. However, the PSELOPT= MULT|SNGL must be explicitly specified on a PCB statement with the PROCSEQD= parameter. There is no default for the PSELOPT= parameter on the PCB statement with the PROCSEQD= parameter because the value on the PCB statement overrides those specified in the XDFLD statement. The PSELOPT= parameter on the PCB statement with the PROCSEQD= parameter cannot default to PSELOPT=MULT, which is the default value on the XDFLD statement, because its value overrides those values that are explicitly specified or implicitly defaulted on XDFLD statements in the primary DEDB DBD.

The PSELOPT=MULT|SNGL parameter can be specified in two places:
- On the PSELOPT=MULT|SNGL on a PCB statement with the PROCSEQD= parameter in a PSB
- On a XDFLD statement for the primary DEDB DBD with a HISAM or a SHISAM secondary index defined

If the PSELOPT= parameter is specified both places, the parameter that is specified on the PCB statement overrides the parameter that is specified on the XDFLD statement.

When PSELOPT=MULT is defined on a PCB statement with the PROCSEQD= parameter or on a XDFLD statement, it means that multiple user partition databases in the user partition group are processed starting from the user partition database selected by the DEDB Partition Selection exit routine and continues to the last user partition database sequentially as defined in the NAME= parameter on the LCHILD statement of a primary DEDB database DBD. A GB status code is returned when it reaches the end of database on the last user partition as defined in the NAME= parameter on the LCHILD statement of the primary DEDB DBD. When PSELOPT=MULT is specified, one or more user partition databases are processed.

When PSELOPT=SNGL is defined on a PCB statement with the PROCSEQD= parameter or on a XDFLD statement, it means that only a single user partition database is processed in a user partition group as selected by the DEDB Partition Selection exit routine. A GB status code is returned when the end of data is reached on the selected user partition database. When PSELOPT=SNGL is specified, only one user partition database is processed.

Restrictions:

The following restrictions apply to DEDB partitioned secondary indexes:
- If the PSELOPT= parameter is specified on a XLFLD statement and only one secondary index database is specified on its corresponding LCHILD statement, the DBDGEN utility terminates with MNOTE 8 and message XDFLD233.
- If the PSELOPT= parameter specified on a XDFLD statement is not PSELOPT= MULT or PSELOPT= SNGL, the DBDGEN utility terminates with MNOTE 8 and message XDFLD234.

Example of user partition with PSELOPT=MULT

The following example illustrates a user partition group with four secondary index databases and PSELOPT=MULT.

```
LCHILD NAME=(NAMEXSEG,(NAMSXDB1,NAMSXDB2,NAMSXDB3,NAMSXDB4)),PTR=SYMB
XDFLD NAME=NAMEINDX,SRCH=COURNAME,PSELRTN=DBFPSE00,PSELOPT=MULT
```

In this example, four secondary index databases are defined in a user partition group. The DEDB Partition Selection exit routine is DBFPSE00. The user partition selection option is defined as PSELOPT=MULT.

The DEDB Partition Selection exit routine selects user partition database NAMSXDB2 on a qualified GU call. Subsequent qualified GN calls with no SSA process user partition databases NAMSXDB2, NAMSXDB3, and NAMSXDB4 until the end of data is reached.

With PSELOPT=MULT and the starting user partition database used as NAMSXDB2, IMS treats NAMSXDB2, NAMSXDB3, and NAMSXDB4 as one logical database. IMS returns a GB status code to an application on qualified GN calls with no SSA processing when the end of data is reached on the last user partition in the logical database: NAMSXDB4.

Example of user partition with PSELOPT=SNGL

The following example illustrates a user partition group with four secondary index databases and PSELOPT=SNGL.

```
LCHILD NAME=(NAMEXSEG,(NAMSXDB1,NAMSXDB2,NAMSXDB3,NAMSXDB4)),PTR=SYMB
XDFLD NAME=NAMEINDX,SRCH=COURNAME,PSELRTN=DBFPSE00,PSELOPT=SNGL
```

In this example, four secondary index databases are defined in a user partition group. The DEDB Partition Selection exit routine is DBFPSE00. The user partition selection option is defined as PSELOPT=SNGL.

When a DEDB Partition Selection exit routine returns user partition database NAMSXDB2 and PSELOPT=SNGL defined, IMS processes the second user partition database only. IMS returns a GB status code to an application on qualified GN calls with no SSA when the end of data is reached on user partition database NAMSXDB2.

Related reference:

➡ Data Entry Database Partition Selection exit routine (DBFPSE00) (Exit Routines)

Multiple index entries for Fast Path secondary indexes

You can create multiple Fast Path secondary index entries from different fields in the same source segment.

More than one search field can be defined for a secondary index from fields in the same source segment. Each search field (or set of search fields) is used to create an entry in the secondary index. These search fields (or sets of search fields) must each be the same size.

To create multiple index entries from a single source segment, define two or more LCHILD/XDFLD statement pairs under the SEGM statement of a target segment and specify the MULTISEG=YES on the LCHILD statement.

In the following DEDB, there are three fields for telephone numbers in the OWNER segment.

```
DBD NAME=ACCTDB,ACCESS=DEDB,RMNAME=RMD4
AREA DD1=ACCT1,SIZE=1024,UOW=(100,10),
ROOT=(236,36)
SEGM NAME=ACCT,PARENT=0,BYTES=100
FIELD NAME=(ACCTNO,SEQ,U),BYTES=12,START=1
LCHILD NAME=(PHONKEY,PHONINDX),PTR=SYMB,MULTISEG=YES
XDFLD NAME=XPHON,SEGMENT=OWNER,SRCH=HOMEPHN
LCHILD NAME=(PHONKEY,PHONINDX),PTR=SYMB,MULTISEG=YES
XDFLD NAME=XPHON,SEGMENT=OWNER,SRCH=WORKPHN
LCHILD NAME=(PHONKEY,PHONINDX),PTR=SYMB,MULTISEG=YES
XDFLD NAME=XPHON,SEGMENT=OWNER,SRCH=CELLPHN
SEGM NAME=OWNER,BYTES=300,PARENT=ACCT
FIELD NAME=OWNNAME,BYTES=40,START=1
FIELD NAME=HOMEPHN,BYTES=10,START=41
FIELD NAME=WORKPHN,BYTES=10,START=51
FIELD NAME=CELLPHN,BYTES=10,START=61
DBDGEN
```

The secondary index DBD for this example is:

```
DBDSX DBD NAME=PHONINDX,ACCESS=(INDEX,VSAM),FPINDEX=YES
DATASET DD1=PHONKSDS,OVFLW=PHONOVFL
SEGM NAME=PHONSEG,PARENT=0,BYTES=22
FIELD NAME=(PHONEKEY,SEQ,U),BYTES=10,START=1
LCHILD NAME=(OWNER,ACCTDB),INDEX=XPHON,PTR=SYMB
DBDGEN
```

Considerations for HALDB partitioned secondary indexes

A secondary index of a HALDB database must be a HALDB partitioned secondary index (PSINDEX).

A PSINDEX can have one or more partitions. The key ranges for the partitions of a secondary index are not likely to match the key ranges of its target database. Partition selection for the secondary index is based on its key; partition selection for the target database is based on its key. Usually, these keys are unrelated. In some cases, you can use a partition selection exit routine to partition the secondary index along the same boundaries of the main database. An appropriate key is required, because the selection of a partition always depends on the root key. The partition selection exit routine uses the parts of the root key that you specify to select the correct partition.

To initialize partitions in a PSINDEX, use the HALDB Partition Data Set Initialization utility (DFSUPNT0). DFSUPNT0 automatically generates recovery

points for the PSINDEX. Recovery points are not created if you delete and redefine your PSINDEX partitions and then turn off their PINIT flags.

Additionally, there are other restrictions and requirements for HALDB partitioned secondary indexes:

- Symbolic pointing is not supported.
- Shared secondary indexes are not supported.
- Secondary indexes must have unique keys.
- /SX and /CK fields can be used to provide uniqueness.
- The reorganization of a target database does not require the rebuilding of its secondary indexes. HALDB databases use an indirect list data set (ILDS) to maintain pointers between the PSINDEX and the target segments.

Related concepts:

"The HALDB self-healing pointer process" on page 691

Chapter 19. Database versioning

When database versioning and the IMS catalog are enabled in an IMS system, you can maintain multiple different versions of a database. Application programs can access the older versions of the database without being changed, while the newer versions of the database can be changed to support new application program requirements.

Database versioning overview

To create a new version of a database, database versioning must be enabled by specifying DBVERSION=Y in the DFSDFxxx PROCLIB member.

After database versioning is enabled, you create a new version of a database by updating the database definition (DBD) of the database with a version number and the structural changes you are introducing with the new version. The version number is specified on the DBVER keyword and must increase incrementally by one in each new version.

After the DBD and ACB members for the new version are generated and the ACB members are activated in the ACB library, the older versions of the database exist only in the DBD record of the database in the IMS catalog. When an application program requests data from a version of the database that is not the active version in the ACB library, IMS retrieves the requested version from the IMS catalog.

Attention: If a DBD record in the IMS catalog contains multiple instances of a version of the DBD, IMS uses the most recent instance to return the data, as determined by the ACB generation timestamp.

The database version that an application program requires can be specified on the DBVER= parameter in the PCB or at runtime by issuing the INIT DL/I call with the VERSION subfunction.

By default, when no version number is specified for an application program, IMS returns data that conforms to the DBD that is currently active in the ACB library. Usually, the version that is active in the ACB library reflects the current physical structure of the database on DASD.

If version 0 of a DBD still exists in the IMS catalog, you can change the default so that IMS returns data by using version 0 of the DBD. Version 0 of a DBD is the most recent instance of the DBD in the IMS catalog that does not contain the DBVER parameter. If version 0 of a DBD no longer exists in the IMS catalog, only the active version of the DBD can be used as the default version of the DBD.

The default can be set at the system level and at the program specification block (PSB) level. The PSB level default overrides the system level default.

The system level default is set by the DBLEVEL= keyword in the DFSDFxxx member. The PSB level default is set by the DBLEVEL= keyword on the PSBGEN statement of the PSB definition.

If an application program requests a database version that does not exist in the IMS catalog, IMS returns a status code to the application program.

When database versioning is not enabled, IMS ignores any specified version numbers in PSB or DBD macros and returns data only in the format that is defined by the physical database structure. If an application program specifies a version number on the DBVER= parameter of a PCB when database versioning is not enabled, IMS either terminates the application program with a 3303 abend or returns a BA DL/I status code to the application program, depending on how your application program is designed.

Recommendation: As soon as you create a new version of a database, confirm that application programs can still access the prior versions of the database. Database versioning supports only certain changes to the structure of a database. The only way to validate that new changes are supported by database versioning is to schedule an application program to access a prior version of the database. Because the most recent version of the database does not require the data to be reformatted before returning it to an application program, accessing the most recent version does not detect unsupported changes.

Database versioning supports the following changes to a database definition:
- An increase to the length of a segment.
- The addition of new fields to existing undefined space at the end of a segment.

Attention: Before you add new fields to a segment in existing space that is undefined to IMS, you must make sure that your existing application programs do not initialize, modify, or otherwise use the free space.

The following database types support database versioning:
- DEDB
- HDAM
- HIDAM
- PHDAM
- PHIDAM

IMS database versioning supports both databases that have logical relationships and databases that have secondary indexes. However, segments that are involved in the relationship between the two databases, or that are paired with another segment in the same database, cannot change in new versions of the DBD. Only the segments in the databases that are not involved in a relationship can change.

Related concepts:

Chapter 5, "Overview of the IMS catalog," on page 43

Related reference:

⇨ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

⇨ Database Description (DBD) Generation utility (System Utilities)

⇨ Program Specification Block (PSB) Generation utility (System Utilities)

IMS catalog support for database versioning

The IMS catalog is required for database versioning.

Except for the version of the DBD that is designated in the IMS catalog as the active DBD, all prior versions of the DBD of a database exist only in the IMS catalog in the DBD record of the database.

When an application program makes a database call that requires a prior version of the DBD, IMS transforms the data that is retrieved from the database by using both the prior version of the DBD and the active DBD. When an application program makes a database call that does not require a prior version of the DBD, IMS uses only the active DBD to return the data.

Attention: If a prior version of a DBD is deleted from the catalog, that definition of the database is lost and application programs that use that version of the DBD can no longer access the database. Moreover, if the entire IMS catalog becomes unavailable, IMS can no longer provide access to any prior version of the database; only the application programs that use the version of the DBD that is active in the IMS catalog can access the database.

Back up and recover the IMS catalog by using the same backup and recovery procedures that are used for other databases. However, when database versioning is used, recovering the IMS catalog is more complex and can take longer because of the multiple DBD versions, PSBs, and ACB members that are involved.

Related concepts:

Chapter 5, "Overview of the IMS catalog," on page 43

Database modifications supported by database versioning

When you maintain multiple versions of a database, you can make only certain structural changes to the new versions of the database.

Database versioning supports the following changes to a database definition:
- An increase to the length of a segment.
- The addition of new fields to existing undefined space at the end of a segment.

Attention: Before you add new fields to a segment in existing space that is undefined to IMS, you must make sure that your existing application programs do not initialize, modify, or otherwise use the free space.

Database versioning does not support the following changes:
- An increase to the length of a variable-length segment
- A change in the starting position of an existing field
- A change in the length of an existing field

The following series of segment definitions provide examples of the types of changes that are supported by database versioning.

For the purpose of these examples, assume that the following segment is defined in an existing database before versioning is enabled. This is version 0 of the database.

```
----------------------------
|  FLD1  |  FLD2  |  space  |
----------------------------
```

After database versioning is enabled, the definition of the database is changed to increase the length of the segment, resulting in more undefined space at the end of the segment in version 1 of the database.

```
-------------------------------------
|  FLD1  |  FLD2  |  space  |  space  |
-------------------------------------
```

In version 2 of the database, a new field, FLD3, is defined in the extra space at the end of the segment, resulting in the following segment.

Attention: Before you make a change like the one shown in the following example, you must confirm with your applications group that none of your existing application programs do anything with the free space, such as initialize it during REPL calls or use it in any way. If they do, extend the length of the segment and add your new field after the existing space, like fields FLD4 and FLD5 in the example for version 3, which is shown after the following example.

```
-----------------------------------
|  FLD1  |  FLD2  |  space |  FLD3  |
-----------------------------------
```

In version 3 of the database, the length of the segment is increased further and more fields, FLD4 and FLD5, and undefined space are added after FLD3.

```
----------------------------------------------------------------
|  FLD1  |  FLD2  |  space |  FLD3  |  FLD4  |  FLD5  |  space  |
----------------------------------------------------------------
```

Related reference:

➡ Database Description (DBD) Generation utility (System Utilities)

Database versioning, existing free space, and new fields

In the IMS definition of a segment, any bytes in the length of the segment that are not used by the fields that are defined in the segment are considered free space.

Although free space appears unused to IMS, application programs can use or otherwise alter that free space without IMS knowing. In fact, the free space in a segment might be used for fields and data that are mapped by application program code, such as a COBOL copybook.

While database versioning supports the addition of new fields in the existing free space at the end of a segment, before you define any new fields for that free space, you must make sure that it is not used by any existing application programs.

The following examples illustrate the possible conflict that can occur between existing application programs and new fields that are added to existing free space.

If the free space is used by any application programs, you can either increase the length of the segment and add the new field after the existing free space, or you can modify the existing application programs so that they do not use the free space.

The following examples illustrate the potential problem and how you can avoid it.

How existing application programs can conflict with new fields in existing free space

In the following example, the base version of a database contains a 40-byte segment that has 10 bytes of free space at the end, which is seen by an existing application program, PGMA:

```
10 bytes  20 bytes            10-byte undefined space at the end
-------------------------------------
|  123   | Maple Ave.     |      |
-------------------------------------
```

In version 1 of the same database, the segment size is increased to 50 bytes and a new 20-byte field is defined at the end of the segment. A change like this is supported by database versioning; however, if the existing application program contains code that initializes the 10-byte space at the end of the segment, the existing application program can corrupt any data that is inserted into the new field in version 1 of the database.

The following example shows how the segment looks in version 1 of the database after the segment is extended by 10-bytes, a new 20-byte field is defined, and an application program, PGMB, updates the segment with data:

```
10 bytes  20 bytes           20 bytes
-------------------------------------------------
|  123    |  Maple Ave.      | San Francisco    |
-------------------------------------------------
```

Then, PGMA makes a REPL call and, because PGMA includes code that initializes the 10 bytes of free space at the end of the segment, overwrites the data added by PGMB, as shown in the following example.

```
10 bytes  20 bytes           20 bytes
-------------------------------------------------
|  456    |  Orchard Ave.    |        cisco     |
-------------------------------------------------
```

Given how PGMA is coded, a new field should not be coded in the existing space unless PGMA is modified. As an alternative to modifying PGMA, the length of the segment can be increased and the new field can be added after the existing free space in the new DBD version..

Example of defining a new field in new space after the existing free space

The following is an address segment as seen by PGMA:

```
10 bytes  20 bytes              10-byte undefined space at the end
-----------------------------------------
|  123    |  Maple Ave.      |       |
-----------------------------------------
```

The segment is extended by 20 bytes and a new 20-byte field is defined. PGMB updates the segment with data, while the original 10-bytes of free space are untouched:

```
10 bytes  20 bytes           10 bytes  20 bytes
-----------------------------------------------------------
|  123    |  Maple Ave.      |         | San Francisco    |
-----------------------------------------------------------
```

PGMA makes a REPL call, which initializes the 10 bytes of free space; however, the new 20-byte field is untouched.

```
10 bytes  20 bytes           10 bytes  20 bytes
-----------------------------------------------------------
|  789    |  Walnut Ave.     |         | San Francisco    |
-----------------------------------------------------------
```

System default for database versioning

When an application program does not specify a specific version of a versioned database, by default IMS returns data to the application program by using the version of the database definition (DBD) that is currently active in the IMS catalog. You can change the default so that IMS returns data by using version 0 of the DBD instead.

The IMS system default for which DBD version IMS uses is controlled by the DBLEVEL= parameter in the database section of the DFSDF*xxx* member of the IMS PROCLIB data set.

The IMS system default that is specified in the DFSDF*xxx* member can be overridden by specifying a DBLEVEL value in the definition of a program specification block (PSB) or by specifying a specific version number either in the definition of a program communication block (PCB) or in a DL/I INIT VERSION call that is issued by the application program.

Specify DBLEVEL=BASE to avoid changing existing application programs

If you have a significant number of application programs that will continue to use version 0 when a new version of a database is introduced, change the system default to return version 0 of the DBD by specifying DBLEVEL=BASE in the DFSDFxxx member. Unless the system default is changed to DBLEVEL=BASE, application programs that use a version other than the current active version will require a change to either:

- Their PSB to specify DBLEVEL=BASE to override the system default
- Their PCB to specify the version that they need on the DBVER parameter
- Their code to issue the INIT VERSION call at run time

Implications of DBLEVEL=BASE

When DBLEVEL=BASE is specified at the system level, the version 0 instance of the DBD for each versioned database must exist in the DBD record in the IMS catalog. If DBLEVEL=BASE is specified and an application program for which a version is not specified attempts to access a database that no longer has a version 0, the application program abends.

If application programs, tools, or products require the current structure of a physical IMS database when DBLEVEL=BASE is specified, override the system default by specifying DBLEVEL=CURR in the PSBs that the application programs, tools, or products use to access IMS databases.

Specify DBLEVEL=CURR if most of your application programs use the currently active database

Use DBLEVEL=CURR when most application programs are modified to use the current database structure, but a few application programs continue to require the structure of an older database version. For application programs that require the older version of a database, the required database version number can be specified on the DBVER= parameter of the PCB.

Data transformation processing for prior versions

Any time application programs requires data in a format defined by a DBD version other than the current, active DBD version, IMS must transform the data from the current format to the format required by the application program. Consequently, if you run with DBLEVEL=BASE or you have a significant number of application programs that use versions of the database other than the current, active version of the DBD, it could increase CPU usage.

When application programs use the current active version of a DBD, IMS does not perform any data transformation, because the data is returned to the application programs in the format in which it is stored on DASD.

Related tasks:

"Implementing database versioning"

Related reference:

↪ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

↪ Program Specification Block (PSB) Generation utility (System Utilities)

Implementing database versioning

To maintain and access multiple versions of a database, you must enable database versioning, define the new version of the database, and generate the necessary DBD, PSB, and ACB members.

Prerequisite: The IMS catalog must be enabled in your IMS system to support database versioning. See IMS catalog definition and tailoring (System Definition).

To enable and implement database versioning, perform the following steps.

1. Enable database versioning by specifying DBVERSION=Y in the database section of the DFSDFxxx member in the IMS.PROCLIB data set.

2. Optional: On the DBLEVEL= parameter in the DFSDFxxx member, set the system default for which version of the database IMS uses when application programs do not specify a version. Valid values are CURR and BASE. CURR returns data by using the version that is currently active in the IMS catalog. This is the default. BASE returns data by using version 0 of the database. Version 0 is the version of a database that existed prior to implementing versioning for the database.

3. Code the changes to the database in the input macros to the DBD Generation utility:

   a. Specify the new version number on the DBVER keyword of the DBD statement Version numbers are integers that must increase in value by one when you define a new version.

   b. If you are changing the length of any segments, code the new length on the BYTES keyword of the SEGM statement.

   c. If you are defining new fields, code the FIELD statements. The starting offsets and lengths of existing fields cannot be changed.

4. Code the versions that the application programs require in the input macros to the PSB Generation utility.

   Note: If multiple database PCBs in a PSB reference the same database, all of the PCBs must specify the same version of the DBD.

a. Optionally, specify a default version level to return to application programs on the DBLEVEL keyword in one or more PSB statements.

   b. As necessary, specify specific versions to return to application programs on the DBVER keyword in the PCB statements.

   c. Generate the PCBs by running the PSB Generation utility.

5. If you are not using an online alter function to apply the database changes to the online database, unload the database offline by using the DBD member for the existing database structure.

6. Generate the DBD member for the new database structure by running the DBD Generation utility.

7. If you are not using an online alter function to apply the database changes to the online database, reload the database by using the DBD member for the new database structure.

8. Generate the ACB members for the new database version and update the IMS catalog by running the ACB Generation and Catalog Populate utility (DFS3UACB).

   If the database that you are altering has secondary indexes or is logically related to another database, in addition to specifying a BUILD DBD statement for the altered database, you must also specify a BUILD DBD statement for each secondary index database and each logically related database in the ACB Maintenance utility control statements.

9. If you are using an online alter function to apply the database changes to an online database, start the alter function.

10. Activate the ACB members for the new database structure and the application programs that use the database. In online systems use the Online Change (OLC) function.

11. Start the database.

12. **Important**: Confirm that changes you have made to the database are supported for database versioning by running an application program that accesses a prior version of the database. IMS does not detect unsupported database changes until the first application program that requires an older version is scheduled to run.

Related concepts:

"System default for database versioning" on page 382

Related reference:

⟶ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

⟶ Database Description (DBD) Generation utility (System Utilities)

⟶ Program Specification Block (PSB) Generation utility (System Utilities)

⟶ Application Control Blocks Maintenance utility (System Utilities)

Logical relationships, secondary indexes, and database versioning

IMS database versioning supports both databases that have logical relationships and databases that have secondary indexes, as long as the segments that are involved in the relationship between the two databases, or that are paired with another segment in the same database, do not change in new versions of the DBD.

Only the segments in the databases that are not involved in a relationship can change.

Databases that have a secondary index are supported by database versioning, but only when the base version of the DBD of the indexed database defines the secondary index relationships and those relationships do not change in later versions of the database. The segment and field relationships of a secondary index are defined in the DBD of the indexed database by XDFLD statements. Both the XDFLD statements and the fields in the indexed database that are referenced by the XDFLD statements must remain unchanged in later versions of the indexed database.

When you create a new version of a database that has a direct logical relationship with another database, you must generate new ACB members for both databases at the same time.

The database definitions (DBDs) for both databases must be explicitly specified on the BUILD statement of the ACB Maintenance utility.

DBDs for databases that are only indirectly related to the database that is being modified do not need to be included on the BUILD statement.

For example, if a new version of database A is being created, and database A is related to database B, which in turn is related to database C, only DBDs for databases A and B need to be specified on the BUILD statement.

As another example, if the a new version is being created for database B from the previous scenario, the DBDs for all three databases, A, B, and C, need to be specified on the BUILD statement.

Related concepts:

Chapter 5, "Overview of the IMS catalog," on page 43

Related reference:

➡ Application Control Blocks Maintenance utility (System Utilities)

Chapter 20. Optional database functions

In addition to logical relationships and secondary indexes, which are described in separate topics, IMS databases support a variety of optional functions that you can choose to implement depending on the database type you are using and the needs of your installation.

The optional functions described in this topic do not apply to GSAM, MSDB, HSAM, and SHSAM databases. Only the variable-length segment function, the Segment Edit/Compression exit routine, and the Data Capture exit routine apply to DEDBs.

Related concepts:

"Design review 3" on page 31

"Overview of optional database functions" on page 18

"How SB buffers data" on page 447

Variable-length segments

Variable-length segments are simply segments whose length can vary in occurrence of some segment types.

A database can contain both variable-length segment and fixed-length segment types.

Database types that support variable-length segments:

- HISAM
- HDAM
- PHIDAM
- HIDAM
- PHDAM
- DEDB

Related concepts:

"Replacing segments" on page 140

Related tasks:

"Adding or converting to variable-length segments" on page 725

How to specify variable-length segments

It is the data portion of a variable-length segment whose length varies. The data portion varies between a minimum and a maximum number of bytes.

As shown in the following figure, you specify minimum and maximum size in the BYTES= keyword in the SEGM statement in the DBD. Because IMS needs to know the length of the data portion of a variable-length segment, you include a 2-byte size field in each segment when loading it. The size field is in the data portion of the segment. The length of the data portion you specify must include the two bytes used for the size field. If the segment type has a sequence field, the minimum length specified in the size field must equal at least the size field and all data to the end of the sequence field.
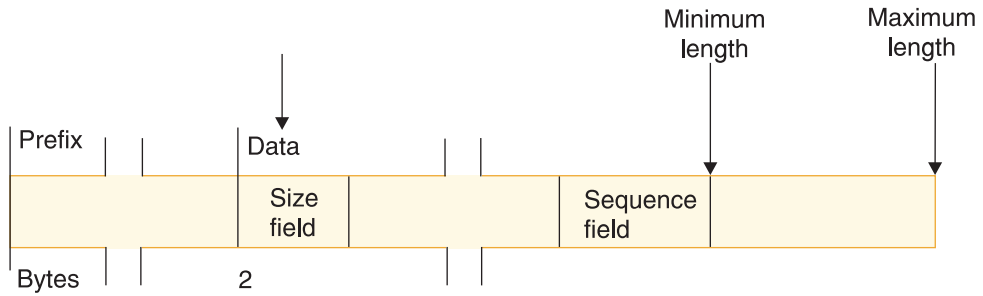
Figure 196. How variable-length segments are specified

How variable-length segments are stored and processed

When a variable-length segment is initially loaded, the space used to store its data portion is the length specified in the MINBYTES operand or the length specified in the size field, whichever is larger.

If the space in the MINBYTES operand is larger, more space is allocated for the segment than is required. The additional space can be used when existing data in the segment is replaced with data that is longer.

The prefix and data portion of HDAM, PHDAM, HIDAM, and PHIDAM variable-length segments can be separated in storage when updates occur. When this happens, the first four bytes following the prefix point to the separated data portion of the segment.

The following figure shows the format of a HISAM variable-length segment. It is also the format of an HDAM, PHDAM, HIDAM, or PHIDAM variable-length segment when the prefix and data portion of the segment have not been separated in storage.
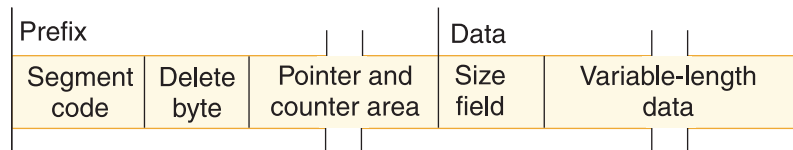
Figure 197. Format of HISAM variable-length segments

The following figure shows the format of an HDAM, PHDAM, HIDAM, or PHIDAM variable-length segment when the prefix and data portion of the segment have been separated in storage.
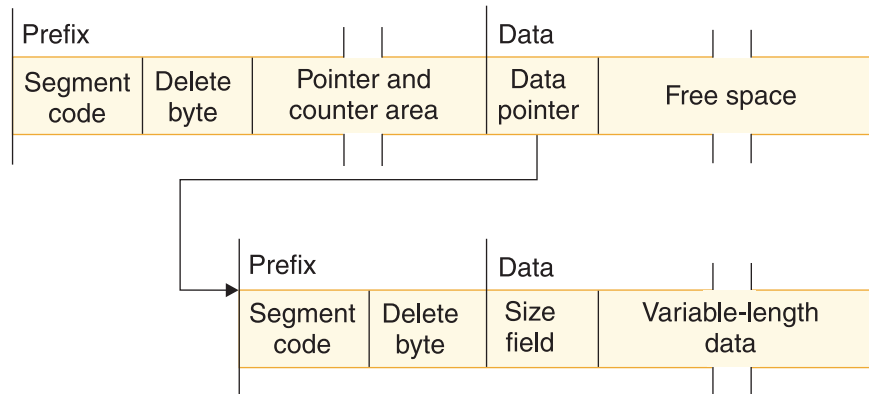
Figure 198. Format of HDAM, PHDAM, HIDAM or PHIDAM variable-length segments

After a variable-length segment is loaded, replace operations can cause the size of data in it to be either increased or decreased. When the length of data in an existing HISAM segment is increased, the logical record containing the segment is rewritten to acquire the additional space. Any segments displaced by the rewrite are put in overflow storage. Displacement of segments to overflow storage can affect performance. When the length of data in an existing HISAM segment is decreased, the logical record is rewritten so all segments in it are physically adjacent.

When a replace operation causes the length of data in an existing HDAM, PHDAM, HIDAM, or PHIDAM segment to be increased, one of two things can happen:

- If the space allocated for the existing segment is long enough for the new data, the new data is simply placed in the segment. This is true regardless of whether the prefix and data portions of the segment were previously separated in the data set.
- If the space allocated for the existing segment is not long enough for the new data, the prefix and data portions of the segment are separated in storage. IMS puts the data portion of the segment as close to the prefix as possible. Once the segment is separated, a pointer is placed in the first four bytes following the prefix to point to the data portion of the segment. This separation increases the amount of space needed for the segment, because, in addition to the pointer kept with the prefix, a 1-byte segment code and 1-byte delete code are added to the data portion of the segment (see Figure 197 on page 388). In addition, if separation of the segment causes its two parts to be stored in different blocks, two read operations will be required to access the segment.

When a replace operation causes the length of data in an existing HDAM, PHDAM, HIDAM, or PHIDAM segment to be decreased, one of three things can happen:

- If prefix and data are not separated, the data in the existing segment is replaced with the new, shorter data followed by free space.
- If prefix and data are separated but sufficient space is not available immediately following the original prefix to recombine the segment, the data in the separated data portion of the segment is replaced with the new, shorter data followed by free space.
- If prefix and data are separated and sufficient space is available immediately following the original prefix to recombine the segment, the new data is placed in

the original space, overlaying the data pointer. The old separated data portion of the segment is then available as free space in HD databases.

When to use variable-length segments

Use variable-length segments when the length of data in your segment varies, for example, with descriptive data.

By using variable-length segments, you do not need to make the data portion of your segment type as long as the longest piece of descriptive data you have. This saves storage space. Note, however, that if you are using HDAM, PHDAM, HIDAM, or PHIDAM databases and your segment data characteristically grows in size over time, segments will split. If a segment split causes the two parts of a segment to be put in different blocks, two read operations will be required to access the segment until the database is reorganized. So variable-length segments work well if segment size varies but is stable (as in an address segment). Variable-length segments might not work well if segment size typically grows (as in a segment type containing a cumulative list of sales commissions).

What application programmers need to know about variable-length segments

If you are using variable-length segments in your database, you need to let application programmers who will be using the database know this.

They need to know which of the segment types they have access to are variable in length and the maximum size of each of these variable-length segment types. In calculating the size of their I/O area, application programmers must use the maximum size of a variable-length segment. In addition, they need to know that the first two bytes of the data portion of a variable-length segment contain the length of the data portion including the size field.

Working with the application programmer, you should devise a scheme for accessing data in variable-length segments.

Segment Edit/Compression exit routine

The Segment Edit/Compression exit routine allows you to encode, edit, or compress the data portion of a segment.

You can use this facility on segment data in full function databases and Fast Path DEDBs. You write the routine (your edit routine) that actually manipulates the data in the segment. The IMS code gives your edit routine information about the segment's location and assists in moving the segment back and forth between the buffer pool and the application program's I/O area.

The following database types support the Segment Edit/Compression exit routine:
- HISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM
- DEDB

Detailed information on how the Segment Edit/Compression exit routine works and how you use it is in *IMS Version 15 Exit Routines*.

The Segment Edit/Compression exit routine lets you:

- **Encode data for security purposes.** Encoding data consists of "scrambling" segment data when it is on the device so only programs with access to the edit routine can see it in decoded form.
- **Edit data.** Editing data allows application programs to receive data in a format other than the one in which it is stored. For example, an application program might receive segment fields in an order other than the one in which they are stored; an application program might require all blank space be removed from descriptive data.
- **Compress data.** This allows better use of DASD storage because segments can be compressed when written to the device and then expanded when passed back to the application program. Segment data might be compressed, for example, by removing all blanks and zeros.
- **Expand Data.** The DEDB Sequential Dependent Scan utility invokes the Segment Edit/Compression exit routine (DFSCMPX0) to expand compressed SDEP segments when you specify both SDEP segment compression in the DBD and the DEDB Scan utility keyword, EXPANDSEG.

  Related Reading: EXPANDSEG and the DEDB Scan utility are described in *IMS Version 15 Database Utilities*. The segment compression exit is described in *IMS Version 15 Exit Routines*.

Two types of segment manipulation are possible using the Segment Edit/Compression exit routine:

- **Data compression**— movement or compression of data within a segment in a manner that does not alter the content or position of the key field. Typically, this involves compression of data from the end of the key field to the end of the segment. When a fixed-length segment is compressed, a 2-byte field must be added to the beginning of the data portion of the segment by the user data compression routine. This field is used by IMS to determine secondary storage requirements and is the only time that the location of the key field can be altered. The segment size field of a variable-length segment cannot be compressed but must be updated to reflect the length of the compressed segment.
- **Key compression**— movement or compression of any data within a segment in a manner that can change the relative position, value, or length of the key field and any other fields except the size field. The segment size field of a variable-length segment must be updated by the compression routine to reflect the length of the compressed segment.

You specify the use of the Segment Edit/Compression exit routine when you define a segment type. Any segment type can be edited or compressed (using either data or key compression) as long as the segment is:

- Not a logical child
- Not in an HSAM, SHISAM, or index database

The use of the segment edit/compression exit routine is defined in physical database DBDs. This exit routine's use cannot be defined in a logical database DBD.

Data compression is allowed but key compression is not allowed when the segment is:

- A root segment in a HISAM database
- A segment in a DEDB database

Related tasks:

"Converting to the Segment Edit/Compression exit routine" on page 756

"Method 2. Converting segments or a database" on page 726

"Encrypting your database" on page 38

"Exit routine modifications and HALDB databases" on page 836

Considerations for using the Segment Edit/Compression exit routine

Before using a Segment Edit/Compression exit routine, you should be aware several points.

The points you should be aware of include:
- Because your edit routine is executed as part of a DL/I call, if it abnormally terminates so does the entire IMS region.
- Your routine cannot use the operating system macros LOAD, GETMAIN, SPIE or STAE.
- The name of the Segment Edit/Compression exit routine must not be the same as the DBDNAME.
- Editing and compressing of each segment on its way to or from an application program requires additional processor time.
- If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

Depending on the options you select, search time to locate a specific segment can increase. If you are fully compressing the segment using key compression, every segment type that is a candidate to satisfy either a fully qualified key or data field request must be expanded or divided. IMS then examines the appropriate field. For key field qualification, only those fields from the start of the segment through the sequence field are expanded during the search. For data field qualification, the total segment is expanded. In the case of data compression and a key field request, little more processing is required to locate the segment than that of non-compressed segments. Only the segment sequence field is used to determine if this segment occurrence satisfies the qualification.

Other considerations can affect total system performance, especially in an online environment. For example, being able to load an algorithm table into storage gives the compression routine a large amount of flexibility. However, this can place the entire IMS control region into a wait state until the requested member is present in storage. It is suggested that all alternatives be explored to lessen the impact of situations such as this.

Related reference:

➡ Segment edit/compression exit routines (Exit Routines)

Preventing split segments from impacting performance

Split segments can negatively affect performance by requiring additional reads to retrieve both parts of the segments.

When segments are split, their prefixes remain in their existing location, but their data parts are stored in a new location, possibly in another block or CI. Replace calls can split the segments when segments in a full-function database grow larger than the size of their current location.

To prevent IMS from splitting compressed segments, you can specify a minimum size for the segments that includes extra padded space. This gives the compressed segment room to grow and decreases the chance that IMS will split the segment.

You specify the minimum size for fixed-length full-function segments differently than you do for variable-length full-function segments:

- For fixed-length segments, specify the minimum size using both the fourth and fifth subparameters on the COMPRTN= parameter of the SEGM statement. The fourth subparameter, *size*, only defines the minimum size if you also specify the fifth subparameter, PAD.
- For variable-length segments, specify the minimum size using the second subparameter, *min_bytes*, of the BYTES= parameter of the SEGM statement.

DEDB segments are never split by replace calls. If a DEDB segment grows beyond the size of its current location, the entire segment, including its prefix, is moved to a new location. For this reason, it is not necessary to pad compressed DEDB segments.

Related reference:

➡ SEGM statements (System Utilities)

Specifying the Segment Edit/Compression exit routine

To specify the use of the Segment Edit/Compression exit routine for a segment, use the COMPRTN= keyword of the SEGM statement in the DBD.

Related reference:

➡ SEGM statements (System Utilities)

Data Capture exit routines

Data Capture exit routines capture segment-level data from a DL/I database for propagation to Db2 for z/OS databases. Installations running IMS and Db2 for z/OS databases can use Data Capture exit routines to exchange data across the two database types.

The Data Capture exit routine is an installation-written exit routine. Data Capture exit routines promote and enhance database coexistence.

The following database types support data capture exit routines:
- HISAM
- SHISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM

- DEDB

Data Capture exit routines can be written in assembler language, C, COBOL, or PL/I.

Data Capture exit routines are supported by IMS Transaction Manager and Database Manager. DBCTL support is for BMPs only.

Data Capture exit routines do not support segments in secondary indexes.

A Data Capture exit routine is called based on segment-level specifications in the DBD. When a Data Capture exit routine is specified on a database segment, it is invoked by all application program activity on that segment, regardless of which PSB is active. Therefore, Data Capture exit routines are global. Using a Data Capture exit routine can have a performance impact across the entire database system.

Related tasks:

"Converting databases for Data Capture exit routines and Asynchronous Data Capture" on page 756

"Exit routine modifications and HALDB databases" on page 836

Related reference:

Data Capture exit routine (Exit Routines)

DBD parameters for Data Capture exit routines

Using Data Capture exit routines requires specification of one or two DBD parameters and subsequent DBDGEN.

This topic contains Product-sensitive Programming Interface information.

The EXIT= parameter identifies which Data Capture exit routines will run against segments in a database. The VERSION= parameter records important information about the DBD for use by Data Capture exit routines.

The EXIT= parameter

To use the Data Capture exit routine, you must use the optional EXIT= parameter in the DBD statement or SEGM statement. You specify EXIT= on either the DBD or SEGM statements of physical database definitions.

Specifying EXIT= on the DBD statement applies a Data Capture exit routine to all segments within a database structure. Specifying EXIT= on the SEGM statement applies a Data Capture exit routine to only that segment type.

You can override Data Capture exit routines specified on the DBD statement by specifying EXIT= on a SEGM statement. EXIT=NONE on a SEGM statement cancels all Data Capture exit routines specified on the DBD statement for that segment type. A physical child does not inherit an EXIT= parameter specified on the SEGM statement of its physical parent.

You can specify multiple Data Capture exit routines on a single DBD or SEGM statement. For example, you might code a DBD statement as:

```
DBD    EXIT=((EXIT1A),(EXIT1B))
```

The name of the Data Capture exit routine that you intend to use is the only required operand for the EXIT= parameter. Exit names can have a maximum of eight alphanumeric characters. For example, if you specify a Data Capture exit routine with the name EXITA on a SEGM statement in a database, the EXIT= parameter is coded as follows:

```
SEGM    EXIT=(EXITA,KEY,DATA,NOPATH,DLET,BEFORE,(CASCADE,KEY,DATA,NOPATH,DLET,BEFORE))
```

KEY, DATA, NOPATH, DLET, BEFORE, CASCADE, KEY, DATA, NOPATH, DLET, and BEFORE are default operands. These defaults define what data is captured by the exit routine when a segment is updated by an application program.

The VERSION= parameter

VERSION= is an optional parameter that supports Data Capture exit routines. VERSION= is specified on the DBD statement as:

```
VERSION='character string'
```

The maximum length of the character string is 255 bytes. You can use VERSION= to create a naming convention that denotes the database characteristics that affect the proper functioning of Data Capture exit routines. You might use VERSION= to flag DBDs containing logical relationships, or to indicate which data capture exit routines are defined on the DBD or SEGM statements. VERSION= might be coded as:

```
DBD    VERSION='DAL-&SYSDATE-&SYSTIME'
```

DAL, in this statement, tells you that Data Capture exit routine A is specified on the DBD statement (D), and that the database contains logical relationships (L). &SYSDATE and &SYSTIME tell you the date and time the DBD was generated.

If you do not specify a VERSION= parameter, DBDGEN generates a default 13-character date-time stamp. The default consists of an 8-byte date stamp and a 5-byte time stamp with the following format:

```
MM/DD/YYHH.MM
```

The default date-time stamp on VERSION= is identical to the DBDGEN date-time stamp.

VERSION= is passed as a variable length character string with a 2-byte length of the VERSION=, which does not include the length of the LL.

Related reference:

DBD statements (System Utilities)

SEGM statements (System Utilities)

"CAPXSEGM segment type format" on page 66

"CAPXDBD segment type format" on page 65

Call sequence of Data Capture exit routines

A Data Capture exit routine is invoked once per segment update for each segment for which the Data Capture exit routine is specified. Data Capture exit routines are invoked multiple times for a single call under certain conditions.

This topic contains Product-sensitive Programming Interface information.

The conditions in which a Data Capture exit routine is invoked multiple times for a single call include:

- Path updates.
- Cascade deletes when multiple segment types or multiple segment occurrences are deleted.
- Updates on logical children.
- Updates on logical parents.
- Updates on a single segment when multiple Data Capture exit routines are specified against that segment. Each exit is invoked once, in the order it is listed on the DBD or SEGM statements.

When multiple segments are updated in a single application program call, Data Capture exit routines are invoked in the same order in which IMS physically updates the segments:

1. Path inserts are executed "top-down" in DL/I. Therefore, a Data Capture exit routine for a parent segment is called before a Data Capture exit routine for that parent's dependent.
2. Cascade deletes are executed "bottom-up". All dependent segments' exits are called before their respective parents' exits on cascade deletes. IMS physically deletes dependent segments on cascade deletes only after it has validated the delete rules by following the hierarchy to the lowest level segment. After delete rules are validated, IMS deletes segments starting with the lowest level segment in a dependent chain and continuing up the chain, deleting the highest level parent segment in the hierarchy last. Data Capture exit routines specified for segments in a cascade delete are called in reverse hierarchical order.
3. Path replaces are performed "top-down" in IMS. In Data Capture exit routines defined against segments in path replaces, parent segments are replaced first. All of their descendents are then replaced in descending hierarchical order.

When an application program does a cascade delete on logically related segments, Data Capture exit routines defined on the logical child are always called before Data Capture exit routines defined on the logical parent. Data Capture exit routines are called even if the logical child is higher in the physical hierarchy, except in recursive structures where the delete results in the deletion of a parent of the deleted segment.

Data passed to and captured by the Data Capture exit routine

Data is passed to Data Capture exit routines when an application program updates IMS with a DL/I insert, delete, or replace call. If the SSPCMD keyword is specified, this data will also include subset pointer command code (M, S, W, Z) updates as well the R command code (DEDB only).

This topic contains Product-sensitive Programming Interface information.

If the SSPCMD keyword is specified, data will also be captured for Get-type calls that update subset pointers. The R command code will be captured, too, if specified. Similarly, data is captured for non-updated segments of the specified path in insert or replace calls that update subset pointers.

If the INPOS keyword is specified and a non-unique segment is inserted with an ISRT rule of HERE, segment data is captured for the following twin, if it exists.

Segment data passed to Data Capture exit routines is always physical data. When the update involves logical children, the data passed is physical data and the concatenated key of the logical parent segment. For segments that use the Segment Edit/Compression exit routine (DFSCMPX0), the data passed is expanded data.

When an application replaces a segment, both the existing and the replacement physical data are captured. In general, segment data is captured even if the application call does not change the data. However, for full-function databases, IMS compares the before and after data. If the data has not changed, IMS does not update the database or log the replace data. Because data is not replaced, Data Capture exit routines specified for that segment are not called and the data is not captured.

Data might be captured during replaces even if segment data does not change when:
1. The application inserts a concatenation of a logical child and logical parent, IMS replaces the logical parent, and the parent data does not change.
2. The application issues a replace for a segment in a DEDB database.
3. If the SSPCMD keyword is specified, non-updated segments of the specified path that contain subset pointer updates (DEDB only).

In each case, IMS updates the database without comparing the before and after data, and therefore the data is captured even though it does not change.

Recommendation: The entire segment, before and after, is passed to Data Capture exit routines when the application replaces a segment. When the exit routine is interested in only a few fields, do not issue the SQL update request until after the before and after replace data for those fields is compared to see if the fields were changed.

Data Capture call functions

Data Capture exit routines are called when segment data is updated by an application program insert, replace, or delete call. If subset pointer updates are to be captured (SSPCMD), get calls will also be captured.

This topic contains Product-sensitive Programming Interface information.

Optionally, Data Capture exit routines are called when DL/I deletes a dependent segment because the application program deleted its parent segment, a process that is known as cascade delete. Data Capture exit routines are passed two functions to identify the following:
1. The action performed by the application program
2. The action performed by IMS

The two functions that are passed to the Data Capture exit routines are:
- Call function. The DL/I call, ISRT, REPL, Gx (get-type), or DLET, that is issued by the application program for the segment.
- Physical function. The physical action, ISRT, REPL, or DLET, performed by IMS as a result of the call. Physical function SSPU is set for segments that are not updated but have subset pointers updates. The physical function is used to determine the type of SQL request to issue when propagating data.

The call and physical functions passed to the exit routine are always the same for replace calls. However, the functions passed might differ for delete or insert calls:

- For delete calls resulting in cascade deletes, the call function passed is CASC (to indicate the cascade delete) and the physical function passed is DLET.
- For insert calls resulting in the insert of a logical child and the replace of a logical parent (because the logical parent already exists), the call function that is passed is ISRT and the physical function that is passed is REPL. IMS physically replaces the logical parent with data inserted by the application program even if the parent data does not change. Both call and physical functions are then used, based on the data propagation requirements, to determine the SQL request to issue in the Data Capture exit routine.

Cascade delete when crossing logical relationships

If the EXIT= options specify NOCASCADE, data is not captured for cascade deletes. However, when a cascade delete crosses a logical relationship into another physical database to delete dependent segments, a Data Capture exit routine needs to be called.

This topic contains Product-sensitive Programming Interface information.

The Data Capture exit routine needs to be called in order to issue the SQL delete for the parent of the physical structure in Db2 for z/OS. Rather than requiring the EXIT= CASCADE option, IMS always calls the exit routine for a segment when deleting the parent segment in a physical database record with an exit routine defined, regardless of the CASCADE/NOCASCADE option specified on the segment. IMS bypasses the NOCASCADE option only when crossing logical relationships into another physical database. As with all cascade deletes, the call function passed is CASC and the physical function passed is DLET.

Data Capture exit routines and logically related databases

Segment data passed to Data Capture exit routines is always physical data. Consequently, you must place restrictions on delete rules in logically related databases supporting Data Capture exit routines.

This topic contains Product-sensitive Programming Interface information.

The following table summarizes which delete rules you can and cannot use in logically related databases with Data Capture exit routines specified on their segments.

Table 63. Delete rule restrictions for logically related databases using Data Capture exit routines

Segment type	Virtual delete rule	Logical delete rule	Physical delete rule
Logical Children	Yes	No	No
Logical Parents	No	Yes	Yes

When a logically related database has a delete rule violation on a logical child:
- The logical child cannot have a Data Capture exit routine specified.
- No ancestor of the logical child can have a Data Capture exit routine specified.

When a logically related database has a delete rule violation on a logical parent, the logical parent cannot have a Data Capture exit routine specified. ACBGEN validates logical delete rule restrictions and will not allow a PSB that refers to a database that violates these restrictions to proceed.

Field-level sensitivity

Field-level sensitivity provides a number of benefits related to the data-independence of application programs, enhanced data security, and increased flexibility in the formatting of segment types.

Field-level sensitivity gives you an increased level of data independence by isolating application programs from:
- Changes in the arrangement of fields within a segment
- Addition or deletion of data within a segment

In addition, field-level sensitivity enhances data security by limiting an application program to a subset of fields within a segment, and controlling replace operations at the field level.

Field-level sensitivity allows you to reformat a segment type. Reformatting a segment type can be done without changing the application program's view of the segment data, provided fields have not been removed or altered in length or data type. Fields can be added to or shifted within a segment in a manner transparent to the application program. Field-level sensitivity gives applications a segment organization that always conforms to what is specified in the SENFLD statements.

The following database types support field-level sensitivity:
- HSAM
- HISAM
- SHISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM

Using field-level sensitivity as a mapping interface

Field-level sensitivity acts as a mapping interface by letting PSBGEN field locations differ from DBDGEN field locations. Mapping is invoked after the segment edit routine on input and before the segment edit routine on output. When creating a sequential data set from database information (or creating database information from a sequential data set), field-level sensitivity can reduce or eliminate the amount of formatting an application program must do.

Using field-level sensitivity with variable-length segments

If field-level sensitivity is used with variable-length segments, you can add new fields to a segment without reorganizing the database. FIELD definitions in a DBDGEN allow you to enlarge segment types without affecting any previous users of the segment. The DBDGEN FIELD statement lets you specify a field that does not yet exist in the physical segment but that will be dynamically created when the segment is retrieved.

Field-level sensitivity can help in the transition of an application program from a non-database environment to a database environment. Application programs that formerly accessed z/OS files might be able to receive the same information in the same format if the database was designed with conversion in mind.

Field-level sensitivity is *not* supported for DEDBs and MSDBs.

Related tasks:

"Changing the position of data in a segment" on page 726

How to specify use of field-level sensitivity in the DBD and PSB

An application program's view of data is defined through the PSBGEN utility using SENFLD statements following the SENSEG statement.

In the SENFLD statement, the NAME= parameter identifies a field that was defined in the segment through the DBDGEN utility. The SENFLD statement is not supported by Fast Path secondary indexing.

The START= parameter defines the starting location of the field in the application program's I/O area. In the I/O area, fields do not need to be located in any particular order, nor must they be contiguous. The end of the segment in the I/O area is defined by the end of the right most field. All segments using field-level sensitivity appear fixed in length in the I/O area. The length is determined by the sum of the lengths of fields on SENFLD statements associated with a SENSEG statement.

The following figure shows an example of field-level sensitivity. After the figure is information about coding field-level sensitivity.

Field-level sensitivity is used below to reposition three fields from a physical segment in the application program's I/O area.
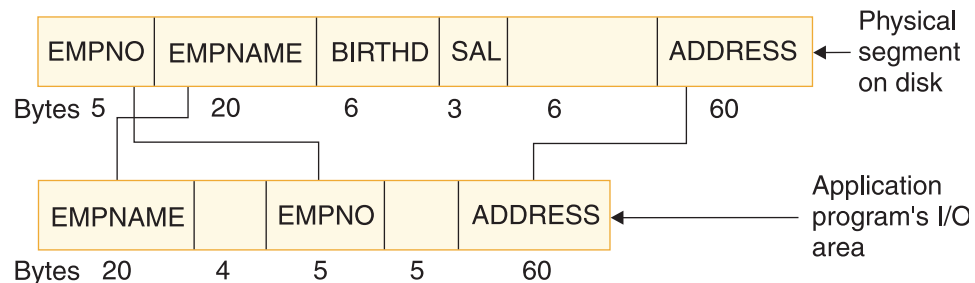
Figure 199. DBD and PSB coding for field-level sensitivity

The following code shows the DBD example for field-level sensitivity shown in the figure above.

```
SEGM  NAME=EMPREC,BYTES=100
 FIELD NAME=(EMPNO,SEQ),BYTES=5,START=1,TYPE=C
 FIELD NAME=EMPNAME,BYTES=20,START=6,TYPE=C
 FIELD NAME=BIRTHD,BYTES=6,START=26,TYPE=C
 FIELD NAME=SAL,BYTES=3,START=32,TYPE=P
 FIELD NAME=ADDRESS,BYTES=60,START=41,TYPE=C
```

The following code shows the PSB for the preceding figure.

```
SENSEG NAME=EMPREC,PROCOPT=A
 SENFLD NAME=EMPNAME,START=1,REPL=N
 SENFLD NAME=EMPNO,START=25
 SENFLD NAME=ADDRESS,START=35,REPL=Y
```

A SENFLD statement is coded for each field that can appear in the I/O area. A maximum of 255 SENFLD statements can be coded for each SENSEG statement, with a limit of 10000 SENFLD statements for a single PSB.

The optional REPL= parameter on the SENFLD statement indicates whether replace operations are allowed on the field. In the figure, replace is not allowed for EMPNAME but is allowed for EMPNO and ADDRESS. If REPL= is not coded on a SENFLD statement, the default is REPL=Y.

The TYPE= parameter on FIELD statements in the DBD is used to determine fill values on insert operations.

Retrieving segments using field-level sensitivity

When you retrieve segments using field-level sensitivity, you should be aware of the following information.
- Gaps between fields in the I/O area are set to blanks on a retrieve call.
- If an application program uses a field in an SSA, that field must be coded on a SENFLD statement. This rule does not apply to sequence fields used in an SSA on retrieve operations.

The following figure shows an example of a retrieve call based on the DBD and PSB in "How to specify use of field-level sensitivity in the DBD and PSB" on page 400.

Figure 200. Example of a retrieve call

Replacing segments using field-level sensitivity

The SENFLD statement must allow replace operations (REPL=Y) if the application program is going to replace data in a segment.

In the example shown in "How to specify use of field-level sensitivity in the DBD and PSB" on page 400, the SENFLD statement for EMPNAME specifies REPL=N. A "DA" status code would be returned if the application program tried to replace the EMPNAME field. The following figure shows an example of a REPL call based on the DBD and PSB in "How to specify use of field-level sensitivity in the DBD and PSB" on page 400.

Figure 201. Example of a REPL call

Inserting segments using field-level sensitivity

The TYPE= parameter on the SEGM statement of the DBD determines the fill value in the physical segment when an application program is not sensitive to a field on insert calls.

TYPE= **Fill Value**

X   Binary Zeros

P   Packed Decimal Zero

C   Blanks

The fill value in the physical segment is binary zeros when:
- Space in a segment is not defined by a FIELD macro in the DBD
- A defined DBD field is not referenced on the insert operation

The following figure shows an example of an insert operation based on the DBD and PCB in Figure 199 on page 400.

I/O area

EMPNAME ADAMS, DICK		EMPNO 23456		ADDRESS VERMONT
1 20		25 29		35 94

Bytes

Physical segment on disk after update

EMPNO 23456	EMPNAME ADAMS, DICK	BIRTHD ɓ←→ɓ	SAL 00+	0←→0	ADDRESS VERMONT
1 5	6 25	26 31	32 34		41 100

Bytes

Figure 202. Example of an ISRT call

Blanks are inserted in the BIRTHD field because its FIELD statement in the DBD specifies TYPE=C. Packed decimal zero is inserted in the SAL field because its FIELD statement in the DBD specifies TYPE=P. Binary zeros are inserted in positions 35 to 40 because no FIELD statement was coded for this space in the DBD.

Using field-level sensitivity when fields overlap

On the SENFLD statement, you code the starting position of fields as they will appear in the I/O area.

If fields overlap in the I/O area, here are the rules you must follow:
- Two different bytes of data cannot be moved to the same position in the I/O area on input.
- The same data can be moved to different positions in the I/O area on retrieve operations.
- Two bytes from different positions in the I/O area cannot be moved to the same DBD field on output.

Using field-level sensitivity when path calls are issued

If an application program issues path calls while using field level sensitivity, you must follow some rules.

Here are the rules you must follow:
- You should not code SENFLD statements so that two fields from different physical segments are in the same segment in the I/O area.
- PROCOPT=P is required on the PCB statement.

Using field-level sensitivity with logical relationships

You must follow some rules when using field-level sensitivity with segments that are involved in a logical relationship

Here are the rules you must follow:
- Application programs can not be insert sensitive to a logical child.

- The same field can be referenced in more than one SENFLD statement within a SENSEG. If the duplicate field names are part of a concatenated segment and the same field name appears in both parts of the concatenation, the first part references the logical child. The second and all subsequent parts reference the logical parent. This referencing sequence determines the order in which fields are moved to the I/O area.
- When using field-level sensitivity with a virtual logical child, the field list of the paired segment is searched after the field list of the virtual segment and before the field list of the logical parent.

Using field-level sensitivity with variable-length segments

When field-level sensitivity is used with a variable-length segment, an application program's view of the segment is fixed in length and does not include the 2-byte length field.

This topic and its subtopics address special situations when field level sensitivity is used with variable-length segments. First, however, here is some general information about using field-level sensitivity with variable-length segments:

- When inserting a variable-length segment, the length used is the minimum length needed to hold all sensitive fields.
- When replacing a variable-length segment, if the length has to be increased to contain data an application program has modified, the length used is the minimum length needed to hold the modified data.
- An application program cannot be sensitive to overlapping fields in a variable-length segment with get or update sensitivity if the data type of any of those fields is not character.
- Existing programs processing variable-length segments that use the length field to determine the presence or absence of a field might need to be modified if segments are inserted or updated by programs using field-level sensitivity.

When field-level sensitivity is used with variable-length segments, two situations exist that you should know about. The first is when fields are missing. The second is when fields are partially present. This topic examines the following information:

- Retrieving Missing Fields
- Replacing Missing Fields
- Inserting Missing Fields
- Retrieving Partially Present Fields
- Replacing Partially Present Fields

Retrieving missing fields

If a field does not exist in the physical variable-length segment at retrieval time, the corresponding field in the application program's I/O area is filled with a value based on the data type specified in the DBD.

The figure below is an example of a missing field on a retrieve call based on the DBD and PSB examples that follow the figure.

Figure 203. Example of a missing field on a retrieve call

The following code is an example DBD for field-level sensitivity with variable-length segments.

DBD

```
SEGM  NAME=EMPREC,BYTES=(102,7)
FIELD NAME=(EMPNO,SEQ),BYTES=5,START=3,TYPE=C
FIELD NAME=EMPNAME,BYTES=20,START=8,TYPE=C
FIELD NAME=BIRTHD,BYTES=6,START=28,TYPE=C
FIELD NAME=ADDRESS,BYTES=60,START=43,TYPE=C
```

The following code is an example PSB for field-level sensitivity with variable-length segments.

PSB

```
SENSEG  NAME=EMPREC,PROCOPT=A
SENFLD  NAME=EMPNAME,START=1,REPL=N
SENFLD  NAME=EMPNO,START=25
SENFLD  NAME=ADDRESS,START=35,REPLY=Y
```

The length field is not present in the I/O area. Also, the address field is filled with blanks, because TYPE=C is specified on the FIELD statement in the DBD.

Related concepts:

"Replacing missing fields"

"Replacing partially present fields" on page 408

Replacing missing fields

A missing field that is not replaced does not affect the physical variable-length segment.

The following figure is an example of a missing field on a replace call based on the DBD and PSB in "Retrieving missing fields" on page 404.

Figure 204. First example of a missing field on a replace call

The length field, maintained by IMS, does not include room for the address field, because the field was missing and not replaced.

On a replace call, if a field returned to the application program with a fill value is changed to a non-fill value, the segment length is increased to the minimum size needed to hold the modified field.

- The 'LL' field is updated to include the full length of the added field and all fields up to the added field.
- The TYPE= parameter in the DBD determines the fill value for non-sensitive DBD fields up to the added field.
- Binary zero is the fill value for space up to the added field that is not defined by a FIELD statement in the DBD.

The following figure is an example of a missing field on a replace call based on the DBD and PSB in "Retrieving missing fields" on page 404.

Physical segment on disk before update

LL 27	EMPNO 12345	EMPNAME SMITH, JOE
1 3	3 7	8 27

Bytes

User I/O area

EMPNAME SMITH, JOE	⌿←→⌿	EMPNO 12345	⌿←→⌿	ADDRESS NEW YORK
1 20		25 29		35 94

Bytes

Physical segment on disk after update

LL 27	EMPNO 12345	EMPNAME SMITH, JOE	BIRTHD ⌿←→⌿	0←→0	ADDRESS NEW YORK
1 2	3 7	8 27	28 33	34 42	43 102

Bytes

Figure 205. Second example of a missing field on a replace call

The 'LL' field is maintained by IMS to include the full length of the ADDRESS field and all fields up to the ADDRESS field. BIRTHD is filled with blanks, because TYPE=C is specified on the FIELD statement in the DBD. Positions 34 to 42 are set to binary zeros, because the space was not defined by a FIELD statement in the DBD.

Related concepts:

"Retrieving missing fields" on page 404

Inserting missing fields

When a variable-length segment is inserted into the database, the length field is set to the value of the minimum size needed to hold all sensitive fields.

- The 'LL' field is updated to include all sensitive fields.
- The TYPE= parameter on the DBD (see the example of a DBD for field-level sensitivity with variable-length segments) determines the fill value for non-sensitive DBD fields.
- Binary zero is the fill value for space not defined by a FIELD statement in the DBD.

The following figure is an example of a missing field on an insert call using the DBD and PSB in the example of a DBD for field-level sensitivity with variable-length segments.

User I/O area					
EMPNAME ADAMS, DICK	ᵇ←→ᵇ	EMPNO 23456	ᵇ←→ᵇ	ADDRESS VERMONT	
1 20		25 29		35 94	

Bytes

Physical segment on disk after insert						
LL 102	EMPNO 23456	EMPNAME ADAMS, DICK	BIRTHD ᵇ←→ᵇ	0←→0	ADDRESS VERMONT
1 2	3 7	8 27	28 33	34 42	43 102

Bytes

Figure 206. Example of a missing field on an insert call

The 'LL' field is maintained by IMS to include the full length of all sensitive fields up to and including the ADDRESS field. BIRTHD is filled with blanks, because TYPE=C was specified on the FIELD statement in the DBD. Positions 34 to 42 are set to binary zeros, because the space was not defined in a FIELD statement in the DBD.

Retrieving partially present fields

If the last field in the physical variable-length segment at retrieval time is only partially present and if the data type is character (TYPE=C), data is returned to the application program padded with blanks on the right. Otherwise, the field is returned with a fill value based on the data type.

The following figure is an example of a partially present field on a retrieval call based on the DBD and PSB in "Retrieving missing fields" on page 404.

Physical segment on disk					
LL 27	EMPNO 12345	EMPNAME SMITH, JOE	BIRTHD ᵇ←→ᵇ		ADDRESS NEW YORK
1 2	3 7	8 27	28 33		43 102

Bytes

User I/O area					
EMPNAME SMITH, JOE	ᵇ←→ᵇ	EMPNO 12345	ᵇ←→ᵇ	ADDRESS NEW YORK	ᵇ←→ᵇ
1 20		25 29		35 94	

Bytes

Figure 207. Example of a partially present field on a retrieval call

The ADDRESS field in the I/O area is padded with blanks to correspond to the length defined on the SEGM statement in the DBD.

Replacing partially present fields

When replacing partially present fields be aware of the following points.

- If segment length is increased on a REPL call, the field returned to the application program is written to the database if it has not been changed.
- If the data type of the field is character and the field is changed on a REPL call, the segment length is increased if necessary to include all non-blank characters in the changed data.
- If the data type is not character and the field is changed on a REPL call, the segment length is increased to contain the entire field.

The following figure is an example of a partially present field on a REPL call based on the DBD and PSB in "Retrieving missing fields" on page 404.

Physical segment on disk before update

LL 50	EMPNO 12345	EMPNAME SMITH, JOE	BIRTHD 480207		ADDRESS NEW YORK

1   2 3   7 8   27 28   33   43   50
Bytes

I/O area

EMPNAME SMITH, JOE	↤→	EMPNO 12345	↤→	ADDRESS NEW YORK

1   20   25   29   35   94
Bytes

Physical segment on disk after update

LL 52	EMPNO 12345	EMPNAME SMITH, JOE	BIRTHD 480207		ADDRESS NEW YORK

1   2 3   7 8   27 28   33   43   52
Bytes

Figure 208. Example of a partially present field on a REPL call

The 'LL' field is changed from 50 to 52 by DL/I to accommodate the change in the field length of ADDRESS.

Related concepts:

"Retrieving missing fields" on page 404

General considerations for using field-level sensitivity

When you use field-level sensitivity, be aware of the following general considerations.

- Field-level sensitivity is not supported for GSAM, MSDB, or DEDB databases.
- Fields referenced in PSBGEN with SENFLD statements must be defined in DBDGEN with FIELD statements.
- The same DBD field can be referenced in more than one SENFLD statement.
- When using field-level sensitivity, the application program always sees a fixed length segment for a given PCB, regardless of whether the segment is fixed or variable.
- Application programs must be sensitive to any field referenced in an SSA, except the sequence field.

- Application programs must be sensitive to the sequence field, if present, for insert or load.
- Field-level sensitivity and segment level sensitivity can be mixed in the same PCB.
- Non-referenced, non-defined fields are set to binary zeros as fill characters, when required, during insert or replace operations.
- Using call/trace with the compare option increases the amount of storage required in the PSB work pool.

Multiple data set groups

HD databases can be stored on multiple data sets; that is, the HD databases can be stored on more than the one or two data sets required for database storage.

The following database types support multiple data set groups:
- HDAM
- PHDAM
- HIDAM
- PHIDAM

When storing a database on multiple data sets, the terms primary and secondary data set group are used to distinguish between the one or more data sets that must be specified for the database (called the primary data set group) and the one or more data sets you are allowed to specify for the database (called secondary data set groups).

In HD databases, a single data set is used for storage rather than a pair of data sets. The primary data set group therefore consists of the ESDS (if VSAM is being used) or OSAM data set (if OSAM is being used) on which you must specify storage for your database. The secondary data set group is an additional ESDS or OSAM data set on which you are allowed to store your database.

As many as ten data set groups can be used in HD databases, that is, one primary data set group and a maximum of nine secondary data set groups.

Related tasks:

When to use multiple data set groups

When you design database records, you design them to meet the processing requirements of many applications. You decide what segments will be in a database record and their hierarchical sequence within a database record.

These decisions are based on what works best for all of your application program's requirements. However, the way in which you arranged segments in a database record no doubt suits the processing requirements of some applications better than others. For example, look at the two database records shown in the following figure. Both of them contain the same segments, but the hierarchical sequence of segments is different.
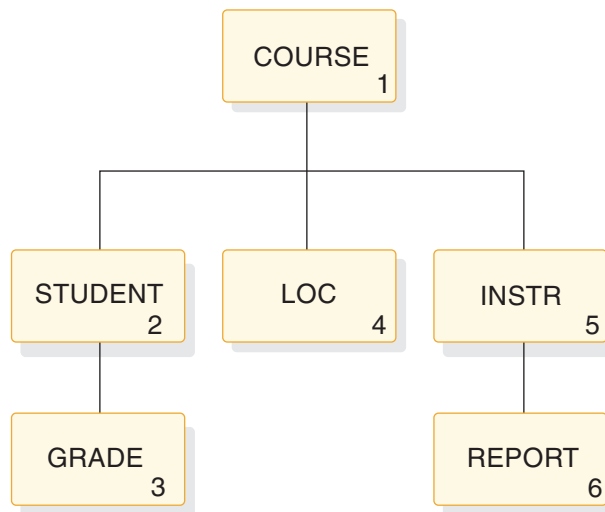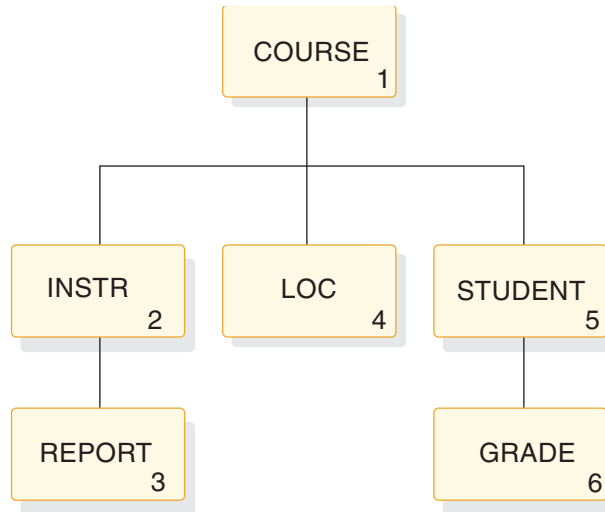
Figure 209. Hierarchy of applications that need to access INSTR and LOC segments

The hierarchy on the top favors applications that need to access INSTR and LOC segments. The hierarchy on the bottom favors applications that need to access STUDENT and GRADE segments. (Favor, in this context, means that access to the segments is faster.) If the applications that access the INSTR and LOC segments are more important than the ones that access the STUDENT and GRADE segments, you can use the database record on the left. But if both applications are equally important, you can split the database record into different data set groups. This will give both types of applications good access to the segments each needs.

To split the database record, you would use two data set groups. As shown in the following figure, the first data set group contains the COURSE, INSTR, REPORT, and LOC segments. The second data set group contains the STUDENT and GRADE segments.
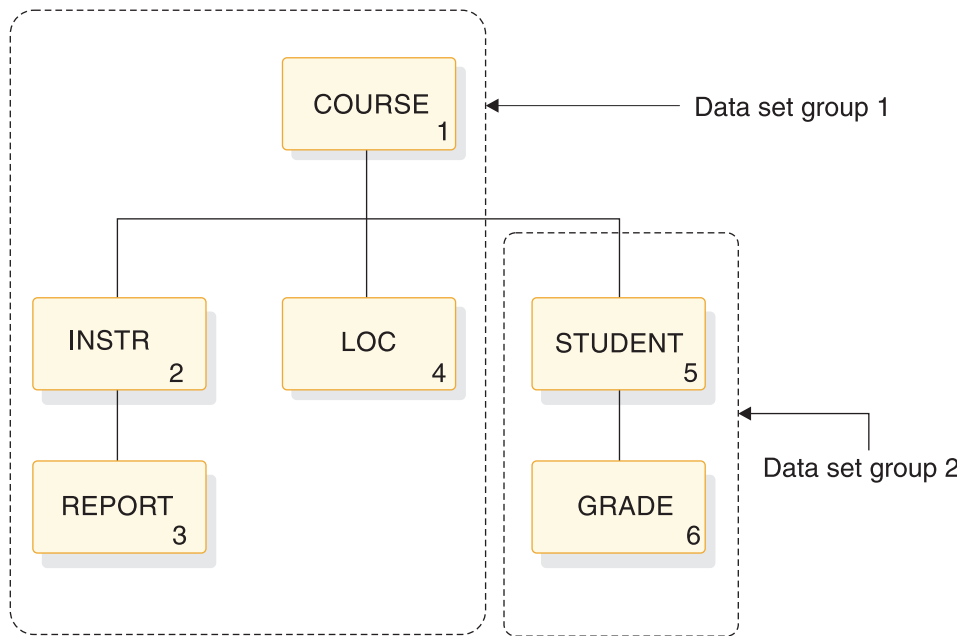
Figure 210. Database record split into two database groups

Other uses of multiple data set groups include:
- Separating infrequently-used segments from high-use segments.
- Separating segments that frequently have information added to them from those that do not. For the former segments, you might specify additional free space so conditions are optimum for additions.
- Separating segments that are added or deleted frequently from those that are not. This can keep space from being fragmented in the main database.
- Separating segments whose size varies greatly from the average segment size. This can improve use of space in the database. Remember, the bitmap in an HD database indicates whether space is available for the *longest* segment type defined in the data set group. It does not keep track of smaller amounts of space. If you have one or more segment types that are large, available space for smaller segments will not be utilized, because the bitmap does not track it.

HD databases using multiple data set groups

You can define as many as ten data set groups when you use multiple data set groups.

The root segment in a database record must be in the primary data set group.

In the database record shown in the following figure, the segments COURSE (1), INSTR (2), LOC (4), and STUDENT (5) could go in one data set group, while segments REPORT (3) and GRADE (6) could go in a second data set group.
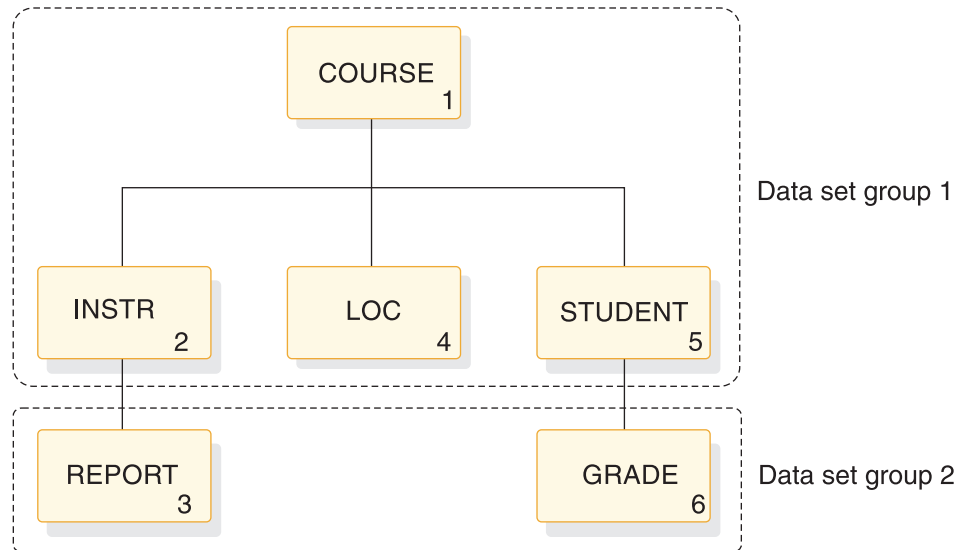
Figure 211. Example of how to divide an HD database record

Examples of how the HD database record shown in the above figure might be divided into three groups are in the following table.

Table 64. Examples of multiple data set grouping

Data set group 1	Data set group 2	Data set group 3
Segment 1	Segments 2, 5, and 6	Segments 3 and 4
Segments 1, 3, and 6	Segments 2 and 5	Segment 3
Segments 1, 3, and 6	Segments 2 and 5	Segment 4

Segments that are separated into different data set groups must be connected by physical child first pointers. For example, in the following figure, the INSTR segment in the primary data set group must point to the first occurrence of its physical child REPORT in the secondary data set group, and STUDENT must point to GRADE.
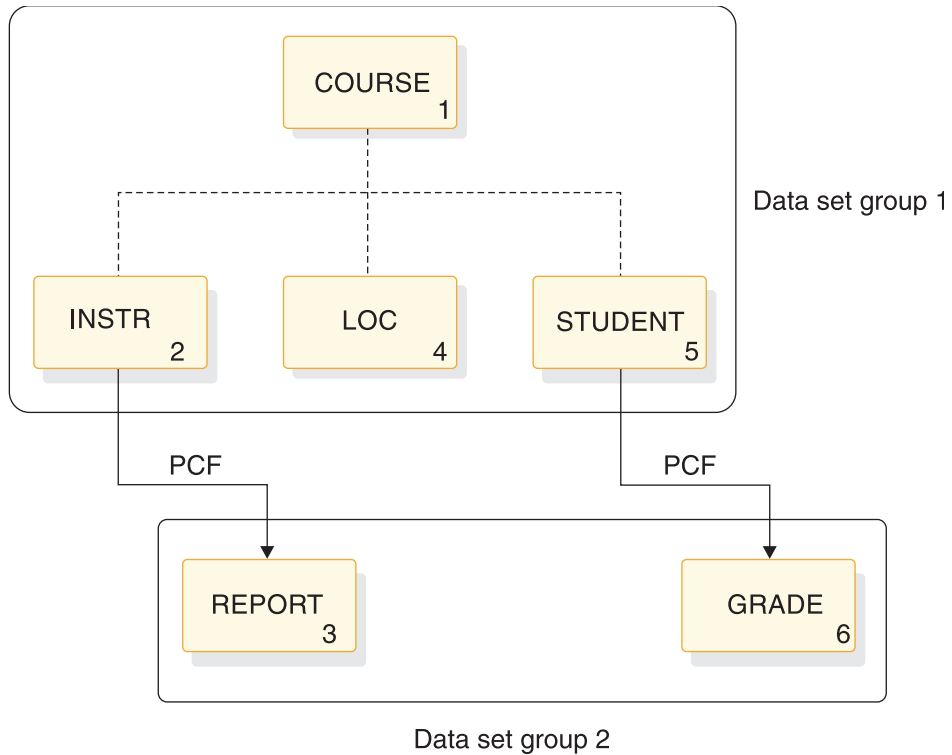
Figure 212. Connecting segments in multiple data set groups using physical child first pointers

How HD records are stored in multiple data set groups

You specify in the DBD statements which segment types are placed in a data set group. IMS then loads the segments into the correct data set group.

The figure below shows one database record:
- Stored in an HDAM or a PHDAM database using two data set groups
- Stored in a HIDAM or a PHIDAM database using two data set groups

In this example, the user specified that four segment types in the database record were put in the primary data set group (COURSE, INSTR, LOC, STUDENT) and two segment types were put in the secondary data set group (REPORT, GRADE).

In the HDAM or PHDAM database, note that only the primary data set group has a root addressable area. The secondary data set group is additional overflow storage.
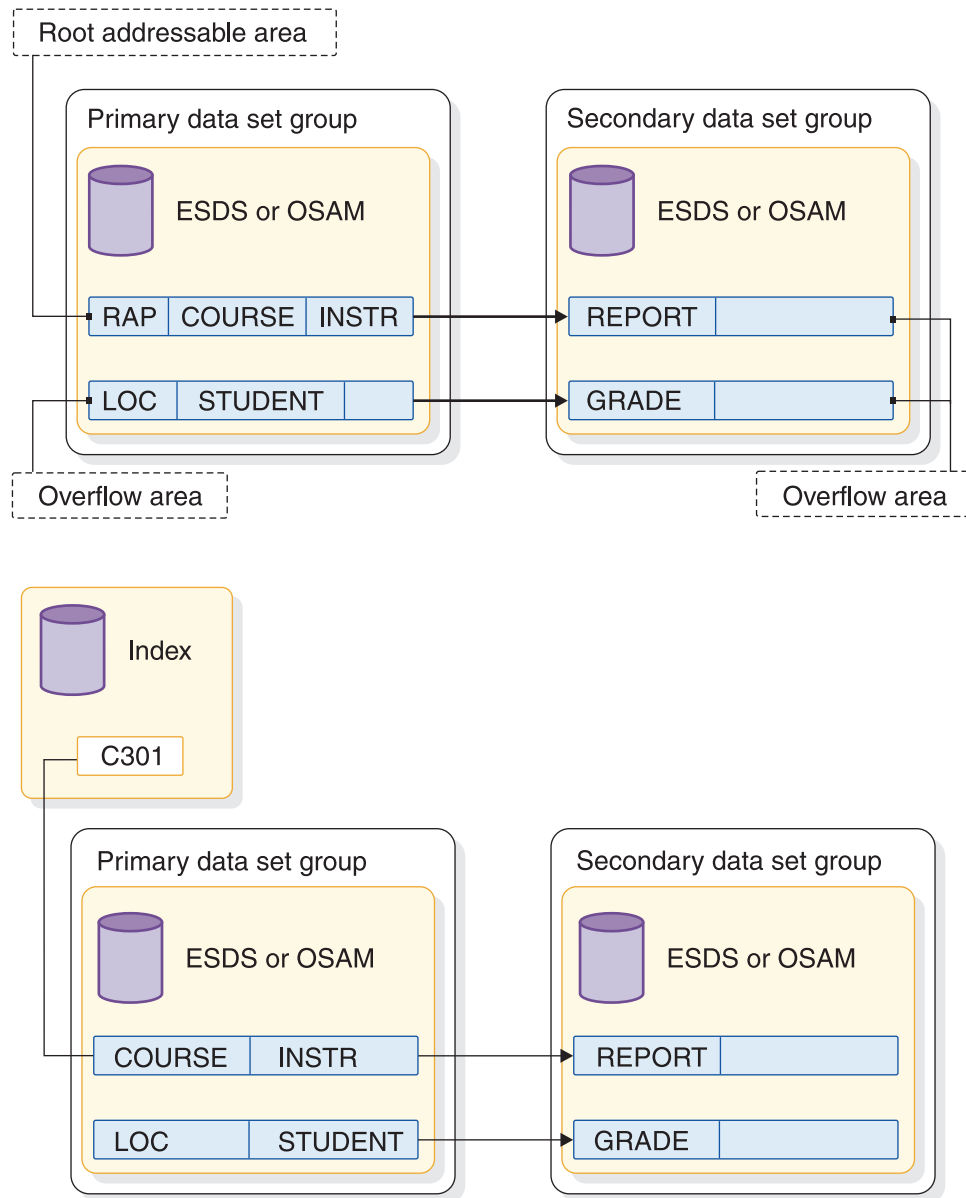
Figure 213. HD database record in storage when multiple data set groups are used

Specifying use of multiple data set groups in HD and PHD databases

You can specify multiple data set groups to IMS in the DBD. For HDAM databases, use the DATASET statement. For PHDAM databases, use the DSGROUP parameter in the SEGM statement.

You can group the segments any way, but you still must list the segments in hierarchical sequence in the DBD.

The following examples use the database record used in "When to use multiple data set groups" on page 410 and "HD databases using multiple data set groups" on page 412. The first example, in the following figure, shows two groups: data set group A contains COURSE and INSTR, data set group B contains all of the other

segments. The second example shows a different grouping. Note the differences in DBDs when the groups are not in sequential hierarchical order of the segments.

Data set group 1
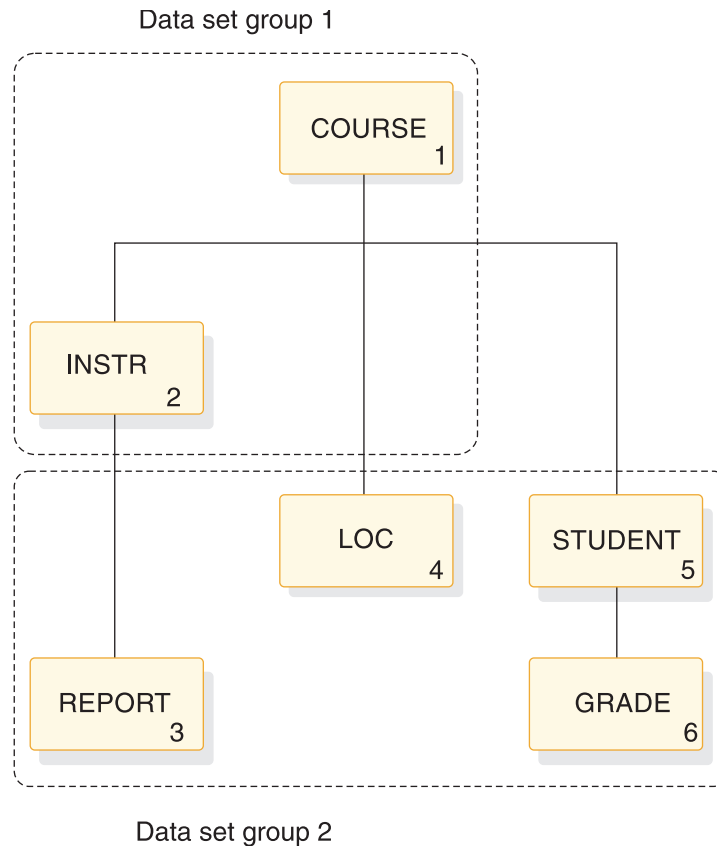
Data set group 2

Figure 214. First example of data set groups

The following code is the HDAM DBD for the first example. Note that the segments are grouped by the DATASET statements preceding the SEGM statements and that the segments are listed in hierarchical order. In each DATASET statement, the DD1= parameter names the VSAM ESDS or OSAM data set that will be used. Also, each data set group can have its own characteristics, such as device type.

```
DBD     NAME=HDMDSG,ACCESS=HDAM,RMNAME=(DFSHDC40,8,500)
DSA     DATASET  DD1=DS1DD
SEGM    NAME=COURSE,BYTES=50,PTR=T
FIELD   NAME=(CODCOURSE,SEQ),BYTES=10,START=1
SEGM    NAME=INSTR,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
DSB     DATASET  DD1=DS2DD,DEVICE=2314
SEGM    NAME=REPORT,BYTES=50,PTR=T,PARENT=((INSTR,SNGL))
SEGM    NAME=LOC,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM    NAME=STUDENT,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM    NAME=GRADE,BYTES=50,PTR=T,PARENT=((STUDENT,SNGL))
DBDGEN
```

The following code shows the DBD for a PHDAM database. Instead of using the DATASET statement, use the DSGROUP parameter in the SEGM statement. The first two segments do not have DSGROUP parameters because it is assumed that they are in the first group.

```
DBD     NAME=HDMDSG,ACCESS=PHDAM,RMNAME=(DFSHDC40,8,500)
SEGM    NAME=COURSE,BYTES=50,PTR=T
FIELD   NAME=(CODCOURSE,SEQ),BYTES=10,START=1
SEGM    NAME=INSTR,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM    NAME=REPORT,BYTES=50,PTR=T,PARENT=((INSTR,SNGL)),DSGROUP=B
```

```
SEGM    NAME=LOC,BYTES=50,PTR=T,PARENT=((COURSE,SNGL)),DSGROUP=B
SEGM    NAME=STUDENT,BYTES=50,PTR=T,PARENT=((COURSE,SNGL)),DSGROUP=B
SEGM    NAME=GRADE,BYTES=50,PTR=T,PARENT=((STUDENT,SNGL)),DSGROUP=B
DBDGEN
```

The second example, in the following figure, differs from the first example in that the groups do not follow the order of the hierarchical sequence. The segments must be listed in the DBD in hierarchical sequence, so additional DATASET statements or DSGROUP parameters are required.
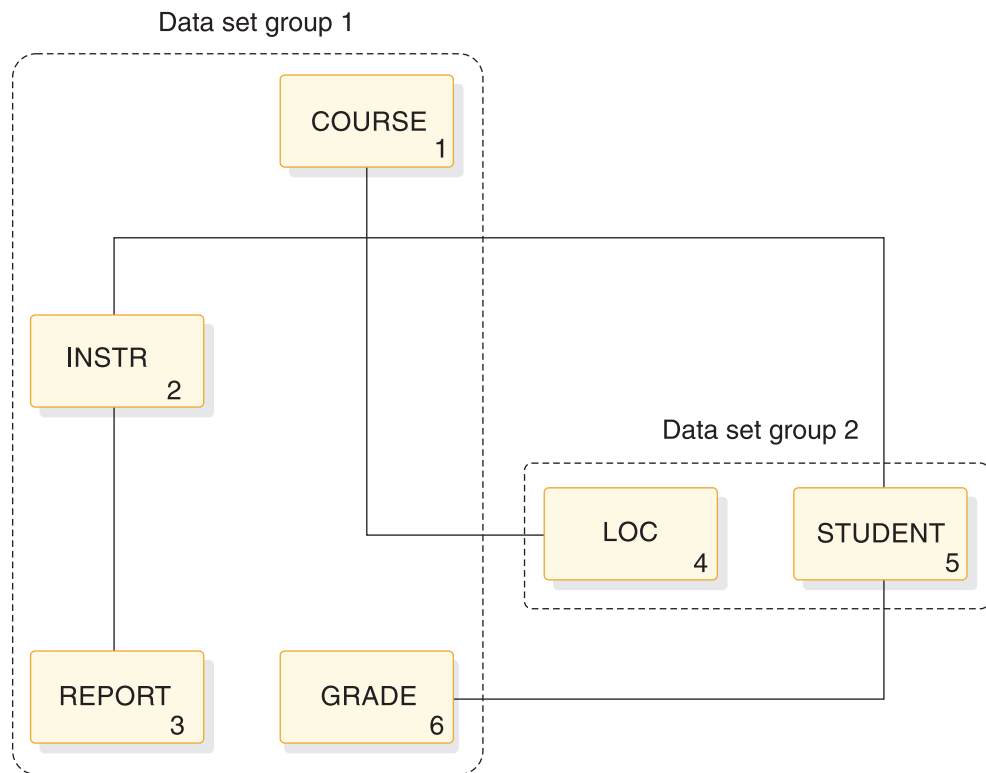
Figure 215. Second example of data set groups

The following code is the DBD for an HDAM database of the second example. It is similar to the first example, except that because the sixth segment is part of the first group, you need another DATASET statement before it with the DSA label. The additional DATASET label groups the sixth segment with the first three.

```
DBD     NAME=HDMDSG,ACCESS=HDAM,RMNAME=(DFSHDC40,8,500)
DSA     DATASET  DD1=DS1DD
SEGM    NAME=COURSE,BYTES=50,PTR=T
FIELD   NAME=(CODCOURSE,SEQ),BYTES=10,START=1
SEGM    NAME=INSTR,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM    NAME=REPORT,BYTES=50,PTR=T,PARENT=((INSTR,SNGL))
DSB     DATASET  DD1=DS2DD,DEVICE=2314
SEGM    NAME=LOC,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM    NAME=STUDENT,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
DSA     DATASET  DD1=DS1DD
SEGM    NAME=GRADE,BYTES=50,PTR=T,PARENT=((STUDENT,SNGL))
DBDGEN
```

The following code is the DBD for a PHDAM database of the second example. It is similar to the first example, except that because the sixth segment is part of the first group, you must explicitly group it with the first two segments by using the DSGROUP parameter.

```
DBD    NAME=HDMDSG,ACCESS=PHDAM,RMNAME=(DFSHDC40,8,500)
SEGM   NAME=COURSE,BYTES=50,PTR=T
FIELD  NAME=(CODCOURSE,SEQ),BYTES=10,START=1
SEGM   NAME=INSTR,BYTES=50,PTR=T,PARENT=((COURSE,SNGL))
SEGM   NAME=REPORT,BYTES=50,PTR=T,PARENT=((INSTR,SNGL)),
SEGM   NAME=LOC,BYTES=50,PTR=T,PARENT=((COURSE,SNGL)),DSGROUP=B
SEGM   NAME=STUDENT,BYTES=50,PTR=T,PARENT=((COURSE,SNGL)),DSGROUP=B
SEGM   NAME=GRADE,BYTES=50,PTR=T,PARENT=((STUDENT,SNGL)),DSGROUP=A
DBDGEN
```

Related tasks:

"Creating HALDB databases with the HALDB Partition Definition utility" on page 542

Related reference:

➡ DATASET statements (System Utilities)

VSAM KSDS CI reclaim for full-function databases

In data-sharing or XRF environments, IMS can reclaim the storage used for empty VSAM KSDS control intervals (CIs) in full-function databases by using a process called CI reclaim.

CI reclaim enhances the performance of database GN, GU, and ISRT calls by reducing the number of empty CIs that VSAM reads on subsequent DL/I calls.

After IMS commits the deletion of the last record in a CI, IMS flags the CI for reclamation. IMS reclaims the CI the next time the CI is read during a DL/I call that has update access. On a single DL/I call, IMS reclaims every empty CI flagged for reclamation that it reads until the DL/I call has been satisfied.

CI reclaim is not a replacement for the routine reorganization of KSDS data sets.

CI reclaim works only under the following circumstances:

- If you are using VSAM subpools, you have specified ERASE=YES, or accepted it as the default, on the DBD statement of the DFSVSMxx PROCLIB member.
- Your KSDS uses unique keys.
- The application program issuing the GN, GU, or ISRT call has update access to the database. Using a processing option PROCOPT of "I", "R", "D" or "A" guarantees the data set is open for update.

Restriction: SHISAM databases do not support CI reclaim. When a large number of records in a SHISAM database are deleted, particularly a large number of consecutive records, serious performance degradation can occur. Eliminate empty CIs and resolve the problem by using VSAM REPRO.

Related Reading: For information about the VSAM REPRO command, see *z/OS DFSMS Access Method Services for Catalogs*.

Related concepts:

"Reorganizing the database" on page 643

➡ Data sharing in IMS environments (System Administration)

➡ Extended Recovery Facility Overview (System Administration)

Part 4. Database design and implementation

This section discusses the design and implementation of IMS databases, including analyzing your data requirements, planning, designing, and implementing each database type.

Chapter 21. Analyzing data requirements

One of the early steps of database design is developing a conceptual data structure that satisfies your end user's processing requirements.

So, before you can develop a conceptual data structure, familiarize yourself with your end user's processing and data requirements.

Developing a data structure is a process of combining the data requirements of each of the tasks to be performed, into one or more data structures that satisfy those requirements. The method explained here describes how to use the local views developed for each business process to develop a data structure.

A business process, in an application, is one of the tasks your end user needs done. For example, in an education application, printing a class roster is a business process.

A local view describes a conceptual data structure and the relationships between the pieces of data in the structure for one business process.

To understand the method explained in this topic, you need to be familiar with the terminology and examples explained in the introductory information on application design in *IMS Version 15 Application Programming*, which explains how to develop local views for the business processes in an application.

Related concepts:

"Design review 3" on page 31

"Design review 4" on page 31

Local view of a business process

Designing a structure that satisfies the data requirements of the business processes in an application requires an understanding of the requirements for each of those business processes.

A local view of the business process describes these requirements because the local view provides:

- A list of all the data elements the process requires and their controlling keys
- The conceptual data structure developed for each process, showing how the data elements are grouped into data aggregates
- The mappings between the data aggregates in each process

This topic uses a company that provides technical education to its customers as an example. The education company has one headquarters, called HQ, and several local education centers, called Ed Centers. HQ develops the courses offered at each of the Ed Centers. Each Ed Center is responsible for scheduling classes it will offer and for enrolling students for those classes.

A class is a single offering of a course on a specific date at an Ed Center. There might be several offerings of one course at different Ed Centers, and each of these offerings is a separate class.

The local views used in this topic are for the following business processes in an education application:

Current Roster

Schedule of Classes

Instructor Skills Report

Instructor Schedules

Notes for local views:

- The asterisks (*) in the data structures for each of the local views indicate the data elements that identify the data aggregate. This is the data aggregate's key; some data aggregates require more than one data element to uniquely identify them.

- The mappings between the data aggregates in each process are given in mapping notation. A one-to-many mapping means for each *A* aggregate there are one or more *B* aggregates; shown like this: ◄━━━━━►►

  A many-to-many relationship means that for each *A* aggregate there are many *B* aggregates, and for each *B* aggregate, there are many *A* aggregates; shown as follows: ◄◄━━━━━►►

Local view 1: current roster

This topic describes the elements, the data structure, the data aggregates, and the mapping of the relationships between the data aggregates used to satisfy the data requirements of the Current Roster business process.

List of current roster data elements

The following is a list of the data elements and their descriptions for our technical education provider example.

Data element
> **Description**

CRSNAME
> Course name

CRSCODE
> Course code

LENGTH
> Length of class

EDCNTR
> Ed Center offering class

DATE Date class is offered

CUST Customer that sent student

LOCTN
> Location of customer

STUSEQ#
> Student's sequence number

STUNAME
> Student's name

STATUS
> Student's enrollment status

ABSENCE
> Student's absences

GRADE
> Student's grade for class

INSTRS
> Instructors for class

The following figure shows the conceptual data structure for the current roster.
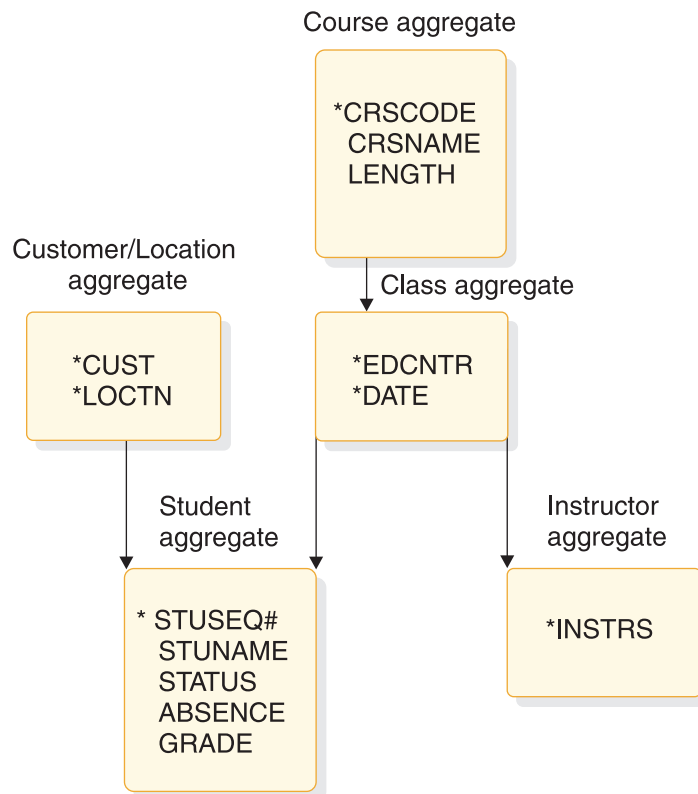
Figure 216. Current roster conceptual data structure

Current roster mappings

The mappings for the current roster are:

> Course ◄────────►► Class
> Class ◄────────►► Student
> Class ◄────────►► Instructor
> Customer/location◄────────►► Student

Local view 2: schedule of classes

This topic describes the elements, the data structure, the data aggregates, and the mapping of the relationships between the data aggregates used to satisfy the data requirements of the Schedule of Classes business process.

List of schedule of classes data elements

The following is a list of the schedule of classes and their descriptions for our example.

Data element
> **Description**

CRSCODE
> Course code

CRSNAME
> Course name

LENGTH
> Length of course

PRICE
> Price of course

EDCNTR
> Ed Center where class is offered

DATE Dates when class is offered at a particular Ed Center

The following figure shows the conceptual data structure for the class schedule.

Course aggregate

```
*CRSCODE
 CRSNAME
 LENGTH
 PRICE
```

Class aggregate

```
*EDCNTR
*DATE
```
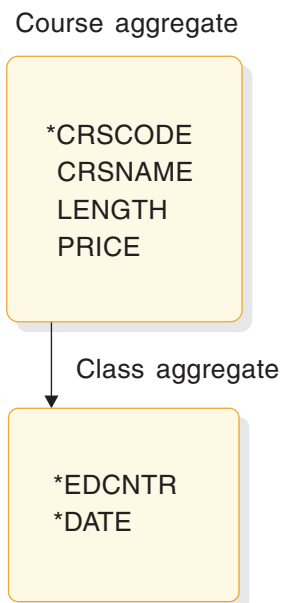
Figure 217. Schedule of classes conceptual data structure

Schedule of classes mappings

The only mapping for this local view is:
> Course ◄─────►► Class
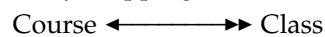
Local view 3: instructor skills report

This topic describes the elements, the data structure, the data aggregates, and the mapping of the relationships between the data aggregates used to satisfy the data requirements of the Instructor Skills Report business process.

List of instructor skills report data elements

The following is a list of the instructor skills report data elements and their descriptions for our technical education provider example.

Data element
    Description

INSTR
    Instructor

CRSCODE
    Course code

CRSNAME
    Course name

The following figure shows the conceptual data structure for the instructor skills report.

Instructor aggregate
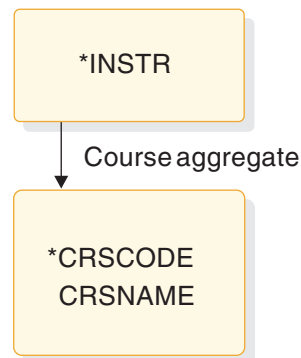
Figure 218. Instructor skills report conceptual data structure

Instructor skills report mappings

The only mapping for this local view is:
    Instructor ◄───────► Course

Local view 4: instructor schedules

This topic describes the elements, the data structure, the data aggregates, and the mapping of the relationships between the data aggregates used to satisfy the data requirements of the Instructor Schedules business process.

List of instructor schedules data elements

The following is a list of the instructor schedules data elements and their descriptions for our example.

Data element
    Description

INSTR
    Instructor

CRSNAME
    Course name

CRSCODE
    Course code

EDCNTR
    Ed Center

DATE Date when class is offered

The following figure shows the conceptual data structure for the instructor schedules.

Instructor aggregate

```
┌─────────────────┐
│                 │
│     *INSTR      │
│                 │
└─────────────────┘
```

Course aggregate

```
┌─────────────────┐
│                 │
│    *CRSCODE     │
│    CRSNAME      │
│                 │
└─────────────────┘
```

Class aggregate

```
┌─────────────────┐
│                 │
│    *EDCNTR      │
│    *DATE        │
│                 │
└─────────────────┘
```
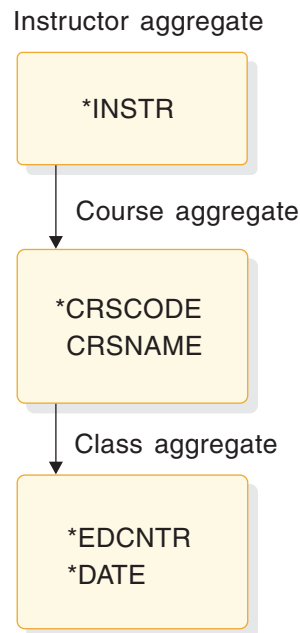
Figure 219. Instructor schedules conceptual data structure

Instructor schedules mappings

The mappings for this local view are:

    Instructor ←————→► Course
    Course ←————→► Class

Related concepts:

➡ Designing an application: Data and local views (Application Programming)

Designing a conceptual data structure

Analyzing the mappings from all the local views is one of the first steps in designing a conceptual data structure.

Two kinds of mappings affect the segments: one-to-many and many-to-many.

A one-to-many mapping means that for each segment *A* there are one or more segment *B*s; shown like this: A ←————→► B. For example, in the Current Roster shown in "Local view of a business process" on page 421, there is a one-to-many relationship between course and class. For each course, there can be several classes scheduled, but a class is associated with only one course. A one-to-many relationship can be represented as a dependent relationship: In the course/class example, the classes are dependent on a particular course.

A many-to-many mapping means that for each segment *A* there are many segment *B*s, and for each segment *B* there are many segment *A*s. This is shown like this: A ◄◄━━━━►► B. A many-to-many relationship is not a dependent relationship, since it usually occurs between data aggregates in two separate data structures and indicates a conflict in the way two business processes need to process that data.

When you implement a data structure with DL/I, there are three strategies you can apply to solve data conflicts:

> Defining logical relationships

> Establishing secondary indexes

> Storing the data in two places (also known as carrying duplicate data).

The first step in designing a conceptual data structure is to combine the mappings of all the local views. To do this, go through the mappings for each local view and make a consolidated list of mappings (see the following table). As you review the mappings:

- Do not record duplicate mappings. At this stage you need to cover each variation, not each occurrence.
- If two data aggregates in different local views have opposite mappings, use the more complex mapping. This will include both mappings when they are combined. For example, if local view #1 has the mapping A ◄━━━━►► B, and local view #2 has the mapping A ◄◄━━━━► B, use a mapping that includes both these mappings. In this case, this is A ◄◄━━━━►► B.

Table 65. Combined mappings for local views

Mapping	Local view
Course ◄━━━━►► Class	1, 2, 4
Class ◄━━━━►► Student	1
Class ◄━━━━►► Instructor	1
Customer/location ◄━━━━►► Student	1
Instructor ◄━━━━►► Course	3, 4

Using the combined mappings, you can construct the data structures shown in the following figure.
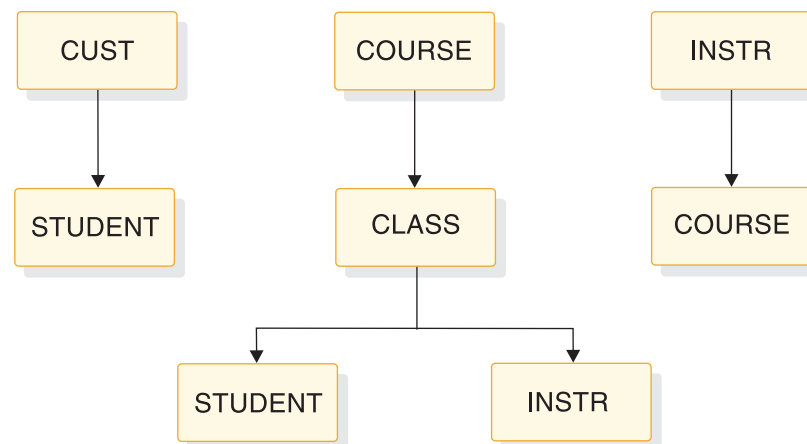
Figure 220. Education data structures

Two conflicts exist in these data structures. First, STUDENT is dependent on both CUST and CLASS. Second, there is an opposite mapping between COURSE and INSTR, and INSTR and COURSE. If you implemented these structures with DL/I, you could use logical relationships to resolve the conflicts.

Related tasks:

"Resolving data conflicts"

Implementing a data structure with DL/I

When you implement a data structure with DL/I, you implement it as a hierarchy. A hierarchy is made up of segments.

In a hierarchy, a one-to-many relationship is called a parent/child relationship. In a hierarchy, each segment can have one or more children, but it can have only one parent.

When you use DL/I, consider how each of the data elements in the structure you developed should be grouped into segments. Also, consider how DL/I can solve any existing data conflicts in the structure.

Assigning data elements to segments

After you determine how data elements are related in a hierarchy, associate each of the data elements with a segment.

To do this, construct a list of all the keys and their associated data elements. If a key and its associated data element appear in several local views, only record the association once.

List the data elements next to their keys, as shown in the following table. The key and its associated data elements become the segment content.

Table 66. Keys and associated data elements

Data aggregate	Key	Data elements
COURSE	CRSCODE	CRSNAME, LENGTH, PRICE
CUSTOMER/LOCATION	CUST, LOCTN	
CLASS	EDCNTR, DATE	
STUDENT	STUSEQ#	STUNAME, ABSENCE, STATUS, GRADE
INSTRUCTOR	INSTR	

If a data element is associated with different keys in different local views, then you must decide which segment will contain the data element. The other thing you can do is to store duplicate data. To avoid doing this, store the data element with the key that is highest in the hierarchy. For example, if the keys ALPHA and BETA were both associated with the data element XYZ (one in local view 1 and one in local view 2), and ALPHA were higher in the hierarchy, store XYZ with ALPHA to avoid having to repeat it.

Resolving data conflicts

The data structure you design can fall short of the application's processing requirements.

For example, one business process might need to retrieve a particular segment by a field other than the one you have chosen as the key field. Another business process might need to associate segments from two or more different data structures. Once you have identified these kinds of conflicts in a data structure and are using DL/I, you can look at two DL/I options that can help you resolve the conflicts: secondary indexing and logical relationships.

Related tasks:

"Designing a conceptual data structure" on page 426

Analyzing requirements for secondary indexes

Secondary indexing allows a segment to be identified by a field other than its key field.

Suppose that you are part of our technical education company and need to determine (from a terminal) whether a particular student is enrolled in a class. If you are unsure about the student's enrollment status, you probably do not know the student's sequence number. The key of the STUDENT segment, however, is STUSEQ#. Let's say you issue a request for a STUDENT segment, and identify the segment you need by the student's name (STUNAME). Instead of the student's sequence number (STUSEQ#), IMS searches through all STUDENT segments to find that one. Assuming the STUDENT segments are stored in order of student sequence numbers, IMS has no way of knowing where the STUDENT segment is just by having the STUNAME.

Using a secondary index in this example is like making STUNAME the key field of the STUDENT segment for this business process. Other business processes can still process this segment with STUSEQ# as the key.

To do this, you can index the STUDENT segment on STUNAME in the secondary index. You can index any field in a segment. When you index a field, indicating to IMS that you are using a secondary index for that segment, IMS processes the segment as though the indexed field were the key.

Analyzing requirements for logical relationships

When a business process needs to associate segments from different hierarchies, logical relationships can make that possible.

Defining logical relationships lets you create a hierarchical structure that does not exist in storage but can be processed as though it does. You can relate segments in separate hierarchies. The data structure created from these logical relationships is called a logical structure. To relate segments in separate hierarchies, store the segment in the path by which it is accessed most frequently. Store a pointer to the segment in the path where it is accessed less frequently.

In the hierarchy shown in the figure in "Designing a conceptual data structure" on page 426, two possible parents exist for the STUDENT segment. If the CUST segment is part of an existing database, you can define a logical relationship between the CUST segment and the STUDENT segment. You would then have the hierarchies shown in the following figure. The CUST/STUDENT hierarchy would be a logical structure.
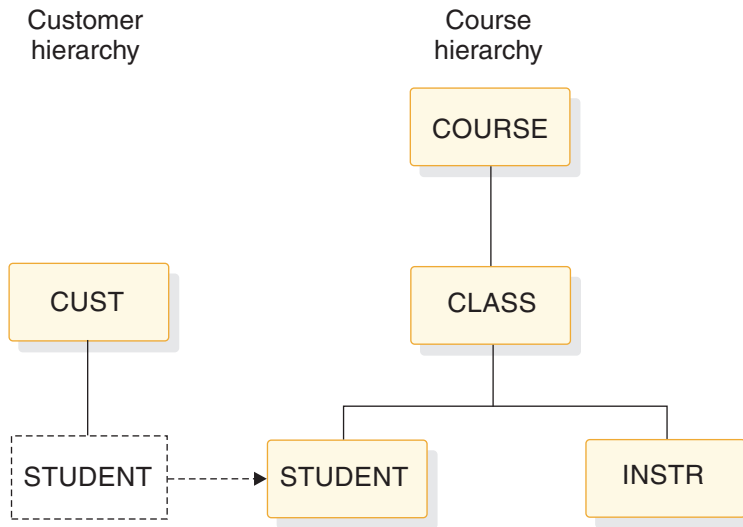
Figure 221. Education hierarchies

This kind of logical relationship is called unidirectional, because the relationship is "one way."

The other conflict you can see in the figure in "Designing a conceptual data structure" on page 426, is the one between COURSE and INSTR. For one course there are several classes, and for one class there are several instructors (COURSE ◄———►► CLASS ◄———►► INSTR), but each instructor can teach several courses (INSTR ◄———►► COURSE). You can resolve this conflict by using a bidirectional logical relationship. You can store the INSTR segment in a separate hierarchy, and store a pointer to it in the INSTR segment in the course hierarchy. You can also store the COURSE segment in the course hierarchy, and store a pointer to it in the COURSE segment in the INSTR hierarchy. This bidirectional logical relationship would give you the two hierarchies shown in the following figure, eliminating the need to carry duplicate data.
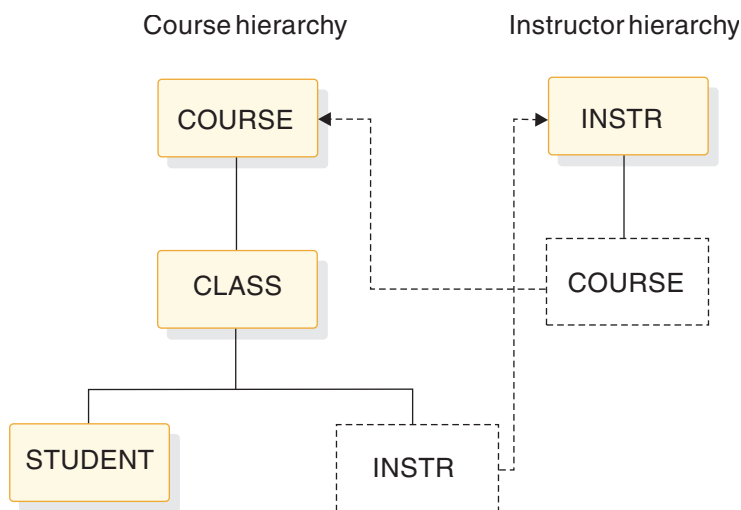
Figure 222. Bidirectional logical relationships

Related tasks:

"Designing a conceptual data structure" on page 426

Chapter 22. Designing full-function databases

After you determine the type of database and optional functions that best suit your application's processing requirements, you need to make a series of decisions about database design and use of options.

This set of decisions primarily determines how well your database performs and how well it uses available space. This series of decisions is made based on:

- The type of database and optional functions you have already chosen
- The performance requirements of your applications
- How much storage you have available for use online

Related concepts:

"Design review 3" on page 31

Related tasks:

"Adjusting HDAM and PHDAM options" on page 700

Specifying free space (HDAM, PHDAM, HIDAM, and PHIDAM only)

Dependent segments inserted after an HD database is loaded are put as close as possible to the segments to which they are related.

However, as the database grows and available space decreases, dependent segments are increasingly put further from their related segments. When this happens, performance decreases, a problem that can only be eliminated by reorganizing the database. (When segments are close to the segments that point to them, the I/O time needed to retrieve a dependent segment is shorter. The I/O time is shorter because the seek time and rotational delay time are shorter.)

To minimize the effect of insert operations after the database is loaded, allocate free space in the database when it is initially loaded. Free space allocation in the database will reduce the performance impact caused by insert operations, and therefore, decrease the frequency with which HD databases must be reorganized.

For OSAM data sets and VSAM ESDS, free space is specified in the FRSPC= keyword of the DATASET statement in the DBD. In the keyword, one or both of the following operands can be specified:

- Free block frequency factor (fbff). The fbff specifies that every nth block or CI in a data set group be left as free space when the database is loaded (where fbff=n). The range of fbff includes all integer values from 0 to 100, except 1. Avoid specifying fbff for HDAM or PHDAM databases. If you specify fbff for HDAM or PHDAM databases and if at load time the randomizing module generates the relative block or CI number of a block or CI marked as free space, the randomizer must store the root segment in another block.

  If you specify fbff, every nth block or CI will be considered a second-most desirable block or CI by the HD Space Search Algorithm. This is true unless you specify SEARCHA=1 in the DATASET statement of the DBDGEN utility. By specifying SEARCHA=1, you are telling IMS not to search for space in the second-most desirable block or CI.

- Free space percentage factor (fspf). The fspf specifies the minimum percentage of each block or CI in a data set group to be left as free space when the database is loaded. The range of fspf is from 0 to 99.

Note: This free space applies to VSAM ESDS and OSAM data sets. It does not apply to HIDAM or PHIDAM index databases or to DEDBs.

For VSAM KSDS, free space is specified in the FREESPACE parameter of the `DEFINE CLUSTER` command. This VSAM parameter is disregarded for a VSAM ESDS data set used for HIDAM, PHIDAM, HDAM, or PHDAM. This command is explained in detail in *z/OS DFSMS Access Method Services for Catalogs*.

Related concepts:

Chapter 32, "Converting database types," on page 841

"How the HD space search algorithm works" on page 177

Related tasks:

"Ensuring a well-organized database" on page 699

"Step 4. Determine the number of blocks or CIs needed for free space" on page 566

Related reference:

▶ DATASET statements (System Utilities)

Estimating the size of the root addressable area (HDAM or PHDAM only)

To estimate the size of the root addressable area, you can use a simple formula.

You can use the following formula to estimate the size of the root addressable area:
`(A x B) / C = D`

where:

A =	the number of bytes of a database record to be stored in the root addressable area
B =	the expected number of database records
C =	the number of bytes available for data in each CI or block CI or block size, minus overhead)
D =	the size you will need, in blocks or CIs, for the root addressable area.

If you have specified free space for the database, include it in your calculations for determining the size of the root addressable area. Use the following formula to accomplish this step:
`(D x E x G) / F = H`

where:

D =	the size you calculated in the first formula (the necessary size of the root addressable area in block or CIs)
E =	how often you are leaving a block or CI in the database empty for free space (what you specified in the fbff operand in the DBD)
F =	(E-1) (fbff-1)

G =	*100*	100 - fspf	The fspf is the minimum percentage of each block or CI you are leaving as free space (what you specified in the fspf operand in the DBD)
H =			the total size you will need, in blocks or CIs

Specify the number of blocks or CIs you need in the root addressable area in the RMNAME=rbn keyword in the DBD statement in the DBD.

Determining which randomizing module to use (HDAM and PHDAM only)

A randomizing module is required to store and access HDAM or PHDAM database records.

The randomizing module converts the key of a root segment to a relative block number and RAP number. These numbers are then used to store or access HDAM or PHDAM root segments. An HDAM database or a PHDAM partition uses only one randomizing module, but several databases and partitions can share the same module. Four randomizing modules are supplied with IMS.

Normally, one of the four randomizing modules supplied with the system will work for your database.

Partition selection is completed prior to invoking the randomizing module on PHDAM databases. The randomizing module selects locations only within a partition.

Write your own randomizing module

If, given your root key distribution, none of these randomizing modules works well for you, write your own randomizing module. If you write your own randomizing module, one of your goals is to have it distribute root segments so that, when subsequently accessing them, only one read and one seek operation is required. When a root key is given to the randomizing module, if the relative block number the randomizer produces is the block actually containing the root, only one read and seek operation is required (access is fast). The randomizing module you write should allow you to vary the number of blocks and RAPs you specify, so blocks and RAPs can be used for tuning the system. The randomizing module should also distribute roots randomly, not randomize to bitmap locations, and keep packing density high.

Assess the effectiveness of the randomizing module

One way to determine the effectiveness of a given randomizing module for your database is to run the IMS High Performance Pointer Checker (HD Tuning Aid). This tool produces a report in the form of a map showing how root segments are stored in the database. It shows you root segment storage based on the number of blocks or CIs you specified for the root addressable area and the number of RAPs you specified for each block or CI. By running the HD Tuning Aid against the various randomizing modules, you can see which module gives you the best distribution of root keys in your database. In addition, by changing the number of RAPs and blocks or CIs you specify, you can see (given a specific randomizing module) which combination of RAPs and blocks or CIs produces the best root segment distribution.

Related concepts:

"Choosing HDAM or PHDAM options"

Related tasks:

Related reference:

⮕ HDAM and PHDAM randomizing routines (DFSHDC40) (Exit Routines)

Choosing HDAM or PHDAM options

In an HDAM or a PHDAM database, the options that you choose can greatly affect performance.

The options discussed here are those you specify in the RMNAME keyword in the DBD statement or when using the HALDB Partition Definition utility. The following figure shows the format for specifying the RMNAME parameter. The definition list that follows explains the meaning of *mod*, *anch*, *rbn*, and *bytes*.

```
RMNAME=(mod,anch,rbn,bytes)
```

mod Name of the randomizing module you have chosen

anch Number of RAPs in a block or CI

rbn Number of blocks or CIs in the root addressable area

bytes Maximum number of bytes of a database record to be put in the root addressable area when segments in the database records are inserted consecutively (without intervening processing operations)

Minimizing I/O operations

In choosing these HDAM or PHDAM options, your primary goal is to minimize the number of I/O operations it takes to access a database record or segment. The fewer I/O operations, the faster the access time. Performance is best when:

- The number of RAPs in a block or CI is equal to the number of roots in the block or CI (block or CI space is not wasted on unused RAPs).
- Unique block and RAP numbers are generated for most root segments (thereby eliminating long synonym chains).
- Root segments are stored in key sequence.
- All frequently used dependent segments are in the root addressable area (access to the root addressable area is faster than access to the overflow area) and in the same block or CI as the root.

Your choice of a randomizing module determines how many addresses are unique for each root and whether roots are stored in key sequence. In general, a randomizing module is considered efficient if roots are distributed evenly in the root addressable area. You can experiment with different randomizing modules. Try various combinations of the anch, rbn, and bytes operands to see what effect they have on distribution of root segments.

Maximizing packing density

A secondary goal in choosing HDAM or PHDAM options is to maximize packing density without adversely affecting performance. Packing density is the percentage

of space in the root addressable area being used for root segments and the dependent segments associated with them. Packing density is determined as follows:

```
Packing density =
( Number of roots  x  root bytes ) /
( Number of CIs in the root addressable area  x  Usable space in the CI )
```

root bytes
> The average number of bytes in each root in the root addressable area.

Usable space in the CI
> The CI or block size minus (as applicable) space for the FSEAP, RAPs, VSAM CIDF, VSAM RDF, and free space.

Packing density should be high, but, as the percentage of packing density increases, the number of dependent segments put into overflow storage can increase. In addition, performance for processing of dependent segments decreases when they are in overflow storage. All of the operands you can specify in the RMNAME= keyword affect packing density. So, to optimize packing density, try different randomizing modules and various combinations of the anch, rbn, and bytes operands.

Related concepts:

Chapter 32, "Converting database types," on page 841

"Determining which randomizing module to use (HDAM and PHDAM only)" on page 433

Related tasks:

"Adjusting HDAM and PHDAM options" on page 700

Choosing a logical record length for a HISAM database

In a HISAM database, your choice of a logical record length is important because it can affect both the access time and the use of space in the database.

The relative importance of each depends on your individual situation. To get the best possible performance and an optimum balance between access time and the use of space, plot several trial logical record lengths and test them before making a final choice.

Logical record length considerations

The following should be considered:
- Only complete segments can be stored in a logical record. Therefore, the space between the last segment that fit in the logical record and the end of the logical record is unused.
- Each database record starts at the beginning of a logical record. The space between the end of the database record and the end of the last logical record containing it is unused. This unused space is relative to the average size of your database records.
- Very short or very long logical records tend to increase wasted space. If logical records are short, the number of areas of unused space increases. If logical records are long, the size of areas of unused space increases. The following figure shows why short or long logical records increase wasted space.

Choose a logical record length that minimizes the amount of unused space at the end of logical records.

The database record shown in the following figure is stored on three short logical records in Figure 224 and in two longer logical records in Figure 225. Note the three areas of unused space.
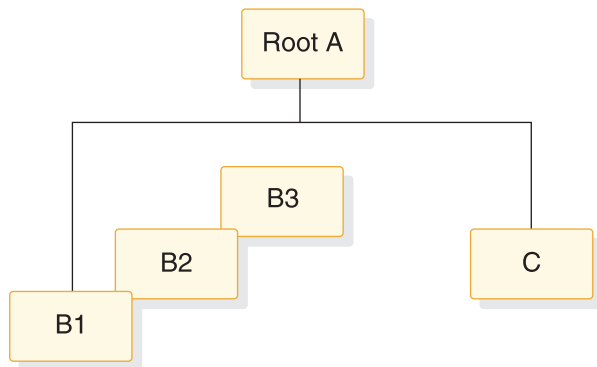
Figure 223. Database record for logical record examples

In the following figure, note the three areas of unused space. In Figure 225, there are only two areas of unused space, rather than three, but the total size of the areas is larger.
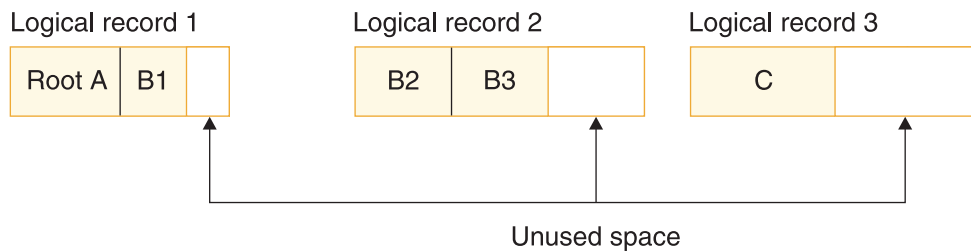
Figure 224. Short logical records
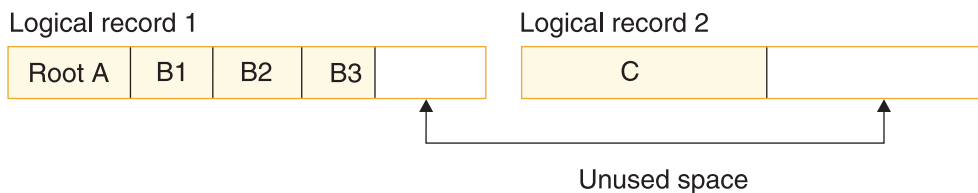
Figure 225. Long logical records

Segments in a database record that do not fit in the logical record in the primary data set are put in one or more logical records in the overflow data set. More read and seek operations, and therefore longer access time, are required to access logical records in the overflow data set than in the primary data set. This is especially true as the database grows in size and chains of overflow records develop. Therefore, you should try to put the most-used segments in your database record in the primary data set. When choosing a logical record length the primary data set should be as close to average database record length as possible. This results in a minimum of overflow logical records and thereby minimizes performance problems. When you calculate the average record length, beware of unusually long or short records that can skew the results.

A read operation reads one CI into the buffer pool. CIs contain one or more logical records in a database record. Because of this, it takes as many read and seek

operations to access an entire database record as it takes CIs to contain it. In Figure 227, each CI contains two logical records, and two CIs are required to contain the database record shown in the following figure. Consequently, it takes two read operations to get these four logical records into the buffer.
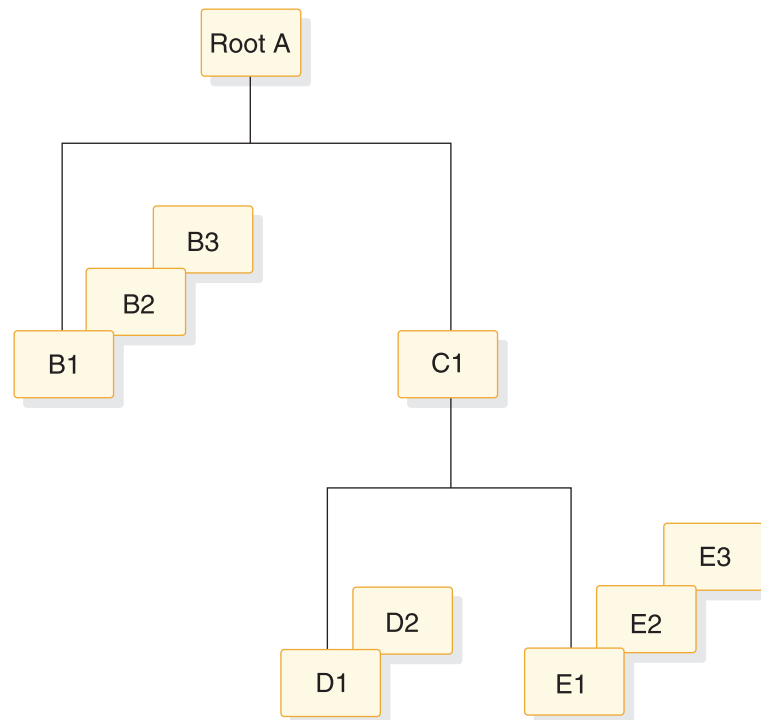
Figure 226. Database record for logical records example
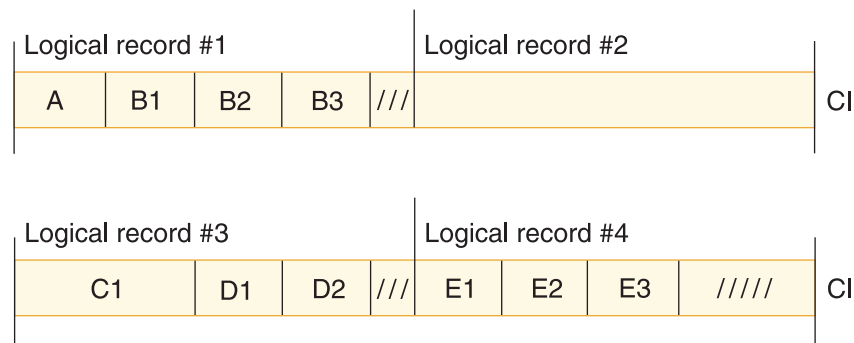
Figure 227. Logical records example with two read operations

The number of read and seek operations required to access a database record increases as the size of the logical record decreases. The question to consider is: Do you often need access to the entire database record? If so, you should try to choose a logical record size that will usually contain an entire database record. If, however, you typically access only one or a few segments in a database record, choice of a logical record size large enough to contain the average database record is not as important.

Consider what will happen in the following setup example in which you need to read database records, one after another:

- Your CI or block size is 2048 bytes.

- Your Logical record size is 512 bytes.
- Your Average database record size is 500 bytes.
- The range of your database record sizes is 300 to 700 bytes.

Because your logical and average database record sizes are about equal (512 and 500), approximately one of every two database records will be read into the buffer pool with one read operation. (This assumption is based on the average size of database records.) If, however, your logical record size were 650, you would access most database records with a single read operation. An obvious trade-off exists here, one you must consider in picking a logical record length for HISAM data sets. If your logical record size were 650, much unused space would exist between the end of an average database record and the last logical record containing it.

Rules to observe

The following rules must be observed when choosing a logical record length for HISAM data sets:
- Logical record size in the primary data set must be at least equal to the size of the root segment, plus its prefix, plus overhead. If variable-length segments are used, logical record size must be at least equal to the size of the longest root segment, plus its prefix, plus overhead. Five bytes of overhead is required for VSAM.
- Logical record size in the overflow data set must be at least equal to the size of the longest segment in the overflow data set, plus its prefix, plus overhead. Five bytes of overhead is required for VSAM.
- Logical record lengths in the overflow data set must be equal to or greater than logical record length in the primary data set.
- The maximum logical record size is 30720 bytes.
- Except for SHISAM databases, logical record lengths must be an even number.

Calculating how many logical records are needed to hold a database record

Calculate the average size of a database record before plotting various logical record sizes. By calculating the average size of a database record, given a specific logical record size, you can see how many logical records it takes to hold a database record (of average size).

Specifying logical record length

Specify the length of the logical records on the RECORD= parameter in the DATASET statement of the DBD Generation utility.

Related tasks:

"Estimating the minimum size of the database" on page 559

Related reference:

➡ DATASET statements (System Utilities)

Choosing a logical record length for HD databases

In HD databases, the important choice is not logical record length but CI or block size.

Logical record length is the same as block size when VSAM is used. Logical record size is equal to CI size, minus 7 bytes of overhead (4 bytes for a CIDF, 3 bytes for an RDF).

As with HISAM databases, specify the length of the logical records in the RECORD= operand of the DATASET statement in the DBD.

Related concepts:

"Determining the size of CIs and blocks"

Chapter 32, "Converting database types," on page 841

Determining the size of CIs and blocks

You can specify the CI size for your database. The DBDGEN utility calculates the CI size for you, and you must use this size in your DEFINE CLUSTER command when you use IDCAMS to create the data set.

Based on CI size, VSAM determines the size of physical blocks on a DASD track. VSAM always uses the largest possible physical block size, because the largest block size best utilizes space on the track. So your choice of a CI size is an important one. Your goal in picking it is to keep a high percentage of space on the track for your data, rather than for device overhead.

Track sizes vary from one device to another, and many different CI sizes you can specify exist. Because you can specify different CI sizes, the physical block size that VSAM picks varies and is based on device overhead factors. For information about using VSAM data sets, refer to *z/OS DFSMS Access Method Services for Catalogs*.

Related concepts:

Chapter 32, "Converting database types," on page 841

Related tasks:

"Changing direct-access storage devices" on page 698

"Choosing a logical record length for HD databases" on page 438

Recommendations for specifying sizes for blocks, CIs, and records

The following recommendations can ensure that your databases are using appropriate block, CI, and record sizes.

- In general, large CI sizes are good for sequential processing, but not for random processing. For indexes and HISAM databases in which sequential processing is most important, CI sizes of at least 8 K are typically best.
- For INDEX databases, such as secondary indexes and HIDAM primary indexes, do not specify the RECORD parameter on the DATASET statement. IMS calculates the appropriate CI sizes automatically.
- For KSDS data sets used by PSINDEX, INDEX, PHIDAM, HIDAM, and HISAM databases, to determine the values for the RECORDSIZE parameter in the IDCAMS DEFINE statements, use the output listing from the DBDGEN process.
- For the data component of KSDS data sets, specify a CI size on the IDCAMS DEFINE statement. The CI size for the data component determines the number of logical records stored in the CI. For the index component of KSDS data sets, do not specify a CI size. By default, DFSMS automatically selects the optimum CI size for the index component of KSDS data sets.
- For the CI size of VSAM ESDS and KSDS data sets, specify the value of the SIZE parameter on the DATASET statement by using the value that is specified on the RECORDSIZE parameter of the IDCAMS DEFINE statement. The CI size for

VSAM ESDS and KSDS data sets is determined by the IDCAMS RECORDSIZE parameter and not the DBDGEN SIZE parameter. Using the same value for each parameter can help avoid confusion about which value is in effect.

- For HDAM and HIDAM OSAM data sets, specify the SIZE parameter, but not the BLOCK parameter in the DATASET statement.
- For HISAM databases, unless your record sizes vary, specify a value for the RECORD parameter of the DATASET statement large enough to accommodate the largest record size. If your HISAM database record sizes vary, to avoid wasting space, you can specify a RECORD size large enough to hold the majority of database records and specify a record size for the overflow data set that is large enough to hold the largest record size.
- For OSAM block sizes and VSAM ESDS CI sizes, the block and CI sizes should be large enough to hold entire database records. When database record sizes are very large or vary greatly in size, make the block or CI size large enough to hold the most frequently accessed segments in the database records.
- On SEGM statements, do not specify the FREQ parameter.

Number of open full-function database data sets

IMS does not enforce a limit on the number of full-function database data sets that can be open at the same time by multiple databases. However, the resources available at your installation and the consumption of those resources by both your IMS configuration and the other z/OS subsystems that your installation might be running, such as Db2 for z/OS, could potentially limit the number of data sets that you can open.

For full-function databases, one of the resources that could become constrained with a large number of open data sets is the private storage of the DL/I separate address space (DLISAS), which requires storage both above and below the 16 MB line.

Buffering options

Database buffers are defined areas in virtual storage. When an application program processes a segment in the database, the entire block or CI containing the segment is read from the database into a buffer. The application program processes the segment while it is in the buffer.

If the processing involves modifying any segments in the buffer, the contents of the buffer must eventually be written back to the database so the database is current.

You need to choose the size and number of buffers that give you the maximum performance benefit. If your database uses OSAM, you might also decide to use OSAM sequential buffering. The subtopics in this topic can help you with these decisions.

Related tasks:

Multiple buffers in virtual storage

You can specify both the number of buffers needed in virtual storage and their size. You can specify multiple buffers with different sizes.

Because a complete block or CI is read into a buffer, the buffer must be at least as large as the block or CI that is read into it. For best performance, use multiple

buffers in virtual storage. To understand why, you need to understand the concept of buffers and how they are used in virtual storage.

When the data an application program needs is already in a buffer, the data can be used immediately. The application program is not forced to wait for the data to be read from the database to the buffer. Because the application program does not wait, performance is better. By having multiple buffers in virtual storage and by making a buffer large enough to contain all the segments of a CI or block, you increase the chance that the data needed by application programs is already in virtual storage. Thus, the reason for having multiple buffers in virtual storage is to eliminate some of an application program's wait time.

In virtual storage, all buffers are put in a buffer pool. Separate buffer pools exist for VSAM and OSAM. A buffer pool is divided into subpools. Each subpool is defined with a subpool definition statement. Each subpool consists of a specified number of buffers of the same size. With OSAM and VSAM you can specify multiple subpools with buffers of the same size.

Subpool buffer use chain

In the subpool, buffers are chained together in the order in which they have been used. This organization is called a *use chain.*

The most recently used buffers are at the top of the use chain and the least recently used buffers are at the bottom.

The buffer handler

When a buffer is needed, an internal component called the buffer handler selects the buffer at the bottom of the use chain, because buffers that are least recently used are less likely to contain data an application program needs to use again.

If a selected buffer contains data an application program has modified, the contents of the buffer are written back to the database before the buffer is used. This causes the application program wait time discussed earlier.

Background write option

If you use VSAM, you can reduce or eliminate wait time by using the background write option.

Otherwise, you control and reduce wait time by carefully choosing of the number and size of buffers.

Related tasks:

Shared resource pools

You can define multiple VSAM local shared resource pools. Multiple local shared resource pools allow you to specify multiple VSAM subpools of the same size.

You create multiple shared resource pools and then place in each one a VSAM subpool that is the same size as other VSAM subpools in other local shared resource pools. You can then assign a specific database data set to a specific subpool by assigning the data set to a shared resource pool. The data set is directed to a specific subpool within the assigned shared resource pool based on the data set's control interval size.

Using separate subpools

If you have many VSAM data sets with similar or equal control interval sizes, you might get a performance advantage by replacing a single large subpool with separate subpools of identically sized buffers. Creating separate subpools of the same size for VSAM data sets offers benefits similar to OSAM multiple subpool support.

You can also create separate subpools for VSAM KSDS index and data components within a VSAM local shared resource pool. Creating separate subpools can be advantageous because index and data components do not need to share buffers or compete for buffers in the same subpool.

Hiperspace buffering

Multiple VSAM local shared resource pools enhance the benefits provided by Hiperspace™ buffering.

Hiperspace buffering allows you to extend the buffering of 4K and multiples of 4K buffers to include buffers allocated in expanded storage in addition to the buffers allocated in virtual storage. Using multiple local shared resource pools and Hiperspace buffering allows data sets with certain reference patterns (for example, a primary index data set) to be isolated to a subpool backed by Hiperspace, which reduces the VSAM read I/O activity needed for database processing.

Hiperspace buffering is activated at IMS initialization. In batch systems, you place the necessary control statements in the DFSVSAMP data set. In online systems, you place the control statements in the IMS.PROCLIB data set with the member name DFSVSMnn. Hiperspace buffering is specified for VSAM buffers through one or two optional parameters applied to the VSRBF subpool definition statement.

The total space that you can allocate to a Hiperspace buffer pool is limited to 2 GB. If the number of buffers multiplied by the buffer size exceeds 2 GB, IMS sets the pool size at 2 GB and issues a warning message.

Related concepts:

"Hiperspace buffering parameters" on page 704

Buffer size

Pick buffer sizes that are equal to or larger than the size of the CIs and blocks that are read into the buffer.

A variety of valid buffer sizes exist. If you pick buffers larger than your CI or block sizes, virtual storage is wasted.

For example, suppose your CI size is 1536 bytes. The smallest valid buffer size that can hold your CI is 2048 bytes. This wastes 512 bytes (2048 - 1536) and is not a good choice of CI and buffer size.

Number of buffers

Pick an appropriate number of buffers of each size so buffers are available for use when they are needed, an optimum amount of data is kept in virtual storage during application program processing, and application program wait time is minimized.

The trade-off in picking a number of buffers is that each buffer uses up virtual storage.

When you initially choose buffer sizes and the number of buffers, you are making a scientific guess based on what you know about the design of your database and the processing requirements of your applications. After you choose and implement buffer size and numbers, various monitoring tools are available to help you determine how well your scientific guess worked.

Buffer size and number of buffers are specified when the system is initialized. Both can be changed (tuned) for optimum performance at any time.

Related concepts:

Chapter 29, "Monitoring databases," on page 635

Chapter 30, "Tuning databases," on page 643

Chapter 32, "Converting database types," on page 841

VSAM buffer sizes

The buffer sizes (in bytes) that you can choose when using VSAM as the access method are.

> 512
>
> 1024
>
> 2048
>
> 4096
>
> 8192
>
> 12288
>
> 16384
>
> 20480
>
> 24576
>
> 28672
>
> 32768

In order not to waste buffer space, choose a buffer size that is the same as a valid CI size. Valid CI sizes for VSAM data clusters are:

- For data components up to 8192 bytes (or 8K bytes), the CI size must be a multiple of 512.
- For data components over 8192 bytes (or 8K bytes), the CI size must be a multiple of 2048 (up to a maximum of 32768 bytes).

Valid CI sizes (in bytes) for VSAM index clusters using VSAM catalogs are:

> 512
>
> 1024
>
> 2048
>
> 4096

Valid CI sizes for VSAM index clusters using integrated catalog facility catalogs are:

- For index components up to 8192 bytes (or 8K bytes), the CI size must be a multiple of 512.
- For index components over 8192 bytes (or 8K bytes), the CI size must be a multiple of 2048 (up to a maximum of 32768 bytes).

OSAM buffer sizes

For OSAM data sets, choose a buffer size that is the same as a valid block size so that buffer space is not wasted. Valid block sizes for OSAM data sets are any size from 18 to 32768 bytes.

The buffer sizes (in bytes) that you can choose when using OSAM as the access method are:

512

1024

2048

Any multiple of 2048 up to a maximum of 32768

Restriction: When using sequential buffering and the coupling facility for OSAM data caching, the OSAM database block size must be defined in multiples of 256 bytes (decimal). Failure to define the block size accordingly can result in ABENDS0DB from the coupling facility. This condition exists even if the IMS system is accessing the database in read-only mode.

Specifying buffers

Specify the number of buffers and their size when the system is initialized.

Your specifications, which are given to the system in the form of control statements, are put in the:

- DFSVSAMP data set in batch, utility.
- IMS.PROCLIB data set with the member name DFSVSMnn in IMS DCCTL and DBCTL environments.

The following example shows the necessary control statements specifications:

- Four 2048-byte buffers for OSAM
- Four 2048-byte buffers and fifteen 1024-byte buffers for VSAM

```
    //DFSVSAMP DD *
⋮
    VSRBF=2048,4
    VSRBF=1024,15
    IOBF=(2048,4)
    /*
```

z/OS DFSMS calculates the optimum size for the CIs of the index component of VSAM data sets. If DFSMS determines that the CI size needs to be increased, DFSMS overrides the CI size specified in the IDCAMS DEFINE statement. If DFSMS increases the size of the CI beyond the size specified for the associate buffer pool, the database cannot open and IMS issues message DFS0730I with a determination code of O,DC. In the event that the CI size is increased beyond the buffer pool size, you must increase the buffer pool size to match the CI size.

OSAM buffers can be fixed in storage using the IOBF= parameter. In VSAM, buffers are fixed using the VSAMFIX= parameter in the OPTIONS statement. Performance is generally improved if buffers are fixed in storage, then page faults do not occur. A page fault occurs when an instruction needs a page (a specific piece of storage) and the page is not in storage.

With OSAM, you can fix the buffers and their buffer prefixes, or the buffer prefixes and the subpool header, in storage. In addition, you can selectively fix buffer

subpools, that is, you can choose to fix some buffer subpools and not others. Buffer subpools are fixed using the IOBF= parameter.

Using the IOBF= parameter you can specify:
- The size of buffers in a subpool.
- The number of buffers in a subpool. If three or fewer are specified, IMS gives you three; otherwise, it gives you the number specified. If you do not specify a sufficient number of buffers, your application program calls could waste time waiting for buffer space.
- Whether the buffers and buffer prefixes in this subpool need to be fixed.
- Whether the buffer prefixes in this subpool and the subpool header need to be fixed.
- An identifier to be assigned to the subpool. The identifier is used in conjunction with the DBD statement to assign a specific subpool to a given data set. This DBD statement is not the DBD statement used in a DBD generation but one specified during execution. The identifier allows you to have more than one subpool with the same buffer size. You can use it to:
  - Get better distribution of activity among subpools
  - Direct new database applications to "private" subpools
  - Control the contention between a BMP and MPPs for subpools

Related concepts:

➡ IMS buffer pools (System Definition)

➡ OSAM subpool definition (System Definition)

Related tasks:

"VSAM options" on page 453

Related reference:

➡ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

➡ Defining OSAM subpools (System Definition)

➡ DD statements for IMS procedures (System Definition)

OSAM sequential buffering

Sequential Buffering (SB) is an extension of the normal buffering technique used for OSAM database data sets.

When SB is active, multiple consecutive blocks can be read from your database with a single I/O operation. (SB does not enhance OSAM write operations.) This technique can help reduce the elapsed time of many programs and utilities that sequentially process your databases.

Related tasks:

"Unloading the existing database" on page 850

Sequential buffering introduction

OSAM sequential buffering performs a *sequential read* of ten consecutive blocks with a single I/O operation, while the normal OSAM buffering method performs a *random read* of only one block with each I/O operation.

Without SB, IMS must issue a random read each time your program processes a block that is not already in the OSAM buffer pool. For programs that process your

databases sequentially, random reads can be time-consuming because the DASD must rotate one revolution or more between each read.

SB reduces the time needed for I/O read operations in three ways:

- By reading 10 consecutive blocks with a single I/O operation, sequential reads reduce the number of I/O operations necessary to sequentially process a database data set.

  When a sequential read is issued, the block containing the segment your program requested plus nine adjacent blocks are read from the database into an SB buffer pool in virtual storage. When your program processes segments in any of the other nine blocks, no I/O operations are required because the blocks are already in the SB buffer pool.

  Example: If your program sequentially processes an OSAM data set containing 100,000 consecutive blocks, 100,000 I/O operations are required using the normal OSAM buffering method. SB can take as few as 10,000 I/O operations to process the same data set.

- By monitoring the database I/O reference pattern and deciding if it is more efficient to satisfy a particular I/O request with a sequential read or a random read. This decision is made for each I/O request processed by SB.

- By overlapping sequential read I/O operations with CPC processing and other I/O operations of the same application. When overlapped sequential reads are used, SB anticipates future requests for blocks and reads those blocks into SB buffers before they are actually needed by your application. (Overlapped I/O is supported only for batch and BMP regions.)

Benefits of sequential buffering

By using OSAM sequential buffering (SB), any application program or utility that sequentially processes OSAM data sets can run faster.

Because many other factors affect the elapsed time of a job, the time savings is difficult to predict. You need to experiment with SB to determine actual time savings.

Programs that can benefit from SB

Some of the programs, utilities, and functions that might benefit from the use of SB are:

- IMS batch programs that sequentially process your databases.
- BMPs that sequentially process your databases.
- Any long-running MPP, Fast Path, and CICS programs that sequentially process your databases.

  Note: SB is possible but not recommended for short-running MPP, IFP, and CICS programs. SB is not recommended for the short-running programs, because SB has a high initialization overhead each time such online programs are run.

- IMS utilities, including:
  - Online Database Image Copy
  - HD Reorganization Unload
  - Partial Database Reorganization
  - Surveyor
  - Database Scan

- – Database Prefix Update
- – Batch Backout
- HALDB Online Reorganization function

Typical productivity benefits of SB

By using SB for programs and utilities that sequentially process your databases, you might be able to:

- Run existing sequential application programs within decreasing "batch window times." For example, if the time you set aside to run batch application programs is reduced by one hour, you might still be able to run all the programs you normally run within this reduced time period.
- Run additional sequential application programs within the same time period.
- Run some sequential application programs more often.
- Make online image copies much faster.
- Reduce the time needed to reorganize your databases.

Flexibility of SB use

IMS provides several methods for requesting SB.

You can request the use of SB for specific programs and utilities during PSBGEN or by using SB control statements. You can also request the use of SB for all or some batch and BMP programs by using an SB Initialization Exit Routine.

IMS also allows a system programmer or master terminal operator (MTO) to override requests for the use of SB by disallowing its use. This is done by issuing an SB MTO command or using an SB Initialization Exit Routine. The use of SB can be disallowed during certain times of the day to avoid virtual or real storage constraint problems.

Related tasks:

"How to request the use of SB" on page 449

How SB buffers data

The following sections describe what sequential buffering (SB) does, including what SB buffers, how and when SB is activated, and what happens to the data that SB buffers.

What SB buffers

HD databases can consist of multiple data set groups. A database PCB can therefore refer to several data set groups. A database PCB can also refer to several data set groups when the database referenced by the PCB is involved in logical relationships. A particular database, and therefore a particular data set group, can be referenced by multiple database PCBs. A specific data set group that is referenced by a specific database PCB is referred to in the following sections as a *DB-PCB/DSG pair*.

When SB is activated, it buffers data from the OSAM data set associated with a specific DB-PCB/DSG pair. SB can be active for several DB-PCB/DSG pairs at the same time, but each pair requires a separate activation.

If you use OSAM SB with a HALDB database, because each database can have multiple partitions, HALDB DB-PCB/DSG pairs are further qualified by partition

IDs. The SB blocks that are created for a HALDB partition cannot be shared among application program PSTs.

Periodic evaluation and conditional activation of SB

IMS does not immediately activate SB when you request it. Instead, when SB is requested for a program, IMS begins monitoring the I/O reference pattern and activity rate for each DB-PCB/DSG pair that is used by the program. After awhile, IMS performs the first of a series of *periodic evaluations* of the buffering process. IMS performs periodic evaluations for each DB-PCB/DSB pair and determines if the use of SB would be beneficial for the DB-PCB/DSG pair. If the use of SB is beneficial, IMS activates SB for the DB-PCB/DSG pair. This activation of SB is known as *conditional activation*.

After SB is activated, IMS continues to periodically evaluate the I/O reference pattern and activity rate. Based on these evaluations, IMS can take either of the following actions:
- Temporarily deactivate SB and continue to monitor the I/O reference pattern and activity rate. Temporary deactivation is implemented to unfix and page-release the SB buffers.
- Temporarily deactivate monitoring of the I/O reference pattern and activity rate. This form of temporary deactivation is implemented only if SB is deactivated and IMS concludes from subsequent evaluations that use of SB would still not be beneficial.

When SB is temporarily deactivated, it can be reactivated later based on the results of subsequent evaluations.

Individual periodic evaluations are performed for each DB-PCB/DSG pair. Therefore, IMS can deactivate SB for one DB-PCB/DSG pair while SB remains active for other DB-PCB/DSG pairs.

Role of the SB buffer handler

When SB is activated for a DB-PCB/DSG pair, a pool of SB buffers is allocated to the pair. Each SB buffer pool consists of *n* buffer sets (the default is four) and each buffer set contains 10 buffers. These buffers are used by an internal component that is called the SB buffer handler to hold the sets of 10 consecutive blocks that are read with sequential reads.

While SB is active, all requests for database blocks not found in the OSAM buffer pool are sent to the SB buffer handler. The SB buffer handler responds to these requests in the following way:
- If the requested block is already in an SB buffer, a copy of the block is put into an OSAM buffer.
- If the requested block is not in an SB buffer, the SB buffer handler analyzes a record of previous I/O requests and decides whether to issue a sequential read or a random read. If it decides to issue a random read, the requested block is read directly into an OSAM buffer. If it decides to issue a sequential read, the requested block and nine adjacent blocks are read into an SB buffer set. When the sequential read is complete, a copy of the requested block is put into an OSAM buffer.
- The SB buffer handler also decides when to initiate overlapped sequential reads.

Note: When the SB buffer handler is processing a request from an online program, the SB buffer handler only searches the SB buffer pools allocated to that online program.

Related concepts:

Chapter 20, "Optional database functions," on page 387

"Virtual storage considerations for SB"

➡ Sysplex data-sharing concepts and terminology (System Administration)

Virtual storage considerations for SB

Each DB-PCB/DSG pair buffered by SB has its own SB buffer pool.

By default, each SB buffer pool contains four buffer sets (although IMS lets you change this value). Ten buffers exist in each buffer set. Each buffer is large enough to hold one OSAM data set block.

The total size of each SB buffer pool is:

```
 4 * 10 * block size
```

The SB buffers are page-fixed in storage to eliminate page faults, reduce the path length of I/O operations, and increase performance. SB buffers are page-unfixed and page-released when a periodic evaluation temporarily deactivates SB.

You must ensure that the batch, online or DBCTL region has enough virtual storage to accommodate the SB buffer pools. This storage requirement can be considerable, depending upon the block size and the number of programs using SB.

SB is not recommended in real storage-constrained environments such as batch and DB/TM.

Some systems are storage-constrained only during certain periods of time, such as during online peak times. You can use an SB Initialization Exit Routine to control the use of SB according to specific criteria (the time) of day.

Related concepts:

"How SB buffers data" on page 447

Related tasks:

"Requesting SB with SB control statements" on page 450

Related reference:

➡ Sequential Buffering Initialization exit routine (DFSSBUX0) (Exit Routines)

How to request the use of SB

IMS provides two methods for specifying which of your programs and databases use OSAM sequential buffering (SB).

- You can explicitly specify which programs and utilities should use SB during PSB generation or by using SB control statements.
- You can specify that by default all or a subset of your batch and BMP programs and utilities should use SB by coding an SB exit routine or by using a sample SB exit routine provided with IMS.

Determine which method you will use. Using the second method is easier because you do not need to know which BMP and batch programs use sequential

processing. However, using SB by default can lead to an uncontrolled increase in real and virtual storage use, which can impact system performance. Generally, if you are running IMS in a storage-constrained z/OS environment, use the first method. If you are running IMS in a non-storage-constrained z/OS environment, use the second method.

Related concepts:

"Flexibility of SB use" on page 447

Requesting SB during PSB generation

To request SB during PSB generation, specify SB=COND in the PCB macro instruction of your application's PSB.

You code this keyword for each database PCB buffered with SB. (This is not possible for IMS utilities that do not use a PSB during execution.)

The following diagram shows the syntax of the SB keyword in the PCB statement.

```
►►──PCB──TYPE=DB,──┤ Other parameters ├──────────────────────────────►◄
                                        └─SB=─┬─NO──┬─┘
                                              └─COND─┘
```
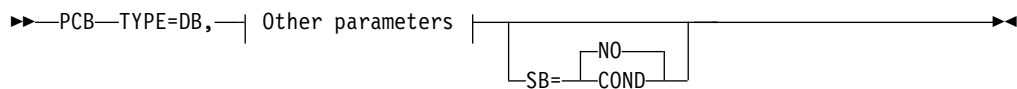
COND

        Specifies that SB should be conditionally activated for this PCB.

NO   Specifies that SB should not be used for this PCB.

        If you do not include the SB keyword in your PCB, IMS defaults to NO unless specified otherwise in the SB exit routine.

The SB keyword value can be overridden by SB control statements.

The following example shows a PCB statement coded to request conditional activation of SB:

```
SKILLA   PCB   TYPE=DB,DBDNAME=SKILLDB,KEYLEN=100,
               PROCOPT=GR,SB=COND
```

Related tasks:

"Requesting SB with SB control statements"

Related reference:

⮕ Program Specification Block (PSB) Generation utility (System Utilities)

Requesting SB with SB control statements

You can put SBPARM control statements in the optional //DFSCTL file. This file is defined by a //DFSCTL DD statement in the JCL of your batch, dependent, or online region.

You can use the SBPARM control statement to:

- Specify which database PCBs (and which data sets referenced by the database PCB) should use SB
- Override the default number of buffer sets

This control statement allows you to override PSB specifications without requiring you to regenerate the PSB.

You can specify keywords that request use of SB for all or specific DBD names, DD names, PSB names, and PCB labels. You can also combine these keywords to further restrict when SB is used.

By using the BUFSETS keyword of the SBPARM control statement, you can change the number of buffer sets allocated to SB buffer pools. The default number of buffer sets is four. Badly organized databases can require six or more buffer sets for efficient sequential processing. Well-organized databases require as few as two buffer sets. An indicator of how well-organized your database is can be found in the optional //DFSSTAT reports.

The example below shows the SBPARM control statement necessary to request conditional activation of SB for all DBD names, DD names, PSB names, and PCBs.

```
SBPARM ACTIV=COND
```

The next example shows the parameters necessary to:

- Request conditional activation of SB for all PCBs that were coded with 'DBDNAME=SKILLDB' during PSB generation
- Set the number of buffer sets to 6

```
SBPARM ACTIV=COND,DB=SKILLDB,BUFSETS=6
```

Related concepts:

"Virtual storage considerations for SB" on page 449

"Adjusting buffers" on page 701

"Buffering options" on page 440

➦ //DFSSTAT reports (System Administration)

Related tasks:

"Requesting SB during PSB generation" on page 450

Related reference:

➦ Sequential buffering control statements (System Definition)

Requesting SB with an SB Initialization exit routine

An SB exit routine allows you to dynamically control the use of SB at application scheduling time.

You can use an SB Initialization exit routine to:

- Request conditional activation of SB for all or some batch and BMP programs
- Allow or disallow the use of SB
- Change the default number of buffer sets

You can do this by writing your own SB exit routine or by selecting a sample SB exit routine and copying it under the name DFSSBUX0 into IMS.SDFSRESL.

IMS supplies five sample SB exit routines in IMS.SDFSSRC and IMS.SDFSRESL. Three of the sample routines request SB for various subsets of application programs and utilities. One sample routine requests SB during certain times of the day and another routine disallows use of SB. You can use these sample routines as written or modify them to fit your needs.

Detailed instructions for the SB Initialization Exit Routine are in *IMS Version 15 Exit Routines*.

SB options or parameters provided by several sources

If you provide the same SB option or parameter in more than one place, IMS determines which option to enforce by following a set order of precedence.

If you provide the same SB option or parameter in more than one place, the following priority list applies (item 1 having the highest priority):

1. SB control statement specifications (the *n*th control statement overrides the *m*th control statement, where $n > m$)
2. PSB specifications
3. Defaults changed by the SB Initialization Exit Routine
4. IMS defaults

Using SB in an online system

To allow the use of SB in an online IMS or DBCTL environment, an IMS system programmer must explicitly request that IMS load the SB modules. This is done by putting an SBONLINE control statement in the DFSVSMxx member.

By default, IMS does not load SB modules in an online environment. This helps avoid a noticeable increase in virtual storage requirements.

The two forms of the SBONLINE control statement are:

```
SBONLINE

  or

SBONLINE,MAXSB=nnnnn
```

where *nnnnn* is the maximum storage (in kilobytes) that can be used for SB buffers.

When the MAXSB limit is reached, IMS stops allocating SB buffers to online applications until terminating online programs release SB buffer space. By default, if you do not specify the MAXSB= keyword, the maximum storage for SB buffers is unlimited.

Detailed instructions for coding the SBONLINE control statement are contained in *IMS Version 15 System Definition*.

Disallowing the use of SB

An IMS system programmer or MTO can disallow the use of SB.

When the use of SB has been disallowed, a request for conditional activation of SB is ignored.

There are three ways to disallow the use of SB. The following list describes the three methods:

1. An SB Initialization Exit Routine can be written (or a sample exit routine adapted) that can dynamically disallow and allow use of SB. This method can be used if you are using SB in an IMS batch, online, or DBCTL environment.
2. The MTO commands /STOP SB and /START SB can be issued to dynamically disallow and allow use of SB within an IMS online subsystem
3. The SBONLINE control statement can be omitted from the DFSVSMxx member. This will keep IMS from loading the SB modules into the online subsystem. No program in the online subsystem will be able to use SB.

Related reference:

VSAM options

Several types of options can be chosen for databases that use VSAM.

Specifying options such as free space for the ESDS data set, logical record size, and CI size are discussed in the preceding topics in this chapter. This topic describes these optional functions:

- Functions specified in the OPTIONS control statement when IMS is initialized.
- Functions specified in the POOLID, VSRBF, and DBD control statements when IMS is initialized.
- Functions specified in the Access Method Services DEFINE CLUSTER command when a data set is defined.

Optional functions specified in the OPTIONS control statement

Several options exist that can be chosen during IMS system initialization for databases using VSAM. These options are specified in the OPTIONS control statement. In a batch system, the options you specify are put in the data set with the DDNAME DFSVSAMP. In an online system, they are put in the IMS.PROCLIB data set with the member name DFSVSMnn. Your choice of VSAM options can affect performance, use of space in the database, and recovery. This topic describes each option and the implications of using it.

Using background write (BGWRT parameter)

When an application program issues a call requiring that data be read from the database, the data is read into a buffer. If the buffer the data is to be read into contains altered data, the altered data must be written back to the database before the buffer can be used. If the data was not written back to the database, the data would be lost (overlaid) when new data was read into the buffer. Then there would be no way to update the database.

For these reasons, when an application program needs data read into a buffer and the buffer contains altered data, the application program waits while the buffer is written to the database. This waiting time decreases performance. The application program is ready to do processing, but the buffer is not available for use. Background write is a function you can choose in the OPTIONS statement that reduces the amount of wait time lost for this reason.

To understand how background write works, you need to know something about how buffers are used in a subpool. You specify the number of buffers and their size. All buffers of the same size are in the same subpool. Buffers in a subpool are on a use chain, that is, they are chained together in the order in which they have been most or least recently used. The most recently used buffers are at the top of the use chain; least recently used buffers are at the bottom.

When a buffer is needed, the VSAM buffer manager selects the buffer at the bottom of the use chain. The buffer at the bottom of the use chain is selected, because buffers that have not been used recently are less likely to contain data that will be used again. If the buffer the VSAM buffer handler picks contains altered

data, the data is written to the database before the buffer is used. It is during this step that the application program is waiting.

Background write solves the following problem: when the VSAM buffer manager gets a buffer in any subpool, it looks (when background write is used) at the next buffer on the use chain. The next buffer on the use chain will be used next. If the buffer contains altered data, IMS is notified so background write will be invoked. Background write has VSAM write data to the database from some percentage of the buffers at the bottom of the use chain. VSAM does this for *all* subpools. The data that is written to the database still remains in the buffers so the application program can still use any data in the buffers.

Background write is a very useful function when processing is done sequentially, but it is not as important to use in online systems as in batch. This is because, in online environments, IMS automatically writes buffers to the database at sync points.

To use background write, specify BGWRT=YES,n on the OPTIONS statement, where n is the percentage of buffers in each subpool to be written to the database. If you do not code the BGWRT= parameter, the default is BGWRT=YES and the default percentage is 34%. If an application program continually uses buffers but does not reexamine the data in them, you can make n 99%. Then, a buffer will normally be available when it is needed.

CICS does not support this function.

Choosing an insert strategy (INSERT parameter)

Get free space in a CI in a KSDS is by specifying it in the `DEFINE CLUSTER` command. Free space for a KSDS cannot be specified using the FRSPC= keyword in the DBD.

To specify free space in the `DEFINE CLUSTER` command, you must decide:
- Whether free space you have specified is preserved or used when more than one root segment is inserted at the same time into the KSDS.
- Whether to split the CI at the point where the root is inserted, or midway in the CI, when a root that causes a CI split is inserted.

These choices are specified in the INSERT= parameter in the OPTIONS statement. INSERT=SEQ preserves the free space and splits the CI at the point where the root is inserted. INSERT=SKP does not preserve the free space and splits the CI midway in the CI. In most cases, specify INSERT=SEQ so free space will be available in the future when you insert root segments. Your application determines which choice gives the best performance.

If you do not specify the INSERT= parameter, the default is INSERT=SKP.

Using the IMS trace parameters

The IMS trace parameters trace information that has proven valuable in solving problems in the specific area of the trace. All traces share sequencing numbers so that a general picture of the IMS environment can be obtained by looking at all the traces.

IMS DL/I, LOCK and retrieve traces are on by default, except in batch regions, where they are off by default. All other trace types are off by default.

The traces can be turned on at IMS initialization time. They can also be started or stopped by the /TRACE command during IMS execution. Output from long-running traces can be saved on the system log if requested.

Determining which dump option to use (DUMP parameter)

The dump option is a serviceability aid that has no impact on performance. It merely describes the type of abend to take place if an abend occurs in the buffer handler (an internal component). If DUMP=YES is specified, the control region will abend when there is an abend in the buffer handler.

Deciding whether to fix VSAM database buffers and IOBs in storage (VSAMFIX parameter)

Each VSAM subpool contains buffers and input/output control blocks (IOBs). Performance is generally improved if these buffers and IOBs are fixed in storage. Then, page faults do not occur. A page fault occurs when an instruction references a page (a specific piece of storage) that is not in real storage.

You can specify whether buffers and IOBs are fixed in storage in the VSAMFIX= parameter of the OPTIONS statement. If you have buffers or IOBs fixed, they are fixed in all subpools. If you do not code the VSAMFIX= parameter, the default is that buffers and IOBs are not fixed.

This parameter can be used in a CICS environment if the buffers were specified by IMS.

Using local shared resources (VSAMPLS parameter)

Specifying VSAMPLS=LOCL in the OPTIONS statement is for local shared resources (LSR). When you specify VSAMPLS=LOCL, VSAM control blocks and subpools are put in the IMS control region. VSAMPLS=LOCL is the only valid operand and the default.

Related concepts:

"Background write option" on page 441

"Adjusting VSAM options" on page 709

Related tasks:

"Specifying buffers" on page 444

"Specifying free space for a KSDS (FREESPACE parameter)" on page 456

Related reference:

DFSVSMxx member of the IMS PROCLIB data set (System Definition)

Optional functions specified in the POOLID, DBD, and VSRBF control statements

Options chosen during IMS initialization determine the size and structure of VSAM local shared resource pools.

In a batch environment, you specify these options in a data set with the DDNAME DFSVSAMP. In online systems, you specify these options in the IMS.PROCLIB data set with the member name DFSVSMnn.

With these options, you can enhance IMS performance by:
- Defining multiple local shared resource pools
- Dedicating subpools to a specific data set
- Defining separate subpools for index and data components of VSAM data sets

Related concepts:

➡ Specifying VSAM and OSAM subpools (System Definition)

Related reference:

➡ DFSVSMxx member of the IMS PROCLIB data set (System Definition)

Optional functions specified in the Access Method Services DEFINE CLUSTER command

You can choose several optional functions that affect performance when you define your VSAM data sets.

These functions are specified in the Access Method Services DEFINE CLUSTER command. HALDB databases require that the REUSE parameter be specified on the DEFINE CLUSTER command. IMS Online Recovery Services takes advantage of the REUSE parameter, if it is specified.

Related Reading: This command and all its parameters are described in detail in *z/OS DFSMS Access Method Services for Catalogs*.

Specifying that 'fuzzy' image copies can be taken with the database image copy 2 utility (DFSUDMT0)

To establish that 'fuzzy' image copies of KSDSs can be taken with the Database Image Copy 2 utility (DFSUDMT0), the KSDS must be SMS-managed and you must specify the BWO(TYPEIMS) parameter on the AMS DEFINE or ALTER command.

Specifying free space for a KSDS (FREESPACE parameter)

To include free space in a CI in a KSDS, use the FREESPACE parameter in the `DEFINE CLUSTER` command.

Free space for a KSDS cannot be specified using the FRSPC= keyword in the DBD.

You specify free space in the FREESPACE parameter as a percentage. The format of the parameter is FREESPACE(x,y) where:

x   is the percentage of space in a CI left free when the database is loaded or when a CI split occurs after initial load

y   is the percentage of space in a control area (CA) left free when the database is loaded or when a CA split occurs after initial load.

Free space is preserved when a CI or CA is split by coding INSERT=SEQ in the OPTIONS control statement.

If you do not specify the FREESPACE parameter, the default is that no free space is reserved in the KSDS data set when the database is loaded.

Related tasks:

Specifying whether data set space is pre-formatted for initial load

When initially loading a VSAM data set, you can specify whether you need the data set pre-formatted in the SPEED | RECOVERY parameter.

When SPEED is specified, it says the data set should *not* be pre-formatted. An advantage of pre-formatting a data set is; if initial load fails, you can recover and continue loading database records after the last correctly-written record. However, IMS does *not* support the RECOVERY option (except by use of the Utility Control Facility). So, although you can specify it, you cannot perform recovery. Because you cannot take advantage of recovery when you specify the RECOVERY parameter, you should specify SPEED to improve performance during initial load.

To be able to recover your data set during load, you should load it under control of the Utility Control Facility. This utility is described in *IMS Version 15 Database Utilities*.

RECOVERY is the default for this parameter.

OSAM options

Two types of options are available for databases using OSAM.

Two types of options are:
1. Options specified in the DBD (free space, logical record size, CI size).

   These options are covered in preceding sections in this chapter.
2. Options specified in the OPTIONS control statement when IMS is initialized.

   In a batch system, the options are put in the data set with the DDNAME DFSVSAMP. In an online system, they are put in the IMS.PROCLIB data set with the member name DFSVSMnn. Your choice of OSAM options can affect performance, recovery, and the use of space in the database.

   The OPTIONS statement is described in detail in *IMS Version 15 System Definition*. The statement and all its parameters are optional.

Dump option (DUMP parameter)

The dump option is a serviceability aid that has no impact on performance.

It describes the type of abnormal termination to take place when abnormal termination occurs in the buffer handler (an internal component).

Planning for maintenance

In designing your database, remember to plan for maintenance. If your applications require, for instance, that the database be available 16 hours a day, you do not design a database that takes 10 hours to unload and reload.

No guideline we can give you for planning for maintenance exists, because all such plans are application dependent. However, remember to plan for it.

A possible solution to the problem just described is to make three separate databases and put them on different volumes. If the separate databases have different key ranges, then application programs could include logic to determine which database to process against. This solution would allow you to reorganize the

three databases at separate times, eliminating the need for a single 10-hour reorganization. Another solution to the problem if your database uses HDAM or HIDAM might be to do a partial reorganization using the Partial Database Reorganization utility.

In the online environment, the Image Copy utilities allow you to do some maintenance without taking the database offline. These utilities let you take image copies of databases or partitions while they are allocated to and being used by an online IMS system.

HALDB provides greatly improved availability for large databases. By partitioning large databases, you can perform offline maintenance on a single partition, while the remaining partitions remain available.

You can also reorganize HALDB databases online, which improves the performance of your HALDB without disrupting access to its data. If you plan to reorganize your HALDB online, make sure that there is enough DASD space to accommodate the reorganization process.

Related concepts:

Chapter 23. Designing Fast Path databases

After you determine the type of database and optional functions that best suit your application's processing requirements, you need to make a series of decisions about database design and the use of options.

This set of decisions primarily determines how well your database performs and how well it uses available space. These decisions are based on:

The type of database and optional functions you have already chosen

The performance requirements of your applications

How much storage you have available for use online

Design guidelines for DEDBs

To define a data entry database (DEDB), you must determine the size of the CI and UOW, and follow design guidelines.

Related concepts:

➡ Data sharing in IMS environments (System Administration)

DEDB design guidelines

The following list describes guidelines for designing DEDBs.

- Except for the relationship between a parent and its children, the logical structure (defined by the PCB) does not need to follow the hierarchical order of segment types defined by the DBD.

  For example, SENSEG statements for DDEP segments can precede the SENSEG statement for the SDEP segment. This implementation prevents unqualified GN processing from retrieving *all* SDEP segments before accessing the first DDEP segments.

- Most of the time, SDEP segments are retrieved all at once, using the DEDB Sequential Dependent Scan utility. If you later must relate SDEP segments to their roots, you must plan for root identification as part of the SDEP segment data.

- A journal can be implemented by collecting data across transactions using a DEDB. To minimize contention, you should plan for an area with more than one root segment. For example, a root segment can be dedicated to a transaction/region or to each terminal. To further control resource contention, you should assign different CIs to these root segments, because the CI is the basic unit of DEDB allocation.

- Following is a condition you might be confronted with and a way you might resolve it. Assume that transactions against a DEDB record are recorded in a journal using SDEP segments and that a requirement exists to interrogate the last 20 or so of them.

  SDEP segments have a fast insert capability, but on the average, one I/O operation is needed for each retrieved segment. The additional I/O operations could be avoided by inserting the journal data as both a SDEP segment and a DDEP segment and by limiting the twin chain of DDEP segments to 20 occurrences. The replace or insert calls for DDEP segments does not necessarily cause additional I/O, since they can fit in the root CI. The root CI is always accessed even if the only call to the database is an insert of an SDEP segment.

The online retrieve requests for the journal items can then be responded to by the DDEP segments instead of the SDEP segments.

- As physical DDEP twin chains build up, I/O activity increases. The SDEP segment type can be of some help if the application allows it.

  The design calls for DDEP segments of one type to be batched and inserted as a single segment whenever their number reaches a certain limit. An identifier helps differentiate them from the regular journal segments. This design prevents updates after the data has been converted into SDEP segments.

DEDB area design guidelines

When designing a DEDB, be aware of the considerations and guidelines related to the DEDB area.

The following considerations for DEDB area design also help explain why DEDBs are divided into areas:

- DEDBs should be divided into areas in a way that makes sense for the application programs.

  For example, a service bureau organization makes a set of applications available to its customers. The design calls for a common database to be used by all users of this set of applications. The area concept fits this design because the randomizing routine and record keys can be set so that data requests are directed to the user's area only. Furthermore, on the operational side, users can be given specific time slots. Their areas are allocated and deallocated dynamically without interrupting other services currently using the same DEDB.

  National or international companies with business locations spanning multiple time zones might take advantage of the partitioned database concept. Because not all areas must be online all the time, data can be spread across areas by time zone.

  Preferential treatment for specific records (specific accounts, specific clients, and so on.) can be implemented without using a new database, for example, by keeping more sequential dependent segments online for certain records. By putting together those records in one area, you can define a larger sequential dependent segment part and control the retention period accordingly.

- The impact of permanent I/O errors and severe errors can be reduced using a DEDB. DL/I requires that all database data sets, except for HALDB databases, be available all the time. With a DEDB, the data not available is limited only to the area affected by the failure. Because the DEDB utilities run at the level of the area, the recovery of the failing area can be done while the rest of the database is accessible to online processing. The currently allocated log volume must be freed by a /DBR AREA command and used in the recovery operation. Track recovery is also supported. The recovered area can then be dynamically allocated back to the operational environment.

  Make multiple copies of DEDB area data sets to make data more available to application programs.

- Space management parameters can vary from one area to another. This includes: CI size, UOW size, root addressable part, overflow part, and sequential dependent part. Also, the device type can vary from one area to the other.

- It is feasible to define an area on more than one volume and have one volume dedicated to the sequential dependent part. This implementation might save some seek time as sequential dependent segments are continuously added at the end of the sequential dependent part. The savings depends on the current size of

the sequential dependent part and the blocking factor used for sequential dependent segments. If an area spans more than one volume, volumes must be of the same type.

- Only the independent overflow part of a DEDB is extendable. Sufficient space should be provided for all parts when DEDBs are designed.

  The /DISPLAY command and the POS call can help monitor the usage of auxiliary space. Unused space in the root addressable and independent overflow parts can be reclaimed through reorganization. It should be noted that, in the overflow area, space is not automatically reused by ISRT calls. To be reused at call time, the space must amount to an entire CI, which is then made available to the ISRT space management algorithm. Local out-of-space conditions can occur, although some available space exists in the database.

- Adding or removing an area from a DEDB requires a DBDGEN and an ACBGEN. Database reload is required if areas are added or deleted in the middle of existing areas. Areas added other than at the end changes the area sequence number assigned to the areas. The subsequent log records written reflect this number, which is then used for recovery purposes. If areas are added between existing areas, prior log records will be invalid. Therefore, an image copy must be made following the unload/reload. Be aware that the sequence of the AREA statements in the DBD determines the sequence of the MRMB entries passed on entry to the randomizing routine. An area does not need to be mounted if the processing does not require it, so a DBDGEN/ACBGEN is not necessary to logically remove an area from processing.

- Careful monitoring of the retention period of each log allows you to make an image copy of one area at a time. Also, because the High-Speed DEDB Direct Reorganization utility logs changes, you do not need to make an image copy following a reorganization.

- The area concept allows randomizing at the area level, instead of randomizing throughout the entire DEDB. This means the key might need to carry some information to direct the randomizing routine to a specific area.

Related tasks:

"Multiple copies of an area data set" on page 465

"Extending DEDB independent overflow online" on page 807

Determining the size of the CI

The choice of a CI size depends on several factors.

The factors include:

- CI sizes of 512, 1 KB, 2 KB, 4 KB, and up to 28 KB in 4 KB increments are supported.
- Only one RAP exists per CI. The average record length has to be considered. In the base section of the root addressable part, a CI can be shared only by the roots that randomize to its RAP and their DDEP segments.
- Track utilization according to the device type.
- SDEP segment writes. A larger CI requires a fewer number of I/Os to write the same amount of SDEP segments.
- The maximum segment size, which is 28,552 bytes if using a 28 KB CI size.

Determining the size of the UOW

The UOW is the unit of space allocation in which you specify the size of the root addressable and independent overflow parts.

Three factors might affect the size of the UOW:

1. The High-Speed DEDB Direct Reorganization utility (DBFUHDR0) runs on a UOW basis. Therefore, while the UOW is being reorganized, none of the CIs and data they contain are available to other processing.

   A large UOW can cause resource contention, resulting in increased response time if the utility is run during the online period. A minor side effect of a large UOW is the space reserved on DASD for the "reorganization UOW," which is no longer used, but retained for compatibility purposes.

   A UOW that is too small can cause some overhead during reorganization as the utility switches from one UOW to the next with very little useful work each time. However, this might not matter so much if reorganization time is not critical.

2. The use of processing option P. This consideration pertains to sequential processing using BMP regions. If the application program is coded to take advantage of the 'GC' status code, this status code must be returned frequently enough to fit in the planned sync interval.

   Assume every root CI needs to be modified and that, for resource control reasons, each sync interval is allowed to process sequentially no more than 20 CIs of data. The size of the UOW should not be set to more than 20 CIs. Otherwise, the expected 'GC' status code would not be returned in time for the application program to trigger a sync point, release the resources, and not lose position in the database.

   A UOW that is too small, such as the minimum of two CIs, can cause too many 'unsuccessful database call' conditions each time a UOW is crossed. On a 'GC' status code, no segment is returned and the call must be reissued after an optional SYNC or CHKP call.

3. The dependent overflow (DASD space) usage is more efficient with a large UOW than a small UOW.

Related concepts:

"SDEP CI preallocation and reporting"

SDEP CI preallocation and reporting

Because of data sharing, SDEP CIs cannot be allocated one at a time. Also, each data sharing system requires its own current CI. Therefore, a set of SDEP CIs are preallocated to each IMS on an allocation call.

The number of CIs obtained by an IMS is a function of the system's insert rate. The insert process obtains the current CI, not the area open process.

Because the insert process obtains the current CI, space use and reporting is complex. If a preallocation attempt cannot obtain the number of CIs requested, the ISRT or sync point call receives status FS, even if there is enough space for that particular call. The FS processing marks the area as full, and any subsequent smaller inserts also fail.

When there are few available SDEP CIs in an area, the number that can actually be used for SDEP inserts varies depending on the system's insert rate. Also, the command /DIS AREA calculates the number of SDEP CIs free as those available for preallocation and any unused CIs preallocated to the IMS issuing the command. Area close processing discards CIs preallocated to the IMS, and the unused CIs are lost until the SDEP Delete utility is run. Therefore, the number of

unused CIs reported by the /DIS AREA command after area close processing is smaller because the preallocated CIs are no longer available.

The Automatic SDEP Buffer Management feature discards preallocated SDEP CIs that are older than a specified age. This eliminates the possibility of a low activity IMS system from holding these old preallocated CIs and preventing other high activity IMS systems from freeing up space in the SDEP section of the area.

Additionally, preallocated CIs become unavailable if either the DEDB Sequential Dependent Delete utility (DBFUMDL0) or the DEDB Sequential Dependent Scan utility (DBFUMSC0) requests to discard them during execution. The DBFUMDL0 and DBFUMSC0 utilities request to discard preallocated CIs if the QUITCI option is specified either by the SDEPQCI parameter of the DFSVSMxx PROCLIB member or the QUITCI control statement in the utility SYSIN data set. The SDEPQCI parameter sets the default QUITCI behavior for either the DBFUMDL0 utility, the DBFUMSC0 utility, or both. When SDEPQCI is specified for a utility, you do not need to include the QUITCI control statement when you run that utility.

Related concepts:

"Determining the size of the UOW" on page 461

Automatic management of Fast Path DEDB SDEP buffers

IMS can be configured to automatically clean up Fast Path DEDB sequential dependent segments (SDEPs) in online buffers by writing them to the physical database at intervals that you specify.

If there is an imbalance in workload among multiple IMSes, you might be unable to physically delete SDEPs from a DEDB Area. To avoid this problem, this enhancement enables automatic periodic clean up of the SDEPs.

To enable this function, you need to specify parameters in the FASTPATH section of the DFSDFxxx member of the IMS PROCLIB data set. You can specify either a time of day or a frequency for the SDEP buffers to be cleaned up.

Related reference:

➡ FASTPATH section of the DFSDFxxx member (System Definition)

Processing option P (PROCOPT=P)

The PROCOPT=P option is specified during the PCB generation in the PCB statement or in the SENSEG statement for the root segment.

The option takes effect only if the region type is a BMP. If specified, it offers the following advantage:

Whenever an attempt is made to retrieve or insert a DEDB segment that causes a UOW boundary to be crossed, a 'GC' status code is set in the PCB but no segment is returned or inserted. The only calls for which this takes place are: G(H)U, G(H)N, POS, and ISRT.

Although crossing the UOW boundary has no particular significance for most applications, the 'GC' status code that is returned indicates this could be a convenient time to invoke sync point processing. This is because a UOW boundary is also a CI boundary. As explained for sequential processing, a CI boundary is a convenient place to request a sync point.

The sync point is invoked by either a SYNC or a CHKP call, but this normally causes position on all currently accessed databases to be lost. The application program then has to resume processing by reestablishing position first. This situation is not always easy to solve, particularly for unqualified G(H)N processing.

An additional advantage with this processing option is, if a SYNC or CHKP call is issued after a 'GC' status code, database position is kept. Database position is such that an unqualified G(H)N call issued after a 'GC' status code returns the first root segment of the next UOW. When a 'GC' status code is returned, no data is presented or inserted. Therefore, the application program should, optionally, request a sync point, reissue the database call that caused the 'GC' status code, and proceed. The application program can ignore the 'GC' status code, and the next database call will work as usual.

Database recovery and change accumulation processing must buffer all log records written between sync points. Sync points must be taken at frequent intervals to avoid exhausting available storage. If not, database recovery might not be possible.

Related concepts:

DEDB randomizing routine design

A DEDB randomizing module is required for placing root segments in a DEDB and for retrieving root segments from a DEDB.

One or more such modules can be used with an IMS system. Only one randomizing module can be associated with each DEDB.

The purpose of the randomizing module is the same as in HDAM processing. A root search argument key field value is supplied by the application program and converted into a relative root anchor point number. Because the entry and exit interfaces are different, DEDB and HDAM randomizing routines are not object code compatible. The main line randomizing logic of HDAM should not need modification if randomizing through the whole DEDB.

Some additional differences between DEDB and HDAM randomizing routines are as follows:

- The ISRT algorithm attempts to put the entire database record close to the root segment (with the exception of SDEP segments). No BYTES parameter exists to limit the size of the record portion to be inserted in the root addressable part.
- With the DEDB, only one RAP can be defined in each root addressable CI.
- CIs that are not randomized to are left empty.

The standard randomizer randomly distributes database records across the entire database. A two-stage randomizer is defined with the RMNAME parameter. The two-stage randomizer selects the specific area and the RAP within that area as the target for that database record. Use the two-stage randomizer to change the UOW or RAP parameters for a specific area using online change, and without affecting other areas in the DEDB. To alter a DEDB database or area by using the DEDB Alter utility, a two-stage randomizing routine is required.

Recommendation: Use a two-stage randomizing routine.

Because of the area concept, some applications might randomize in a particular area rather than through all the DEDBs, as in HDAM processing. Therefore, the expected output of such a randomizing module is made up of a relative root anchor point number in an area and the address of the control block (DMAC) representing the area selected.

Keys that randomize to the same RAP are chained in ascending key sequence.

DEDB logic runs in parallel, so DEDB randomizing routines must be reentrant. The randomizing routines operate out of the common storage area (CSA). If they use operating system services like LOAD, DELETE, GETMAIN, and FREEMAIN, the routines must follow the same rules as described in *IMS Version 15 Exit Routines*.

DEDB randomizing routines are stored in ECSA storage and are usually unloaded from storage when access to the DEDB is stopped by issuing either the type-1 command `/DBRECOVERY DB` or the type-2 command `UPDATE DB STOP(ACCESS)`. You can stop access to the DEDB without unloading the DEDB randomizer by using the NORAND parameter of the type-2 command `UPDATE DB STOP(ACCESS)`.

Related reference:

↪ Sample data entry database randomizing routines (DBFHDC40 / DBFHDC20 DBFHDC44 / DBFHDC24 DBFHDC2S) (Exit Routines)

Multiple copies of an area data set

The data in an area is in a VSAM data set called the area data set (ADS). Installations can create as many as seven copies of each ADS, making the data more available to application programs.

When multiple copies of an area data set are used, the ADS is referred to as a *multiple area data set* (MADS).

Each copy of a MADS contains exactly the same user data. Fast Path maintains data integrity by keeping identical data in the copies during application processing. When an application program updates data in an area, Fast Path updates that data in each copy of the MADS.

When an application program reads data from an area, IMS always attempts to read from the first ADS shown in the RECON list. If the first ADS is not available or if it is in a long busy state, IMS attempts to read from each subsequent ADS in the list until an available ADS is found. If all of the ADSs are in a long busy state, IMS uses the first ADS in the list.

All copies of a MADS must have the same definition but can reside on different devices and on different device types.

If your MADS copies reside on different devices, place the first ADS registered in the RECON data set on your fastest DASD for the best read performance. Subsequent copies of the ADS can reside on slower DASD without affecting overall read performance.

Using MADS can also be helpful in DASD migration; for example, from a 3380 device to a 3390 device.

To create a copy of an area data set, issue the DBRC command `INIT.ADS`. In the `AREA(name)` parameter, specify the name of the original ADS as it is recorded in the RECON data set. Issue the `INIT.ADS` command for each additional copy. For more

information about creating MADS, see the `INIT.ADS` command in *IMS Version 15 Commands, Volume 3: IMS Component and z/OS Commands*.

If an ADS fails to open during normal open processing of a DEDB, none of the copies of the ADS can be allocated, and the area is stopped. However, when open failure occurs during emergency restart, only the failed ADS is deallocated and stopped. The other copies of the ADS remain available for use.

Related concepts:

"DEDB area design guidelines" on page 460

Record deactivation

If an error occurs while an application program is updating a DEDB, it is not necessary to stop the database or the area.

IMS continues to allow application programs to access that area, and it only prevents them from accessing the control interval in error. If multiple copies of the ADS exist, one copy of the data is always available. (It is unlikely that the same control interval is in error in seven copies of the ADS.) IMS automatically deactivates a record when a count of 10 errors is reached.

Record deactivation minimizes the effect of database failures and errors to the data in these ways:

- If multiple copies of an area data set are used, and an error occurs while an application program is trying to update that area, the error does not need immediate correction. Other application programs can continue to access the data in that area through other available copies of that area.
- If a copy of an area has errors, you can create a new copy from existing copies of the ADS using the DEDB Data Set Create utility. The copy with the errors can then be destroyed.

Physical child last pointers

The PCL pointer makes it possible to access the last physical child of a segment type directly from the physical parent. Using the INSERT rule LAST avoids the need to follow a potentially long physical child pointer chain.

Subset pointers

Subset pointers help you avoid unproductive get calls when you need to access the last part of a long segment chain.

These pointers divide a chain of segment occurrences under the same parent into two or more groups, or subsets. You can define as many as eight subset pointers for any segment type, dividing the chain into as many as nine subsets. Each subset pointer points to the start of a new subset.

<u>Restrictions</u>: When you unload and reload a DEDB containing subset pointers, IMS does not automatically retain the position of the subset pointers. When unloading the DEDB, you must note the position of the subset pointers, storing the information in a permanent place. (For example, you could append a field to each segment, indicating which subset pointer, if any, points to that segment.) Or, if a segment in a twin chain can be uniquely identified, identify the segment a subset pointer is pointing to and add a temporary indication to the segment for reload. When reloading the DEDB, you must redefine the subset pointers, setting them to the segments to which they were previously set.

Designing a main storage database (MSDB)

This topic describes the choices you might need to make in designing an MSDB and proposes guidelines to help you make these choices.

Consider the following list of questions when designing an MSDB database:
- How are virtual storage requirements for the database calculated?
- How are virtual storage requirements for the Fast Path buffer pool calculated?
- What are the storage requirements for the I/O area?
- Should FLD calls or other DL/I calls be used for improved MSDB and DEDB performance?
- How can the difference in resource allocation between an MSDB and a DL/I database be a key to good performance?
- What are the requirements in designing for minimum resource contention in a mixed-mode environment?
- How is the number of MSDB segments loaded into virtual storage controlled?
- What are the auxiliary storage requirements for an MSDB?
- How can an MSDB be checkpointed?

Calculating virtual storage requirements for an MSDB

You can calculate the storage requirements for an MSDB by using a formula.

The formula for calculating the storage requirements for an MSDB is as follows:

`(L + 4)S + C + 14F + X`

where:

S = the number of segments in the MSDB as specified by the member DBFMSDBx in the IMS.PROCLIB
L = the segment length as specified in the DBD member

C = 80 for non-related MSDBs without a terminal-related key, *or*
94 for the other types of MSDB

F = the number of fields defined in the DBD member

X = 2 if C + 14F is not a multiple of 4, OR
0 if C + 14F is a multiple of 4

MSDBs reside in the z/OS extended common storage area (ECSA).

Considerations for MSDB buffers

When calculating buffer requirements for an MSDB database, you should be aware of a few considerations.

The following considerations apply during execution:
- Fast Path buffer requirements vary with the type of call to the MSDB.

- With a GHx/REPL call sequence, an entire segment is kept in the Fast Path buffer until a sync point is reached. If the total size of a series of segments exceeds the NBA (normal buffer allocation), the NBA parameter needs to be adjusted rather than using the OBA (overflow buffer) on a regular basis. You should accommodate the total number of segments used between sync points.
- When using a FLD call, the VERIFY and CHANGE logic reside in the Fast Path buffer.

Related concepts:

"Designing a DEDB or MSDB buffer pool" on page 476

Calculating the storage for an application I/O area

A GHx/REPL call requires an I/O area large enough to accommodate the largest segment to be processed. The FLD call requires storage to accommodate the total field search argument (FSA) requirements.

Understanding resource allocation, a key to performance

The MSDB resource allocation scheme is different from that of DL/I.

For performance reasons, the understanding of resource allocation for MSDBs is important.

An MSDB record can be shared by multiple users or be owned exclusively by one user. The same record can have both statuses (shared and exclusive) at the same time.

Updates to MSDBs are applied during sync point processing. The resource is always owned in exclusive mode for the duration of sync point processing.

The different enqueue levels of an MSDB record, when a record is enqueued, and the duration are summarized in the following table.

Table 67. Levels of enqueue of an MSDB record

Enqueue level	When	Duration
READ	GH with no update intent	From call time until sync point (phase 1)[1]
	VERIFY/get calls	Call processing
HOLD	GH with update intent	From call time until sync point (phase 1)[1]
	At sync point, to reapply VERIFYs	Phase 1 of sync point processing, then released
UPDATE[2]	At sync point, to apply the results of CHANGE, REPL, DLET, or ISRT calls	Sync point processing, then released

Notes:

1. If there was no FLD/VERIFY call against this resource or if this resource is not going to be updated, it is released. Otherwise, if only FLD/VERIFY logic has to be reapplied, the MSDB record is enqueued at the HOLD level. If the same record is involved in an update operation, it is enqueued at the UPDATE level as shown in the table above.
2. At DLET/REPL call time, no enqueue activity takes place because it is the prior GH call that set up the enqueue level.

The following table shows that the status of an MSDB record depends on the enqueue level of each program involved. Therefore, it is possible for an MSDB record to be enqueued with the shared and exclusive statuses at the same time. For example, such a record can be shared between program A (GH call for update) and program B (GU call), but cannot be shared at the same time with a third program, C, which is entering sync point with update on the record.

Table 68. Example of MSDB record status: Shared (S) or Owned Exclusively (E).

Enqueue level in program B	Enqueue level in program A		
	READ	HOLD	UPDATE
READ	Shared	Shared	Exclusive
HOLD	Shared	Exclusive	Exclusive
UPDATE	Exclusive	Exclusive	Exclusive

The FLD/CHANGE call does not participate in any allocation; therefore, FLD/CHANGE calls can be executed even though the same database record is being updated during sync point processing.

If FLD/CHANGE and FLD/VERIFY calls are mixed in the same FLD call, when the first FLD/VERIFY call is encountered, the level of enqueue is set to READ for the remainder of the FLD call.

Designing to minimize resource contention

One reason to use an MSDB is its fast access to data and high availability for processing.

To maintain high availability, you should design to avoid the contention for resources that is likely to happen in a high transaction rate environment.

The following is a list of performance-related considerations. Some of the considerations do not apply exclusively to MSDBs, but they are listed to give a better understanding of the operational environment.

- Access by Fast Path transactions to DL/I databases and use of the alternate PCB should be kept to a minimum. Use of the alternate PCB should be kept to a minimum because FP transactions must contend for resources with IMS transactions (some of which could be long running). Also, common sync point processing is invoked and entirely serialized in the IMS control region.
- To avoid resource contention when sharing MSDBs between Fast Path and DL/I transactions, You should try to make commit processing often and to avoid long-running scans.
- GH for read/update delays any sync point processing that intends to update the same MSDB resource. Therefore, GH logic should be used only when you assume the referenced segments will not be altered until completion of the transaction. If the resource is being updated, release is at the completion of sync point. Otherwise, the release is at entry to sync point.
- The following consideration deals with deadlock prevention. Deadlock can occur if transactions attempt to acquire (GH calls) multiple MSDB resources.

  Whenever a request for an MSDB resource exists that is already allocated and the levels involved are HOLD or UPDATE, control is passed to IMS to detect a potential deadlock situation. Increase in path length and response time results. The latter can be significant if a deadlock occurs, thus requiring the pseudo abend of the transaction.

In order to reduce the likelihood of deadlocks caused by resource contention, sync point processing enqueues (UPDATE level) MSDB resources in a defined sequence. This sequence is in ascending order of segment addresses. MSDB segments are acquired in ascending order of keys within ascending order of MSDB names, first the page-fixed ones then the pageable MSDBs.

The application programmer can eliminate potential deadlock situations at call time by also acquiring (GH calls) MSDB resources using the same sequence.

- From the resource allocation scheme discussed earlier, you probably realize that FLD logic should be used whenever possible instead of GH/REPL logic.
  - The FLD/VERIFY call results in an enqueue at the READ level, and if no other levels are involved, then control is not passed to IMS. This occurrence results in a shorter path length.
  - The FLD/CHANGE call, when not issued in connection with VERIFY logic does not result in any enqueue within either Fast Path or IMS.
  - FLD logic has a shorter path length through the Program Request Handler, since only one call to process exists instead of two needed for GH/REPL logic.
  - The FLD/CHANGE call *never* waits for any resource, even if that same resource is being updated in sync point processing.
  - The FLD/VERIFY call waits only for sync point processing during which the same resource is being updated.
  - With FLD logic, the resource is held in exclusive mode only during sync point processing.

In summary, programming with FLD logic can contribute to higher transaction rates and shorter response times.

The following examples show how the MSDB record is held in exclusive mode:
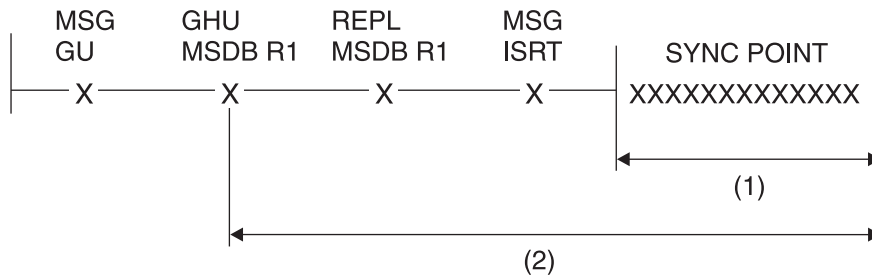
Figure 228. First example MSDB record held in exclusive mode

The following notes are for the preceding figure:
1. MSDB record R1 is held in exclusive mode against:
   - Any MSDB calls except CHANGE calls
   - Any other sync point processing that intends to update the same record
2. MSDB record R1 is held in exclusive mode against:
   - Any other GH for update
   - Any other sync point processing that intends to update the same record

```
MSG        VERIFY/CHANGE        MSG
GU         MSDB R1              ISRT          SYNC POINT
 X ─────────── X ──────────────── X ──── XXXXXXXXXXXXX
             (2)                                (1)
```

Figure 229. Second example MSDB record held in exclusive mode

The following notes are for the preceding figure.

1. MSDB record R1 is held in exclusive mode against:
   - Any MSDB calls except CHANGE calls
   - Any other sync point processing that intends to update the same record
2. MSDB record is held in exclusive mode for the duration of the FLD call against any other sync point processing that intends to update the same resource

Choosing MSDBs to load and page-fix

Deciding which MSDBs to load and page-fix involves a trade-off between desired application performance and the amount of real storage available. This decision is made with total Fast Path application requirements in mind.

IMS system initialization requires additional information before MSDBs can be loaded and page fixed. This information is specified in member DBFMSDBx of IMS.PROCLIB. This member is called by executing the control region startup procedure IMS. The suffix 'x' matches the parameter supplied in the MSDB keyword of the EXEC statement in procedure IMS.

The control information that loads and page fixes MSDBs is in 80-character record format in member DBFMSDBx. Either you supply this information or it can be supplied by the output of the MSDB maintenance utility. When the /NRE command requests MSDBLOAD, the definition of the databases to be loaded is found in the DBFMSDBx procedure.

The definition in DBFMSDBx can represent a subset of the MSDBs currently on the sequential data set identified by DD statement MSDBINIT. Explicitly state each MSDB that you want IMS to load. If each MSDB is not explicitly stated, IMS abends.

The format for DBFMSDBx is as follows:

```
►►──DBD=dbd_name,──NBSEGS=nnnnnnnnn──────────────────────────►◄
                                   └─,F─┘
```

dbd_name
>   The DBD name as specified during DBDGEN.

nnnnnnnnn
>   The number you specify of expected database segments for this MSDB. This number must be equal to or great than the number of MSDB segments loaded during restart.
>
>   The NBRSEGS parameter is also used to reserve space for terminal-related dynamic MSDBs for which no data has to be initially loaded.

F   The optional page-fix indicator for this MSDB.

If the MSDBs are so critical to your Fast Path applications that IMS should not run without them, place a first card image at the beginning of the DBFMSDBx member. For each card image, the characters "MSDBABND=n" must be typed without blanks, and all characters must be within columns 1 and 72 of the card image. Four possible card images exist, and each contains one of the following sets of characters:

MSDBABND=Y

This card image causes the IMS control region to abend if an error occurs while loading the MSDBs during system initialization. Errors include:

- Open failure on the MSDBINIT data set
- Error in the MSDB definition
- I/O error on the MSDBINIT data set

MSDBABND=C

This card image causes the IMS control region to abend if an error occurs while writing the MSDBs to the MSDBCP1 or MSDBCP2 data set in the initial checkpoint after IMS startup.

MSDBABND=I

This card image causes the IMS control region to abend if an error occurs during the initial load of the MSDBs from the MSDBINIT data set, making one or more of the MSDBs unusable. These errors include data errors in the MSDBINIT data set, no segments in the MSDBINIT data set for a defined MSDB, and those errors described under "MSDBABND=Y."

MSDBABND=A

This card image causes the IMS control region to abend if an error occurs during the writing of the MSDBs to the MSDBCPn data set (described in "MSDBABND=C"), or during the initial load of the MSDBs from the MSDBINIT data set (described in "MSDBABND=I").

MSDBABND=B

This card image causes the IMS control region to abend if an error occurs during the writing of the MSDBs to the MSDBCPn data set (described in "MSDBABND=C"), or during the loading of the MSDBs in system initialization (described in "MSDBABND=Y").

Auxiliary storage requirements for an MSDB

DASD space is needed to keep image copies of MSDBs when they are dumped at system and shutdown checkpoints. The data sets involved are the MSDBCP1 and MSDBCP2 data sets.

The same calculations apply to the MSDBDUMP data set, which contains a copy of the MSDBs following a /DBDUMP DATABASE MSDB command.

The data sets just discussed are written in 2K-byte blocks. Because only the first extent is used, the allocation of space must be on cylinder boundaries and be contiguous.

Space allocation is calculated like this:

`SPACE=(2048,(R),,CONTIG,ROUND)`

The calculation of the number of records (R) to be allocated can be derived from the formula:

```
(E + P + 2047)/2048
```

where:

E = main storage required, in bytes, for the Fast Path extension of the CNTs (ECNTs)

P = main storage required for all MSDBs as defined by the PROCLIB member DBFMSDBx

E is determined by the following formula:

```
E = (20 + 4D)T
```

where:

D = number of MSDBs using logical terminal names as keys

T = total number of logical terminal names defined in the system

High-speed sequential processing (HSSP)

High-Speed Sequential Processing (HSSP) is a function of Fast Path that handles sequential processing of DEDBs.

Benefits of the HSSP function

The high-speed sequential processing (HSSP) function provides a number of benefits.

Some of the benefits of the HSSP function include:
- HSSP generally has a faster response time than regular batch processing.
- HSSP optimizes sequential processing of DEDBs.
- HSSP reduces program execution time.
- HSSP typically produces less output than regular batch processing.
- HSSP reduces DEDB updates and image copy operation times.
- HSSP image copies can assist in database recovery.
- HSSP locks at UOW level to ease "bottle-necking" of cross IRLM communication.
- HSSP uses utility private buffers, allocated in either 31-bit extended common storage or 64-bit common storage, which reduces the impact on NBA/OBA buffers.
- HSSP allows for execution in both a mixed mode environment, concurrently with other programs, and in an IRLM-using global sharing environment.
- HSSP optimizes database maintenance by allowing the use of the image-copy option for an updated database.

Related concepts:

"Limitations and restrictions when using HSSP" on page 474

"HSSP processing option H (PROCOPT=H)" on page 475

Related tasks:

"Using HSSP" on page 474

Limitations and restrictions when using HSSP

Though HSSP can execute in a mixed-mode environment as well as concurrently with other programs, and in an environment with global sharing using IRLM; a program using HSSP can only execute as a non-message-driven BMP.

Other restrictions and limitations of HSSP include:
* Only one HSSP process can be active on an area at any given time. The /DIS AREA command identifies the IMSID of any HSSP job processing an area.
* HSSP processing and online utilities cannot process on the same area concurrently.
* Non-forward referencing while using HSSP is not allowed.
* Programs using HSSP must properly process the 'GC' status code by following it with a commit process.

Restrictions and limitations involving image copies include:
* The image copy option is available only for HSSP processing.
* The utility private buffers for HSSP processing must be located in 31-bit extended common storage (ECSA) and not in 64-bit common storage.
* HSSP image copying is allowed only if PROCOPT = H.
* The image copy process can only be done if a database is registered with DBRC. In addition, image copy data sets must be initialized in DBRC.

The following restrictions and limitations apply for PROCOPT=H:
* PROCOPT=H is allowed only for DEDBs.
* PROCOPT=H is not allowed on the segment level, only on the PCB level.
* Backward referencing while using HSSP is not allowed. You cannot use an HSSP PCB to refer to a prior UOW in a DEDB.
* Only one PROCOPT=H PCB per database per PSB is allowed.
* A maximum of four PROCOPTs can be specified, including H.
* PROCOPT=H must be used with other Fast Path processing options, such as GH and IH.
* When a GC status code is returned, the program must cause a commit process before any other call can be made to that PCB.
* An ACBGEN must be done to activate the PROCOPT=H.
* H is compatible with all other PROCOPTs except for PROCOPT=O.

Related concepts:

Using HSSP

To use HSSP, you must specify PROCOPT=H during PSBGEN.

Additionally, you need to make sure that the programs using HSSP properly process the 'GC' status code by following it with a commit process.

HSSP functions similarly to utilities, is subject to the same requirements as utilities, and uses utility private buffers in addition to common buffers.

HSSP includes the image-copy option, the ability to set area ranges, and the option to put the private utility buffers in 64-bit common storage. To use these functions, you need one or more of the following:

- The SETR statement
- The SETO statement
- The SETU statement
- A DFSCTL data set for the dependent regions
- DBRC
- PROCOPT=H

The utility private buffers for HSSP are located in 31-bit extended common storage (ECSA) unless you specify the SETU statement to locate the buffers in 64-bit common storage.

Related concepts:

"Benefits of the HSSP function" on page 473

Related reference:

➡️ High-speed sequential processing control statements (System Definition)

HSSP processing option H (PROCOPT=H)

PROCOPT=H is a PSBGEN OPTION. It allows you to define whether processing, with respect to a PCB, should be treated as an HSSP process.

The use of PROCOPT=H provides HSSP capability for the application program using this PSB. Following is an example of macros and keywords for a PSBGEN using PROCOPT=H:

```
Label      PCB TYPE = DB
          ,DBDNAME = name
          ,PROCOPT = AH
```

`Label` is an optional parameter of the PCB macro. It can be up to 8 characters long and is identical to the label on the associated SETO or SETR statements. H is compatible with any other Fast Path PROCOPT, except for PROCOPT=O, and PROCOPT=H can be used in one or more PCBs.

Each PCB that includes a specification of PROCOPT=H uses additional space in the EPCB pool in ECSA storage. To help size the EPCB pool, the output of the ACB Maintenance utility includes message DFS0943I, which lists the minimum and maximum amount of storage that the associated PSB requires in the EPCB pool. Until the PSB is scheduled, however, the exact amount of additional storage that the PSB requires cannot be predicted.

Related concepts:

"Benefits of the HSSP function" on page 473

"Limitations and restrictions when using HSSP" on page 474

Related reference:

➡️ High-speed sequential processing control statements (System Definition)

Image-copy option

Selecting the image-copy option with HSSP reduces the total elapsed times of DEDB updates and subsequent image-copy operations.

As database administrator, you decide whether to make an image copy of a database using HSSP. If you specify image copying, HSSP creates an asynchronous copy that is similar to a concurrent image copy.

The image copy process can only be done if a database is registered with DBRC. In addition, image copy data sets must be initialized in DBRC.

HSSP image copies can also be used for database recovery. However, the Database Recovery Utility must know that an HSSP image copy is supplied.

Related concepts:

Chapter 27, "Database backup and recovery," on page 589

➡ HSSP image copy (System Administration)

➡ IMS failure recovery (Operations and Automation)

UOW locking

In a globally shared environment, data is shared not only between IMS subsystems, but also across central processor complexes (CPC).

In such an environment, communication between two IRLMs could potentially "bottleneck" and become impeded. To ease this problem, HSSP locks at a UOW level in update mode, reducing the locking overhead. Non-HSSP or DEDB online processing locks at a UOW level in a shared mode. Otherwise, the locking for DEDB online processing is at the CI level. For information on UOW locking, refer to *IMS Version 15 System Administration*.

Private buffer pools

HSSP jobs use a combination of utility private buffers and common buffers (NBA/OBA), with the utility private buffers being primarily used.

HSSP dynamically allocates up to three times the number of CIs per area in one UOW, with each buffer being a CI in size.

HSSP uses the private buffers for reading RAP CIs and the common buffers for reading IOVF CIs. An FW status code may be received during the run of an HSSP job when NBA has been exceeded just as in a non-HSSP job.

The private buffer pools are located in 31-bit extended common storage (ECSA) or 64-bit common storage. The common buffers are in located in ECSA or, if the Fast Path 64-bit buffer manager is enabled, in 64-bit storage.

Designing a DEDB or MSDB buffer pool

Buffers needed to fulfill requests resulting from database calls are obtained from a global pool called the Fast Path buffer pool.

If you are using the Fast Path 64-bit buffer manager, IMS creates and manages the Fast Path buffer pools for you and places DEDB buffers in 64-bit storage. When the Fast Path 64-bit buffer manager is enabled, you do not need to design DEDB or MSDB buffer pools or specify the DBBF, DBFX, and BSIZ parameters that define Fast Path buffer pools.

You can enable the Fast Path 64-bit buffer manager by specifying FPBP64=Y and FPBP64M in the DFSDFxxx PROCLIB member. When the Fast Path 64-bit buffer manager is enabled, IMS ignores the DBBF, DBFX, and BSIZ parameters, if specified.

If you are not using the Fast Path 64-bit buffer manager, you must specify the characteristics of the pool yourself during IMS system definition and during IMS startup.

When specifying the characteristics yourself, three parameters characterize the Fast Path buffer pool:

DBBF
Total number of buffers.

The buffer pool is allocated at IMS startup in the ECSA or, if FPBUFF=LOCAL is specified in DFSFDRxx, in the FDBR private region. During emergency restart processing, the entire buffer pool can be briefly page-fixed. Consider the amount of available real storage when setting the DBBF value. IMS writes the total number of buffers to the X'5937' log.

For more information about page-fixing IMS resources, see *IMS Version 15 System Administration*.

DBFX
System buffer allocation.

This is a set of buffers that are page fixed during IMS initialization. The value should approximate the maximum number of buffers that are expected to be active in output thread processing at any one time. If the value is too small, dependent regions might have to wait for buffers.

BSIZ
Buffer size.

The size must be larger than or equal to the size of the largest CI of any DEDB to be processed. The buffer size can be up to 28 KB.

Related concepts:
"Considerations for MSDB buffers" on page 467

Fast Path buffer uses

Fast Path buffers are used to hold various types of Fast Path data.

Fast Path buffers are used to hold:
- Update information such as:
  - MSDB FLD/VERIFY call logic
  - MSDB FLD/CHANGE call logic
  - MSDB updates (results of REPL, ISRT, and DLET calls)
  - Inserted SDEP segments
- Referenced DEDB CIs from the root addressable part and the sequential dependent part.
- Updated DEDB CIs from the root addressable part.
- SDEP segments that have gone through sync point. The SDEP segments are collected in the current SDEP segment buffer. One such buffer allocated for each area defined with the SDEP segment type exists. This allocation takes place at area open time.

Fast Path 64-bit buffer manager

The Fast Path 64-bit buffer manager autonomically controls the number and size of Fast Path buffer pools, including buffer pools for data entry databases (DEDB), main storage databases (MSDB), and Fast Path system services. The Fast Path 64-bit buffer manager eliminates the need for system programmers to manually set buffer pool specifications during system definition.

The Fast Path 64-bit buffer manager also places the DEDB buffer pools above the bar in 64-bit control region private storage, which reduces the usage of ECSA storage.

When the Fast Path 64-bit buffer manager is used, the following items continue to be managed in ECSA storage:
- Fast Path buffer for MSDB databases
- Buffers for inserting sequential dependent (SDEP) segments
- Buffers for system services
- Buffer headers
- Internal IMS work areas for FLD calls and MSDBs

When the Fast Path 64-bit buffer manager is not used, the number and size of Fast Path buffer pools must be set during system definition by using the DBBF, DBFX, and BSIZ execution parameters, all Fast Path buffer pools are placed in 32-bit ECSA storage, dependent region access to overflow buffers is serialized, and, after IMS is started, the number and size of the Fast Path buffer pools cannot be changed without stopping and restarting IMS.

The Fast Path 64-bit buffer manager is enabled by one parameter, FPBP64, in the Fast Path section (SECTION=FASTPATH) of the DFSDFxxx PROCLIB member. When enabled, the Fast Path 64-bit buffer manager allocates additional buffer subpools when needed. Also, if a database is added to the online IMS system and none of the active buffer subpools can accommodate the CI size of the database, the Fast Path 64-bit buffer manager allocates a new buffer subpool of the appropriate size.

Requirements: The Fast Path 64-bit buffer manager requires:
- A minimum of 2.1 gigabytes of 64-bit storage.
- If the Fast Path 64-bit buffer manager is used on systems that are being tracked by a Fast Database Recovery (FDBR) address space, the DFSDF= keyword must be specified on the FDR procedure.

Because the Fast Path 64-bit buffer manager places the DEDB buffer pools in 64-bit storage, more buffer pools can be allotted to each dependent region that issues calls against a DEDB. For each additional buffer pool, the usage of ECSA storage increases only by the amount needed for the buffer header. The more buffers that a dependent region has available, the more work an application program can perform between checkpoints.

The Fast Path 64-bit buffer manager also enables multiple dependent regions to access overflow buffers in parallel, eliminating contention among dependent regions for overflow buffers.

The maximum number of buffers that can be allocated to a dependent region, both normal and overflow buffers combined, is determined by the NBA and OBA parameters in the dependent region definition and is not limited by the Fast Path 64-bit buffer manager.

When the Fast Path 64-bit buffer manager is enabled:

- You can change database CI sizes without having to adjust the buffer sizes to match. If no active buffer subpools can accommodate a new or changed CI size, the Fast Path 64-bit buffer manager automatically allocates a buffer subpool with the correct CI size.
- You can display statistics for Fast Path buffers by issuing the IMS type-2 command QUERY POOL TYPE(FPBP64).
- You can capture usage statistics for the Fast Path 64-bit buffers by issuing the command UPDATE IMS SET(LCLPARM(FPBP64STAT(Y))). IMS captures the usage statistics for each unit of work in a dependent region and writes the statistics to the online log data set as X'5945' log records, which are mapped by macros DBFL5945 and DBFBPND6.
- You can see if the logging of Fast Path 64-bit buffer usage statistics is enabled in an IMS system by using the QUERY IMS SHOW (ALL | LOCAL) command.

Related concepts:

"Designing a DEDB buffer pool in the DBCTL environment" on page 483

Normal buffer allocation (NBA)

Fast Path regions and IMS regions accessing Fast Path resources require that the number of buffers to be allocated as normal buffers be specified in the region startup procedure by using the NBA startup parameter.

Because buffers allocated as normal buffers are used first, the number of normal buffers allocated must accommodate most of the transaction requirements.

The number of buffers a transaction or a sync interval is allowed to use must be specified for each region if Fast Path resources are likely to be accessed.

The combined value of the NBA and OBA parameters define the maximum number of buffers that IMS can allocate to the region.

If you use the Fast Path 64-bit buffer manager, the number of normal buffers specified by the NBA parameter are page fixed in the Fast Path buffer pool when the buffers are allocated. If you do not use the Fast Path 64-bit buffer manager, the number of normal buffers specified by the NBA parameter are page fixed in the Fast Path buffer pool at the start of the region.

Overflow buffer allocation (OBA)

The overflow buffer allocation (OBA) is optional and is used for exceptional buffer requirements when the normal buffer allocation (NBA) has been exhausted.

If you use the Fast Path 64-bit buffer manager, access to the overflow buffers is multi-threaded and multiple regions can use overflow buffers at the same time.

If you do not use the Fast Path 64-bit buffer manager, the access to the overflow buffers by a region is dependent on obtaining a latch that serializes all regions currently in an overflow buffer state. If the latch is not available, the region has to wait until it is available. After the latch has been obtained, the NBA value is increased by the OBA value and normal processing resumes. The overflow buffer latch is released during sync point processing. At any point in time, only the largest OBA request among all the active regions is page fixed in the Fast Path buffer pool.

When the Fast Path 64-bit buffer manager is used, the combined value of the NBA and OBA parameters specified for a dependent region define the maximum number of buffers that the Fast Path 64-bit buffer manager can allocate to that region.

Fast Path buffer allocation algorithm

Fast Path buffers are allocated on demand up to a limit specified by the NBA parameter for each dependent region. Buffers so specified are called NBA to be used by one sync point interval.

Before satisfying any request from the NBA allocation, an attempt is made to reuse any already allocated buffer containing an SDEP CI. This process goes on until the NBA limit is reached. From that point on, a warning in the form of an FW status code returned to Fast Path database calls is sent to BMP regions. MD and MPP regions do not get this warning.

The next request for an additional buffer causes the buffer stealing facility to be invoked and then the algorithm examines each buffer and CI already allocated. As a result, buffers containing CIs being released are sent to a local queue (SDEP buffer chain) to be reused by this sync interval.

If, after invoking the buffer stealing facility, no available buffer is found, a request for the overflow buffer latch is issued. The overflow buffer latch governs the use of an additional buffer allocation called overflow buffer allocation (OBA). The OBA allocation is also specified as a parameter at region start time. From that point on, any time a request cannot be satisfied locally, a buffer is acquired from the OBA allocation until the OBA limit is reached. At that time, MD and BMP regions have their FW status code replaced by an FR status code after an internal ROLB call is performed. In MD and MPP regions, the transaction is abended and stopped.

Related concepts:

Fast Path buffer allocation when the DBFX parameter is used

From the total number of Fast Path buffers specified by the DBBF parameter, IMS set asides a number of the buffers to use for DEDB writes. The number of buffers that IMS sets aside is specified by the DBFX parameter.

The result of one transaction or sync interval is written back by one output thread. These output threads run from the control region in SRB mode. Buffers allocated to an output thread are therefore not available to dependent regions until after the CI they contain is written back.

If the Fast Path buffer pool is defined exactly as the sum of all NBAs, dependent regions must wait for the buffers to come back to the global pool. Fast Path regions can process the next transaction as soon as the sync point completes. Sync point processing does not wait for the output thread to complete. The allocation of buffers is page fixed at the start of the first region specifying an NBA request.

If you use the Fast Path 64-bit buffer manager, IMS manages the Fast Path buffers for you and you do not specify need to specify the DBFX, DBBF, or BSIZ parameters. The allocation of buffers is page fixed when the buffers are allocated.

Determining the Fast Path buffer pool size

If you are not using the Fast Path 64-bit buffer manager, you can calculate the number of buffers a Fast Path buffer pool must contain by using the formula `DBBF ≥ A + N + OBA + DBFX`.

The terms in the formula above are described as follows:

DBBF Fast Path buffer pool size as specified

A Number of active areas that have SDEP segments

NBA Normal buffer allocation of each active region

N Total of all NBAs

OBA Largest overflow buffer allocation

DBFX System buffer allocation

Fast Path buffer performance considerations

The performance considerations for Fast Path buffers differ depending on whether you are using the Fast Path 64-bit buffer manager.

If you are using the Fast Path 64-bit buffer manager to create and manage your Fast Path buffer pools, IMS optimizes many aspects of the buffer pool performance for you.

If you are not using the Fast Path 64-bit buffer manager, you might need to modify the buffer pool specifications yourself to maintain optimum performance of the buffer pools.

The following considerations apply to both buffers managed by the Fast Path 64-bit buffer manager and Fast Path buffers that are not managed by the Fast Path 64-bit buffer manager.

- An NBA value that is too large can increase the probability of contention (and delays) for other transactions. All CIs can be acquired at the exclusive level and be kept at that level until the buffer stealing facility is invoked. This occurrence happens after the NBA limit is reached. Therefore, an NBA that is too large can increase resource contention.
- A (NBA + OBA) value that is too small might result in more frequent unsuccessful processing. This means an 'FR' status code condition for BMP regions, or transaction abend for MD and MPP regions.
- Inquiry-only programs do not make use of an OBA specification, as buffers already allocated are reused when the NBA limit is reached.
- IMS logs information about buffers and their use to the X'5937' log. This information can be helpful in determining how efficiently the Fast Path buffers are being used.

The following considerations apply only when Fast Path buffers are not managed by the Fast Path 64-bit buffer manager.

- An incorrect specification of DBBF (too small) can result in the rejection of an area open or a region initialization. The system calculates the size of the buffer pool and rejects the open or initialization if the actual DBBF value is smaller.
- A DBFX value that is too small is likely to cause region waits and increase response time.

- An NBA value that is too small might cause the region processing to be serialized through the overflow buffer latch and again cause delays.

Related tasks:

"Determining the Fast Path buffer pool size" on page 481

The NBA limit and sync point

In BMP regions, when the NBA limit is reached, an 'FW' status code is returned. This status code is presented to every subsequent Fast Path database call until the OBA limit condition is reached.

The first occurrence of the 'FW' status code indicates no more NBA buffers exist. This occurrence is a convenient point at which to request a sync point. Fast Path resources (and others) would be released and the next sync point interval would be authorized to use a new set of NBA buffers. The overflow buffer latch serializes all the regions in an overflow buffer state and therefore causes delays in their processing.

If processing is primarily sequential, the sync point should be invoked on a UOW boundary crossing.

Related concepts:

"Processing option P (PROCOPT=P)" on page 463

The DBFX value and the low activity environment

If the IMS or Fast Path activity in the system is relatively low, log buffers are written less often, and therefore output threads are scheduled or dispatched less frequently. This situation is likely to result in many buffers waiting to be written and therefore could cause wait-for-buffer conditions.

To alleviate or avoid wait-for-buffer conditions you can enable the Fast Path 64-bit buffer manager, which manages Fast Path buffers for you dynamically. When the Fast Path 64-bit buffer manager is enabled, IMS tracks buffer usage, adds or removes buffers as needed, and ignores the DBBF, DBFX, and BSIZ parameters, if they are specified.

If you are not using the Fast Path 64-bit buffer manager, specify a larger DBFX value.

When the Fast Path 64-bit buffer manager is not used, a special case to be considered is the BMP region loading or processing a DEDB and being the only activity in the system. For example, assume an NBA of 20 buffers exists. To avoid a wait-for-buffer condition, the DBFX value must be specified as between one or two times the NBA value. This can result in a DBBF specification of three times the NBA number, which gives 60 buffers to the Fast Path buffer pool.

Except for the following case, there is no buffer look-aside capability across transactions or sync intervals (global buffer look-aside).

Assume that a region requests a DEDB CI resource that is currently being written or is owned by another region that ends up being written (output thread processing). Then, this CI and the buffer are passed to the requestor after the write (no read required) completes successfully. Any other regions must read it from disk.

Designing a DEDB buffer pool in the DBCTL environment

In a DBCTL environment, buffers needed to fulfill requests from database calls are obtained from a global pool called the Fast Path buffer pool.

If you are using the Fast Path 64-bit buffer manager, IMS creates and manages the Fast Path buffer pools for you and places DEDB buffers in 64-bit storage. When the Fast Path 64-bit buffer manager is enabled, you do not need to design DEDB or MSDB buffer pools or specify the DBBF, DBFX, and BSIZ parameters that define Fast Path buffer pools.

You can enable the Fast Path 64-bit buffer manager by specifying FPBP64=Y in the DFSDFxxx PROCLIB member. When the Fast Path 64-bit buffer manager is enabled, IMS ignores the DBBF, DBFX, and BSIZ parameters, if specified.

If you are not using the Fast Path 64-bit buffer manager, you must specify the characteristics of the pool yourself during IMS system definition and during IMS startup.

When specifying the characteristics yourself, three parameters characterize the Fast Path buffer pool:

DBBF
> Total number of buffers.
>
> The buffer pool is allocated at IMS startup in the ECSA or, if FPBUFF=LOCAL is specified in DFSFDRxx, in the FDBR private region. IMS writes the total number of buffers to the X'5937' log.

DBFX
> System buffer allocation.
>
> This is a set of buffers in the Fast Path buffer pool that is page fixed at startup of the first region with access to Fast Path resources.

BSIZ
> Buffer size.
>
> The size must be larger than or equal to the size of the largest CI of any DEDB to be processed. The buffer size can be up to 28 KB.

Related concepts:
"Fast Path 64-bit buffer manager" on page 478

Fast Path buffer uses in a DBCTL environment

Fast Path buffers are used to hold various types of Fast Path data.

Fast Path buffers are used to hold:
- Update information such as inserted SDEP segments.
- Referenced DEDB CIs from the root addressable part and the sequential dependent part.
- Updated DEDB CIs from the root addressable part.
- SDEP segments that have gone through sync point. The segments are collected in the current SDEP segment buffer. One buffer allocated for each area defined with the SDEP segment type exists. This allocation takes place at area open time.

Normal buffer allocation for BMPs in a DBCTL environment

BMP regions accessing Fast Path resources require that the number of buffers to be allocated as normal buffers be specified in the region startup procedure by using the NBA startup parameter.

Because buffers allocated as normal buffers are used first, the number of normal buffers allocated must accommodate most of the transaction requirements.

The number of buffers a transaction or a sync interval is allowed to use must be specified for each region if Fast Path resources are likely to be accessed.

The combined value of the NBA and OBA parameters define the maximum number of buffers that IMS can allocate to the region.

If you use the Fast Path 64-bit buffer manager, the number of normal buffers specified by the NBA parameter are page fixed in the Fast Path buffer pool when the buffers are allocated. If you do not use the Fast Path 64-bit buffer manager, the number of normal buffers specified by the NBA parameter are page fixed in the Fast Path buffer pool at the start of the region.

Normal buffer allocation for CCTL regions and threads

You must specify the normal buffer allocation for both CCTL regions and CCTL threads when CCTL (coordinator control) regions require fast path resources.

To specify the normal buffer allocation, use the following parameters specified in the database resource adapter (DRA) startup table:

CNBA
Specifies the normal buffer allocation of each active CCTL region

FPB
Specifies the normal buffer allocation for CCTL threads

When the CCTL connects to DBCTL, the number of CNBA buffers is page fixed in the fast path buffer pool. However, if CNBA buffers are not available, the connect fails.

Each CCTL thread that requires DEDB buffers is assigned its fast path buffers (FPB) out of the total number of CNBA buffers.

For more information about the CCTLNBA parameter, refer to *IMS Version 15 System Administration*.

Overflow buffer allocation for BMPs

Overflow buffer allocation for BMPs is optional and is used for exceptional buffer requirements when the normal buffer allocation has been exhausted.

If you use the Fast Path 64-bit buffer manager, access to the overflow buffers is multi-threaded and multiple BMP regions and CCTL threads can use overflow buffers at the same time.

If you do not use the Fast Path 64-bit buffer manager, the access to the overflow buffers by a BMP region or CCTL thread is dependent on obtaining a latch that serializes all BMPs and CCTL threads currently in an overflow buffer state. If the latch is not available, the region has to wait until it is available. After the latch has

been obtained, the NBA value is increased by the OBA value and normal processing resumes. The overflow buffer latch is released during sync point processing. At any point in time, only the largest OBA request among all the active BMPs and CCTL threads is page fixed in the Fast Path buffer pool.

The combined value of the NBA and OBA parameters specified for a dependent region define the maximum number of buffers that Fast Path can allocate to the region.

Overflow buffer allocation for CCTL threads

OBA for CCTL threads is similar to that for BMPs. The OBA value used for each thread is set with the FPOB parameter in the startup table.

This buffer allocation is optional and is used for exceptional buffer requirements when the FPB has been exhausted. Its use is dependent on obtaining a latch that serializes all BMPs and CCTL threads currently in an overflow buffer state. If the latch is not obtained, the FPB value is increased by the FPOB value, and normal processing resumes. The overflow buffer latch is released during sync point processing. At any point in time, only the largest OBA/FPOB request among all the active BMPs and CCTL threads is page fixed in the fast path buffer pool.

Fast Path buffer allocation algorithm for BMPs

FPBs are allocated on demand up to a limit specified at the start of the region. Buffers specified as NBAs are used by one sync point interval.

Before satisfying any request from the NBA allocation, an attempt is made to reuse any already allocated buffer containing an SDEP CI. This process goes on until the NBA limit is reached. From that point on, a warning in the form of an 'FW' status code returned to Fast Path database calls is sent to BMP regions.

The next request for an additional buffer causes the buffer stealing facility to be invoked and then the algorithm examines each buffer and CI already allocated. As a result, buffers containing CIs being released are sent to a local queue (SDEP buffer chain) to be reused by this sync interval.

If, after invoking the buffer stealing facility, no available buffer is found, a request for the overflow buffer latch is issued. The overflow buffer latch governs the use of an additional buffer allocation, OBA. This allocation is also specified as a parameter at region start time. From that point on, any time a request cannot be satisfied locally, a buffer is acquired from the OBA allocation until the OBA limit is reached. At that time, BMP regions have their 'FW' status code replaced by an 'FR' status code after an internal ROLB call is performed.

Fast Path buffer allocation algorithm for CCTL threads

When a CCTL thread issues a schedule request using FPB, buffers are allocated out of the CNBA total.

Unless CNBA=0 is specified or you are using the Fast Path 64-bit buffer manager, if FPB cannot be satisfied out of CNBA, the schedule request fails. When either CNBA=0 or the Fast Path 64-bit buffer manager is used, if FPB cannot be satisfied out of CNBA, IMS acquires additional buffers to allow the thread to schedule.

Before satisfying any request from the FPB allocation, an attempt is made to reuse any already allocated buffer containing an SDEP CI. This process goes on until the

FPB limit is reached. From that point on, a warning in the form of an 'FW' status code returned to Fast Path database calls is sent to the CCTL threads.

The next request for an additional buffer causes the buffer stealing facility to be invoked, and then the algorithm examines each buffer and CI already allocated. As a result, buffers containing CIs being released are sent to a local queue (SDEP buffer chain) to be reused by this sync interval.

If, after invoking the buffer stealing facility, no available buffer is found, a request for the overflow buffer latch is issued. The overflow buffer latch governs the use of an additional buffer allocation, OBA (FPOB). From that point on, any time a request cannot be satisfied locally, a buffer is acquired from the FPOB allocation until the FPOB limit is reached. At that time, CCTL threads have their 'FW' status code replaced by an 'FR' status code after an internal ROLB call is performed.

Fast Path buffer allocation in DBCTL environments

IMS allocates Fast Path buffers because DEDB writes are deferred until after sync point processing.

The result of one sync interval is written back by one output thread. These output threads run from the control region in SRB mode. Buffers allocated to an output thread are therefore not available to BMPs and CCTL threads until after the CI they contain is written back.

If you use the Fast Path 64-bit buffer manager, IMS manages the allocation of Fast Path buffers for you. The allocation of buffers is page fixed when the buffers are allocated.

If you do not use the Fast Path 64-bit buffer manager, you specify the number of buffers that IMS allocates by using the DBFX parameter. If the Fast Path buffer pool is defined exactly as the sum of all NBAs, BMPs and CCTL threads must wait for the buffers to come back to the global pool. BMPs and CCTL threads can process the next transaction as soon as the sync point completes. Sync point processing does not wait for the output thread to complete. The allocation of buffers is page fixed at the start of the first region specifying an NBA or FPB request.

Determining the size of the Fast Path buffer pool for DBCTL

If you are not using the Fast Path 64-bit buffer manager, you can calculate the number of buffers a Fast Path buffer pool must contain in a DBCTL environment by using the formula DBBF ≥ A + N + LO + DBFX + CN.

The terms in the formula above are described as follows:

DBBF Fast Path buffer pool size as specified

A Number of active areas that have SDEP segments

N Total of all NBAs

LO Largest overflow buffer allocation among active BMPs and CCTL threads

DBFX System buffer allocation

CN Total of all CNBAs

Fast Path buffer performance considerations for DBCTL

The performance considerations for Fast Path buffers differ depending on whether you are using the Fast Path 64-bit buffer manager.

If you are using the Fast Path 64-bit buffer manager to create and manage your Fast Path buffer pools, IMS optimizes many aspects of the buffer pool performance for you.

If you are not using the Fast Path 64-bit buffer manager, you might need to modify the buffer pool specifications yourself to maintain optimum performance of the buffer pools.

The following considerations apply to both buffers managed by the Fast Path 64-bit buffer manager and Fast Path buffers that are not managed by the Fast Path 64-bit buffer manager.

An NBA/FPB value that is too large can increase the probability of contention (and delays) for other BMPs and CCTL threads. All CIs can be acquired at the exclusive level and be kept at that level until the buffer stealing facility is invoked. This happens after the NBA limit is reached. Therefore, an NBA/FPB that is too large can increase resource contention. Also, an FPB value that is too large indicates that fewer CCTL threads can concurrently schedule fast path PSBs.

A (NBA + OBA) value that is too small might result in more frequent unsuccessful processing. This means an 'FR' status code condition for BMP regions and CCTL threads.

Inquiry-only BMP or CCTL programs do not make use of the overflow buffer specification logic, as buffers already allocated are reused when the NBA/FPB limit is reached.

IMS logs information about buffers and their use to the X'5937' log. This information can be helpful in determining how efficiently the Fast Path buffers are being used.

The following considerations apply only when Fast Path buffers are not managed by the Fast Path 64-bit buffer manager.

An incorrect specification of DBBF (too small) can result in the rejection of an area open or a region initialization. The system calculates the size of the buffer pool and rejects the open or initialization if the actual DBBF value is smaller.

A DBFX value that is too small is likely to cause region waits and increase response time.

An NBA/FPB value that is too small might cause the region processing to be serialized through the overflow buffer latch and again cause delays.

Related tasks:

"Determining the size of the Fast Path buffer pool for DBCTL" on page 486

The NBA/FPB limit and sync point in a DBCTL environment

In BMP regions and CCTL threads, when the NBA/FPB limit is reached, an 'FW' status code is returned. This status code is presented to every subsequent Fast Path database call until the OBA/FPOB limit condition is reached.

The first occurrence of the 'FW' status code indicates no more NBA/FPB buffers exist. This occurrence is a convenient point at which to request a sync point. Fast Path resources (and others) would be released and the next sync point interval would be authorized to use a new set of NBA/FPB buffers. The overflow buffer latch serializes all the regions in an overflow buffer state and therefore causes delays in their processing.

Related concepts:

"Processing option P (PROCOPT=P)" on page 463

Low activity and the DBFX value in a DBCTL environment

If the IMS or Fast Path activity in the system is relatively low, log buffers are written less often and therefore output threads are scheduled or dispatched less frequently. This situation is likely to result in many buffers waiting to be written and therefore could cause wait-for-buffer conditions.

To alleviate or avoid wait-for-buffer conditions you can enable the Fast Path 64-bit buffer manager, which manages Fast Path buffers for you dynamically. When the Fast Path 64-bit buffer manager is enabled, IMS tracks buffer usage, adds or removes buffers as needed, and ignores the DBBF, DBFX, and BSIZ parameters, if they are specified.

If you are not using the Fast Path 64-bit buffer manager, specify a larger DBFX value.

When the Fast Path 64-bit buffer manager is not used, a special case to be considered is the BMP region loading or processing a DEDB and being the only activity in the system. For example, assume that an NBA of 20 buffers exists. To avoid a wait-for-buffer condition, the DBFX value must be between once or twice the NBA value. This can result in a DBBF specification of three times the NBA number, giving 60 buffers to the Fast Path buffer pool.

Except for the following case, there is no buffer look-aside capability across BMP regions and CCTL threads or sync intervals (global buffer look-aside).

Assume that a region requests a DEDB CI resource that is currently being written or is owned by another region that ends up being written (output thread processing). Then, this CI and the buffer are passed to the requestor after the successful completion of the write (no read required). Any other BMP regions and CCTL threads must read it from disk.

Fast Path buffer allocation in IMS regions

IMS regions that access Fast Path resources must have the NBA and OBA parameters specified in their startup procedures.

With MODE=MULT, these allocations must be large enough to accommodate all buffer requirements for transactions processed between sync points.

With MODE=SNGL, transaction classes should be set up so transactions with similar buffer requirements are run in the same region.

Chapter 24. Implementing database design

After you design your databases and application programs you must describe their characteristics to IMS before you can use them.

In an IMS system in which the IMS management of ACBs is enabled, you can describe the characteristics of your databases and application programs to IMS by using either SQL Data Definition Language (DDL) statements or by coding IMS utility macro instructions. If the IMS management of ACBs is not enabled, you can use only the IMS utility macro instructions.

In IMS, the description of a database is referred to as a *database descriptor* (DBD). A DBD describes the physical and logical characteristics for each database.

The description of the characteristics of an application program is called a *program specification block* (PSB). A PSB describes the characteristics of an application program and how the program uses data and terminals. Because a PSB also defines what an application program can see of a database, a PSB is also referred to as a *program view*.

When the IMS management of ACBs is enabled, DBDs and PSBs are stored in the IMS catalog only. When the IMS management of ACBs is not enabled, DBDs and PSBs are stored in DBD and PSB libraries and, optionally, in the IMS catalog.

For use during run time, the information from a set of related DBDs and PSBs is combined and converted into *application control blocks* (ACBs).

In IMS systems that manage ACBs, IMS converts the DBD and PSB information into ACBs for you. In IMS systems that use ACB libraries, you can either build the ACBs by using the ACB Maintenance utility or, for some batch application programs, IMS can build the ACBs dynamically during run time.

Related concepts:

"Who attends code inspection 1" on page 32

Chapter 22, "Designing full-function databases," on page 431

Chapter 23, "Designing Fast Path databases," on page 459

Using DDL to define databases and program views

In IMS systems that are enabled to manage application control blocks (ACBs), you can define databases and program views by using standard SQL data definition language (DDL) statements.

Using DDL statements, you can describe the physical and logical characteristics of databases, as well as the characteristics of the programs that use the databases and how the programs use data and terminals.

To simplify the definition of IMS databases with DDL, IMS provides defaults for many IMS-specific settings. The default values conform to common best practices in IMS database design.

When you submit DDL statements, IMS creates the ACBs automatically and stores the database (DBD) and program view (PSB) definitions in the IMS catalog. You do not use generation utilities.

Depending on the DDL specifications and the specifications in the IMS system, IMS can automatically activate the database and program view definitions in the IMS system. When DDL changes are not activated automatically, the definitions are stored in the staging data set of the IMS catalog and you activate them by issuing the IMPORT DEFN SOURCE(CATALOG) command. You do not use the online change process to activate the changes.

In IMS systems that support DDL, the IMS catalog is the single, trusted repository for all of your database definitions and the IMS catalog determines which definitions are active in the IMS system. The online IMS system does not use PSB, DBD, or ACB libraries. If you have tools or procedures that require these libraries, you can create them from the definitions in the IMS catalog by using the IMS Catalog Library Builder utility (DFS3LU00).

Related concepts:

➡ IMS management of ACBs (System Definition)

Chapter 5, "Overview of the IMS catalog," on page 43

Related reference:

➡ IMPORT DEFN SOURCE(CATALOG) command (Commands)

➡ IMS Catalog Library Builder utility (DFS3LU00) (System Utilities)

DDL for IMS

The IMS implementation of the SQL data definition language (DDL) generally conforms to the well-defined, standard DDL syntax that is used with relational databases throughout the industry, so it makes creating and modifying IMS databases much easier for application developers and database administrators that are not familiar with IMS.

Because DDL was developed for relational databases, some of the standard DDL keywords and parameters are different than the native IMS terminology that is used to describe the physical and logical characteristics of a hierarchical database.

IMS preserves the relational terminology of the standard DDL syntax and, like most other database products, extends the syntax with IMS-specific terms when the existing, standard terminology falls short.

For example, the relational term TABLE is preserved in the IMS implementation of DDL and defines the IMS segment structure. The relation term COLUMN defines an IMS field structure.

As an example of how the IMS extends the standard DDL syntax, the IMS implementation of DDL includes a SENSEGVIEW keyword in the statements that define a program view. SENSEGVIEW corresponds to the IMS SENSEG macro instruction statement and defines the segments an application program can see.

For an explanation of how relational terms map to the hierarchical structures that are used in IMS, see Comparison of hierarchical and relational databases (Application Programming).

Related concepts:

➡ IMS management of ACBs (System Definition)

Chapter 5, "Overview of the IMS catalog," on page 43

Automation options for DDL CREATE statements

Automation options for DDL CREATE statements prepare and bring the DDL CREATE definition for databases and program resources online without users having to manually perform the process.

The **AUTOCREATE** and **AUTOIMPORT(CREATE)** parameters are added for users to perform the following operations automatically:

- Activating application control blocks (ACBs)
- Creating runtime attribute blocks
- For DEDB databases, creating new data sets

When Dynamic Resource Definition (DRD) is enabled (`MODBLKS=DYN`), the **AUTOCREATE** and **AUTOIMPORT(CREATE)** parameters provide DRD support to dynamically create database and program resource runtime attributes if they do not exist for these new resources without having to issue the CREATE DB and CREATE PGM commands.

You can specify AUTOCREATE or AUTOIMPORT(CREATE) in the `<SECTION=CATALOG>` section of the DFSDFxxx PROCLIB data set.

Both the **AUTOCREATE** and **AUTOIMPORT(CREATE)** parameters do not support XRF.

For both functions, if **ACBSHR=Y** is specified, one DDL CREATE job can be run in an IMSplex at one time. If more than one DDL CREATE jobs are to be run concurrently in an IMSplex, you must submit them to one IMS system that serves as the master IMS to coordinate and serialize the DDL CREATE processes.

The DDL AUTOIMPORT(CREATE) parameter

The **DDL AUTOIMPORT(CREATE)** parameter supports the following databases:

- DEDB databases, including DEDB single area data sets (SADS)
- Full function databases
- HALDB databases

Note: For HALDB databases, this function creates an HALDB master database. After you finish preparing the HALDB partitions, issue the /STA DB command to complete the HALDB database partitions.

The **AUTOIMPORT(CREATE)** parameter does not support Fast Path Secondary Index databases.

The DDL AUTOCREATE= parameter

Use the DDL **AUTOCREATE=** parameter to automatically create data sets for DEDB databases. This parameter supports only DEDB databases and can be used in either **ACBSHR=Y** or **ACBSHR=N** environments.

Before you submit DDL that creates DEDB area data sets, specify the attributes for those data sets in the DDL section of the DFSDFxxx PROCLIB member. IMS uses the DDL section of the DFSDFxxx member of the PROCLIB data set to customize

the data set information without an IMS shutdown. The DDL section supports DEDB only. For more information, see DDL section of the DFSDFxxx member (System Definition).

After the DEDB area data sets are created, the IMS system issues the /START DB and /START AREA commands. If **ACBSHR=Y**, the IMS system issues the /START DB GLOBAL and /START AREA GLOBAL commands. The master IMS creates all the data sets and coordinates the changes across all of the other sharing IMS systems.

Preventing unauthorized DDL

DDL is tied into your SAF authorization. A user will not be able to submit DDL if they do not have access to the catalog update PSB (DFSCP001).

Submitting DDL through the IMS Explorer for Development

IMS Explorer for Development enables you to generate DDL from an existing DBD. You can generate DDL by right-clicking on a DBD or Segment and selecting **Generate DDL**. You can edit the generated DDL and submit the new statements to IMS.

For more information, see Generating DDL with IMS 14 and Submitting DBD field changes to IMS with IMS 14.

Activating DDL-defined resources with the IMPORT DEFN command

In many cases, DDL-defined resources are not activated automatically, but instead stored in a pending state in a staging data set of the IMS catalog. To activate pending DDL-defined resources use the IMPORT DEFN SOURCE(CATALOG) command.

In an IMSplex, if you are changing only PSBs, you can use the IMPORT DEFN SOURCE(CATALOG) command with the UPDATEPSB and REFRESHPSB options to activate the PSB change in a subset of systems in the IMSplex before activating the change in the IMSplex as a whole.

Related concepts:

➡ Online changes in managed-ACB environments (System Administration)

Related tasks:

➡ Activating PSBs in subsets of systems in an IMSplex (System Administration)

Related reference:

➡ IMPORT DEFN SOURCE(CATALOG) command (Commands)

Submitting DDL through the SQL Batch utility

You can use the SQL Batch utility in IMS to submit DDL and other specific SQL statements to IMS. The IBM Java for z/OS (JZOS) Batch Launcher is used to invoke the statements from a standardized z/OS job.

Prerequisites: Before you can run the SQL Batch utility, the following prerequisites must be met:
- The IBM Java for z/OS (JZOS) Batch Launcher is installed.
- The IMS Open Database solution is installed.
- The IMS catalog is installed and configured.

To submit SQL statements with the SQL batch utility, complete the following steps:

1. Create the SQL statements to be run. You can specify all of the SQL statements and syntax that the IMS Universal JDBC driver supports and the CONNECT, COMMIT, ROLLBACK, and DISCONNECT statements. Each SQL statement must be delimited with a semicolon.

   The SQL statements can be submitted inline with the JCL, from a data set on z/OS, or from a file on UNIX System Services (USS).

2. Set up the JCL to run the SQL Batch utility.

3. Submit the JCL.

Related reference:

➡ SQL Batch utility (Database Utilities)

➡ Examples for the SQL Batch utility (Database Utilities)

Related information:

➡ **IBM Java for z/OS (JZOS) Batch Launcher and Toolkit**

Defining databases with DDL

To create or modify the definition (DBD) of the physical and logical structures of an IMS database by using DDL, you code DDL statements and submit them to IMS by using the SQL Batch utility or a product like the IMS Enterprise Suite Explorer for Development.

Related reference:

➡ SQL statements (Application Programming APIs)

High-level procedure for defining databases with DDL

IMS supports the standard CREATE, ALTER, and DROP DDL statements to create, modify, and delete IMS databases and program views.

The database (DBD) and program view (PSB) attributes and characteristics that you can specify on the DDL CREATE and ALTER statements are generally the same as the attributes and characteristics that you can specify on the macro instruction statements for the DBD and PSB generation utilities; however, the names of some parameters might be different, either to conform to the standard DDL terminology or for enhanced clarity.

To implement changes to an existing database, you must use the ALTER DATABASE statement. You can change existing tables, table spaces, or columns using the ALTER TABLE and ALTER TABLESPACE statements. You can also add or remove these objects from the database description using CREATE and DROP statements.

To make changes to an existing table in a database, use the ALTER TABLE statement.

To make changes to the columns, logical child (LCHILD), and map descriptions in the table, also use the ALTER TABLE statement.

Some things are general to all database types, for example the change data capture exit. Other things are specific to a DB type, for example the partition selection exit for HALDBs and randomizers for DEDBs, HDAMs, and PHDAMs.

Recommendation: Use the IMS Enterprise Suite Explorer for Development to create or modify database and program view definitions and submit them to IMS. Using IMS Explorer for Development, you can create new database and program view definitions from the existing databases and program views in your IMS system and then modify the definitions as necessary.

1. Code your DDL statements, defining the database, tables, and tablespaces.
   - If you are creating a new database, use CREATE statements.
   - If you are altering an existing database, you can use a combination of ALTER, CREATE, and DROP statements.
   - Include a final COMMIT statement.

2. If you are creating a new database, make sure that the following steps are complete, planned for, or are handled automatically by IMS:
   - The creation of the database data sets.
   - The creation of a PSB to access the new database.
   - The creation of the runtime attribute control blocks (DDIRs and PDIRs) in the online IMS systems for the database and application programs.

3. Submit the DDL statements to IMS by using either the IMS Enterprise Suite Explorer for Development or the IMS SQL Batch utility. Upon receiving the DDL with a COMMIT statement, IMS takes the following action:
   - Validates the DDL syntax and, if you are creating a new database, confirms that the database does not already exist.
   - Updates the DBD record in the IMS catalog
   - Generates the required application control blocks (ACBs) and, if the resources are not activated automatically, stores them in the staging data set of the IMS directory.
   - If you are creating a new database, executes any automation options that might be specified by the following parameters in the CATALOG section of the IMS DFSDF*xxx* PROCLIB member:

     AUTOIMPORT(CREATE)
     When specified, IMS takes the following actions automatically:
     – Loads the ACBs into the IMS directory.
     – Activates the ACBs in the online system.
     – Creates the runtime attribute blocks for the databases (DDIRs) and, if CREPGM=Y is specified, the program views (PDIRs) in the online systems.

     AUTOCREATE=YES
     For Fast Path DEDB databases, in addition to activating the ACBs and the runtime attribute control blocks, IMS automatically creates the database data sets.
   - Updates the resource list indicating that this database was updated.

4. If you are altering an existing database, you might need to reorganize the database.

5. If the database and program view definitions still needs to be activated in online IMS systems, issue the IMPORT DEFN SOURCE(CATALOG) command to activate the database and program definitions.

Related reference:

⇨ ALTER DATABASE (Application Programming APIs)

⇨ ALTER TABLE (Application Programming APIs)

➡ ALTER TABLESPACE (Application Programming APIs)

Creating IMS databases

To create a database, you must start with a CREATE DATABASE statement followed by the CREATE TABLESPACE and CREATE TABLE statements as they become applicable.

The default database description generated by a CREATE DATABASE statement is a database with an IMS PHIDAM OSAM access type. If you want a different database access type, then you can override it by specifying the desired type on the statement or use the LIKE keyword to generate a database description using attributes of an existing resource.

The following tables display the most commonly used DBD/PSB source and the equivalent DDL. For more information about type-1 and type-2 commands, see Equivalent IMS type-1 and type-2 commands (Commands).

Table 69. DDL equivalents for the DEDB DBD

Task	With DBD source	Similar DDL statement

Define a DEDB database	`DEDB1 DBD NAME=DEDB0001,`
	`ACCESS=DEDB,RMNAME=RMOD1`
		`CREATE DATABASE`
		`DEDB0001 ACCESS`
		`DEDB RMNAME(RMOD1)`

Table 70. DDL equivalents for the HDAM DBD

Task	With DBD source	Similar DDL statement

Define a HDAM database	`DBD NAME=SKILLINV,`
	`ACCESS=HDAM,RMNAME=`
	`(DFSHDC40,20,500,824)`
		`CREATE DATABASE`
		` SKILLINV ACCESS`
		`HDAM VSAM RMNAME`
		`(DFSHDC40 RMANCH 20`
		`RMRBN 500 RMBYTES 824)`

Table 71. DDL equivalents for creating a root segment

Task	With DBD source	Similar DDL statement

Define a root segment in a database	`SEGM NAME=A,BYTES=(48,27),`
	`PARENT=0`
		`CREATE TABLE`
		`A INTERNALNAME A`
		`MAXBYTES 48 MINBYTES 27`
		`IN DATABASE SKILLINV`

Multiple DDL statements are needed in order to describe the make-up of a database, such as:

* databases organization and access method
* table spaces
* tables and columns mappings in a database record
* relationships with other databases or tables
* primary keys and foreign keys
* programview (PSB) describing a programs characteristics and its use of tables, columns, and logical terminals

After you design your database and code the necessary DDL statements, you submit them to IMS through a Java IMS application. IMS continues to accept DDL statements to generate a DBD record until you also submit a COMMIT DDL

statement, which ends the unit of work. IMS is then triggered to write the DBD record out to the IMS catalog and prepares for activation and use during database processing.

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

➡ Equivalent IMS type-1 and type-2 commands (Commands)

Creating databases best practices:

Although IMS DDL conforms with standard DDL syntax, it also extends the syntax with IMS specific keywords. You may choose to utilize these IMS keywords or allow IMS to take the default values for them as described in the IMS DDL syntax descriptions.

Ensure that you set-up the required infrastructure and connections to communicate with IMS via ODBM. You can use installation verification program (IVP) to do this for you.

1. Use IMS Enterprise Suite Explorer for Development to generate the DDL statements for program views which reference the database. At least one of the program views should be defined with PROCOPT L. For more information, see Generating DDL with IMS 14.

2. Submit DDL statements, including a final COMMIT DDL statement. DDL changes are held in IMS in a pending state. For more information, see Submitting DBD field changes to IMS with IMS 14.

3. Allocate your database data sets. For more information, see "Allocating the indexed database data sets" on page 866.

4. Optional: Define DBRC definitions.

5. Optional: For non-HALDB full-function DBs, define IMS database for dynamic allocation (MDA).

6. Initialize the partitions. For example,
   - For Fast Path DBs, run DBFUMIN0, or
   - For full-function DBs, run DFSUPNT0 and/or Prereorg (DFSURPR0) utilities.

   For more information, see "Initializing the partitions" on page 855.

7. Load IMS databases as needed. For example, HALDB and Fast Path might not need an initial load. However, full-function DBs require it. For more information, see "Loading each database as a HALDB database" on page 875, "Loading Fast Path databases" on page 584 and "Loading a DEDB" on page 585.

8. Establish database recovery point by taking an image copy (IC). For more information, see Image copy utilities (DFSUDMP0, DFSUDMT0, DFSUICP0) (System Administration).

9. Define database and program online resource definitions:
   - If DRD is enabled, issue the CREATE DB / PGM command to create the database and program (DRD).
   - If DRD is not enabled, specify DATABASE and PROGRAM system definition macros.

   At this point, all of the above steps need to be completed, regardless of the order in which you choose to do them. For more information, see Modifying system resources online (Operations and Automation).

10. Issue the IMPORT DEFN SOURCE(CATALOG) command to activate the database and program definitions. For more information, see IMPORT DEFN SOURCE(CATALOG) command (Commands).

11. Start **IMS resources - Databases, Programs, Transactions, Routing Codes**.

Related concepts:

➮ IMS installation verification program (IVP) overview (Installation)

Related reference:

➮ CREATE DATABASE (Application Programming APIs)

Creating full-function databases:

When defining a new database, use the CREATE DATABASE statement. The default database description generated is with an IMS PHIDAM OSAM access type, however this can be overridden with the ACCESS keyword. Depending on the type of database you are creating, you may be required to also define at least one table space and at least one table. This can be accomplished by using a CREATE TABLESPACE statement and CREATE TABLE statement.

The following rules apply depending on the database access type you are defining.

For HSAM and SHSAM:
- At least one table with a maximum of 255 tables for the database.
- 0-255 columns for each table, and a maximum of 10,000 for the database.
- You cannot specify use of hierarchic, physical child, or physical twin pointers between tables in the database.
- You cannot specify use of logical or index relationships between tables.
- Optionally, you can define a simple HSAM (SHSAM) database that can contain only one fixed-length table. In this case, no prefixes are built in occurrences of the table.

For HISAM and SHISAM:
- At least one table with a maximum of 255 tables for the database.
- 0-255 columns for each table, and a maximum of 10,000 for the database, one of which must be a primary key in the root table.
- You cannot specify the use of hierarchic or physical child or physical twin pointers between segments in a HISAM database.
- Optionally, you can define a simple HISAM (SHISAM) database that can contain only one fixed-length table. In this case, no prefixes are built in occurrences of the table. The logical record length specified for a SHISAM database must be equal to or greater than the table length specified.
- Optional edit proc identifying an edit or compression exit routine to enable user-supplied routines to manipulate each occurrence of a table to or from auxiliary storage.
- Optional logical relationships using symbolic pointer options when a table in a HISAM database points to another table in a HISAM database, and direct or symbolic pointer options when a table in a HISAM database points to a table in an HDAM or HIDAM database.
- Optional maximum of 32 secondary index relationships per table and a maximum of 1000 for the database.
- Optional Data Capture exit routine to enable DB2 for z/OS users access to updated IMS data. This exit routine can be used in SHISAM also.

For HDAM and HIDAM:
- At least one table with a maximum of 255 tables for the database.
- Use of hierarchic, physical child, or physical twin pointers between tables in the database.
- 0-255 columns for each table, and a maximum of 10,000 for the database.
- Optional edit proc identifying an edit or compression exit routine to enable user-supplied routines to manipulate each occurrence of a table on their way to or from auxiliary storage.
- Optional logical relationships between tables using direct address or symbolic pointer options.
- Optional maximum of 32 secondary index relationships per table and a maximum of 1000 for the database.
- Optional Data Capture exit routine, to enable DB2 for z/OS users access to updated IMS data.
- Optional data set group definitions based on the size of the database partitions and usage of the tables by applications.

For INDEX and PSINDEX, the primary HIDAM index is composed of one index table that indexes occurrences of the HIDAM root table. PHIDAM does not have a DBD for the primary index. An index table contains:
- The sequence field key of the root segment occurrence it indexes.
- In its prefix, a direct address pointer to the root segment occurrence.

A primary HIDAM index must specify:
- One table.
- The index relationship required between the primary HIDAM index database and the root table of a HIDAM database.
- One column within the table as the primary key.
- You cannot specify any additional columns as you might for a secondary index.

Secondary index DBD defines a secondary index database made up of 1 to 16 index pointer tables. These are used to index target tables in HISAM, SHISAM, HDAM, PHDAM, HIDAM, or PHIDAM databases.

A full-function secondary index must specify:
- One table.
- One column for each table.

A logical DBD generation creates a logical database made up of logical tables. A logical table is defined in a logical database that represents a table or the concatenation of two tables defined in a physical database or databases.
- From 1 to 255 tables. Each defines the name of a logical table, and the name of the table or tables in physical databases that are to be processed when a call is issued to process the logical table.
- The logical relationships used to create a logical database must be defined in a physical database or databases.
- All columns required for tables in a logical database must have been defined in physical databases.

Creating HALDB databases:

When defining a new HALDB database, use the CREATE DATABASE statement.

The default database description generated is with an IMS PHIDAM OSAM access type. To override this default setting, you can use the ACCESS keyword to indicate either PHIDAM or PHDAM. The OS access value may also be changed from OSAM to VSAM. Alternately, you can use the LIKE keyword to generate a database description using attributes of an existing resource.

The ddname is not defined in DDL. However, data set groups are defined through DDL. They are dependent on the DSGROUP=A-J setting on each table that is defined. Later when the partitions are defined to DBRC, it will pull the definition from the catalog to determine how many data set groups were defined for the database.

The DDL statements used for a new HALDB is purely for defining the hierarchical structure and relationships of the data. DDL does not define each individual partition.

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

Defining partitions:

You can define partitions using DBRC, not DDL. However, the database must be defined with DDL first, then it can be created with DBRC.

Ensure that you have already defined the database using DDL. Ensure that you have update authority for the RECON data sets.

1. Register the HALDB master database with DBRC by using either DBRC batch command INIT.DB.

2. Define the partitions to DBRC using the DBRC batch command INIT.PART.

Related concepts:

"Partitions in PHDAM, PHIDAM, and PSINDEX databases" on page 182

Related reference:

➡ INIT.DB command (Commands)

➡ INIT.PART command (Commands)

Creating Fast Path databases:

Creating a Fast Path database applies to DEDBs only. MSDBs are not supported by the IMS DDL interface. When defining a new DEDB database, use the CREATE DATABASE statement while also specifying the ACCESS DEDB clause. You can also use the LIKE keyword to generate a database description using attributes of an existing DEDB resource.

When defining a DEDB database you may specify an optional optional Data Capture Exit routine to enable DB2 for z/OS users access to updated IMS data.

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

Creating GSAM databases:

When defining a new GSAM database, use the CREATE DATABASE statement while also specifying the ACCESS GSAM clause.

The default OS access type is VSAM, however you can specify VSAM or BSAM. You can also use the LIKE keyword to generate a database description using attributes of an existing GSAM resource.

When defining a GSAM database, consider the following:
- You can define only one table space (data set group)
- The ddname of an input data set, which is the table space name provided on the CREATE TABLESPACE statement, is used by the application to retrieve data from the database.
- The ddname of an output data set, which is the name provided on the OUTPUT keyword on the CREATE TABLESPACE statement, is used when loading the database.

Related concepts:

"GSAM databases" on page 142

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

Dropping databases:

You can drop an IMS database by issuing the DROP DATABASE statement. When you drop a database, all of its table spaces and tables are dropped too. Issuing DROP will remove all instances of the database from the IMS catalog and render it inactive to IMS.

You will need to perform the following tasks:
- Removal of DRD resources.
- Removal of program views that reference the database being dropped.

Related reference:

➡ DROP DATABASE (Application Programming APIs)

Creating IMS table spaces

A table space stores one or more IMS segment types, which are referred to as tables in DDL statements for IMS. A table space name is an identifier of up to eight characters. You can associate it to a specific database with the IN keyword. Tables you want stored in a table space can be associated with the table space also with the IN keyword on either the CREATE TABLE or ALTER TABLE statement.

Design the type of database you want to create before you implement one or more table spaces.

Creating table spaces for full-function databases:

When defining a new table space (data set group), use the CREATE TABLESPACE statement. Some full-function databases require that you define table spaces, some full-function databases do not require it. For example, this is true for the DBs that are defined using the Gen utilities.

The following rules apply depending on the database access type you are defining.

For HSAM and SHSAM:
- You can define only one table space (data set group).
- The ddname of an input data set, which is the table space name provided on the CREATE TABLESPACE statement, is used by the application to retrieve data from the database.
- The ddname of an output data set, which is the name provided on the OUTPUT keyword on the CREATE TABLESPACE statement, is used when loading the database.

For HISAM and SHISAM:
- You can define only one table space (data set group).
- The ddname of one VSAM key sequenced data set (KSDS) and one VSAM entry sequenced data set (ESDS).

For HDAM and HIDAM:
- One to 10 data set groups.
- You can define how free space is to be distributed in each data set group.
- The ddname of an OSAM or ESDS data set, which is the name provided on the CREATE TABLESPACE statement, represents the data set group.

For index, a primary HIDAM index must specify:
- One table space (data set group). You must specify the ddname of one KSDS as the name on the CREATE TABLESPACE statement.

A full-function secondary index must specify:
- One data set group. If all index pointer table keys are unique, you must specify the ddname of one KSDS. If index pointer table keys are non-unique you must specify the ddnames of one KSDS and one ESDS.

For logical, there are no physical table spaces or data set groups for a logical database. The CREATE TABLESPACE statement is invalid when defining a logical database.

Creating table spaces for HALDB databases:

The DDL TABLESPACE statement is not needed for HALDB databases. A CREATE TABLESPACE, ALTER TABLESPACE, or DROP TABLESPACE statement cannot be specified for HALDB databases at this time. You may specify the DSGROUP keyword on a CREATE TABLE or ALTER TABLE statement to identify which table space or data set group that table belongs to.

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

➡ CREATE TABLESPACE (Application Programming APIs)

Creating table spaces for DEDB databases:

Table spaces are used to define Areas of a DEDB. This is done with the CREATE TABLESPACE statement.

When defining table spaces for a DEDB, consider the following:
- At least one table space with a maximum of 2048 for the database.

- The ddname or area name, which is the table space name provided on the CREATE TABLESPACE statement, describes an area.

Related reference:

➡ CREATE DATABASE (Application Programming APIs)

➡ CREATE TABLESPACE (Application Programming APIs)

Creating IMS tables

When you create an IMS table, you define a segment in the IMS database, its position in a database hierarchy, physical characteristics, and how it relates to other tables.

Consider these basic requirements when creating a new table:
- The table name is an identifier of up to 128 characters.
- The table name cannot be a SQL keyword or begin with DFS.
- Provide an INTERNALNAME for non-GSAM databases.
- Create program views that reference the INTERNALNAME on each SENSEGVIEW clause.

Restriction: If you specify any of the following keywords on the CREATE TABLE statement, you cannot change the keyword and keyword value by using the ALTER TABLE statement. To change the keyword and keyword value, you must first use the DROP TABLE statement to delete the table. Then, you must re-create the table by using the CREATE TABLE statement and specify the keyword and keyword value again.
- The INTERNALNAME*internalname* keyword to specify the internal name of the segment type being defined.
- DIRECT DEPENDENT | SEQUENTIAL DEPENDENT

Creating tables for HALDB databases:

When defining a new table for a HALDB database, use the CREATE TABLE statement.

Define the physical characteristics of the table and its relationship with other tables. Its hierarchy in the database is determined by the FOREIGN KEY clause.

Consider the following guidelines when creating a new table for a PHDAM or PHIDAM HALDB database:
- At least one table with a maximum of 255 tables for the database.
- Use of hierarchic, physical child, or physical twin pointers between tables in the database.
- 0-255 columns for each table, and a maximum of 10,000 for the database.
- Optional Editproc identifying an edit or compression exit routine to enable user-supplied routines to manipulate each occurrence of a table on their way to or from auxiliary storage.
- Optional logical relationships between tables using direct address or symbolic pointer options.
- Optional maximum of 32 secondary index relationships per table and a maximum of 1000 for the database.
- Optional Data Capture exit routine, to enable DB2 for z/OS users access to updated IMS data.

- Optional data set group definitions based on the size of the database partitions and usage of the tables by applications.

Related concepts:

"Multiple data set groups" on page 410

Related reference:

➜ CREATE DATABASE (Application Programming APIs)

➜ CREATE TABLE (Application Programming APIs)

Creating tables for DEDB databases:

When defining a new table for a DEDB, use the CREATE TABLE statement. You will define the physical characteristics of the table and its relationship with other tables. Its hierarchy in the database is determined by the FOREIGN KEY clause.

Consider the following when creating a new table for a DEDB:
- At least one table with a maximum of 127 tables for the database.
- 0-255 columns for each table, and a maximum of 1000 for the database, one of which must be a unique sequence field for the root segment type.
- Optional Editproc identifying an edit/compression exit routine to enable user-supplied routines to manipulate each occurrence of a table on their way to or from auxiliary storage.
- Optional maximum of 8 subset pointers for each child of the parent.
- Optional Data Capture exit routine, to enable DB2 for z/OS users access to updated IMS data.
- You cannot specify a logical relationship between tables.

Related reference:

➜ CREATE DATABASE (Application Programming APIs)

➜ CREATE TABLE (Application Programming APIs)

Creating tables for GSAM databases:

A single table can be defined to a GSAM database with the CREATE TABLE statement.

The table is optional and will be treated as a virtual table. The columns contained within the table will be non-searchable by IMS but are used by the universal drivers to map the contents of the record.

The only table option allowed on the CREATE TABLE statement are:
- IN DATABASE
- CCSID
- column definitions
- map definitions

The columns must be non-searchable and defined without an INTERNALNAME or PRIMARY KEY.

Related reference:

➜ CREATE TABLE (Application Programming APIs)

Creating secondary indexes

Secondary indexes are indexes that process a segment type in a sequence other than the one that is defined by the segment's key. A secondary index can also process a segment type based on a qualification in a dependent segment.

The following database types support secondary indexes:
- HISAM
- HDAM
- PHDAM
- HIDAM
- PHIDAM
- DEDB

Related concepts:

Chapter 12, "IMS catalog secondary index," on page 109

Related tasks:

"Altering IMS indexes" on page 510

"Dropping an index" on page 749

Related reference:

"XDFLD segment type format" on page 94

Creating logical relationships

Logical relationships resolve conflicts in the way application programs need to view segments in the database.

With logical relationships, application programs can access:
- Segment types in an order other than the one defined by the hierarchy
- A data structure that contains segments from more than one physical database.

An alternative to using logical relationships to resolve the different needs of applications is to create separate databases or carry duplicate data in a single database. However, in both cases this creates duplicate data. Avoid duplicate data because:
- Extra maintenance is required when duplicate data exists because both sets of data must be kept up to date. In addition, updates must be done simultaneously to maintain data consistency.
- Extra space is required on DASD to hold duplicate data.

By establishing a path between two segment types, logical relationships eliminate the need to store duplicate data.

To establish a logical relationship, three segment types are always defined:

  A physical parent
  A logical parent
  A logical child

The following database types support logical relationships:
- HISAM
- HDAM
- PHDAM
- HIDAM

• PHIDAM

Two databases, one for orders that a customer has placed and one for items that can be ordered, are called ORDER and ITEM. The ORDER database contains information about customers, orders, and delivery. The ITEM database contains information about inventory.

If an application program needs data from both databases, this can be done by defining a logical relationship between the two databases. As shown in the following figure, a path can be established between the ORDER and ITEM databases using a segment type, called a logical child segment, that points into the ITEM database. The following figure shows a simple implementation of a logical relationship. In this case, ORDER is the physical parent of ORDITEM. ORDITEM is the physical child of ORDER and the logical child of ITEM.

In a logical relationship, there is a logical parent segment type and it is the segment type pointed to by the logical child. In this example, ITEM is the logical parent of ORDITEM. ORDITEM establishes the path or connection between the two segment types. If an application program now enters the ORDER database, it can access data in the ITEM database by following the pointer in the logical child segment from the ORDER to the ITEM database.

Figure 230. A simple logical relationship

The physical parent and logical parent are the two segment types between which the path is established. The logical child is the segment type that establishes the path. The path established by the logical child is created using pointers.

Related concepts:

"Adding logical relationships" on page 514

Altering IMS databases
You can alter an IMS database by changing the description of a database in the IMS catalog. You do this by issuing the ALTER DATABASE statement.

Altering a full-function database:

You can issue the ALTER DATABASE statement to change the description of a full function database in the IMS catalog. This works with both HALDB and non-HALDB databases.

The supported changes include, but are not limited to, changing the encoding, randomizer name and attributes, data capture exit routines. Depending on the database access type, some attributes do not apply so changes for them are not allowed. For a list of supported keyword attributes based on database access type, see the ALTER DATABASE (Application Programming APIs) syntax diagrams.

There are two scenarios for altering a HALDB database:

1. If the HALDB has already been defined in the RECON then it should be altered in the RECON with DBRC batch CHANGE commands, or

2. If the HALDB was just created with DDL, but it has not been defined to the RECON. You can then alter a HALDB database with an alter statement. You would then be able to alter the HALDB partition selection exit name (PSNAME in the DBD).

Related reference:

➡️ ALTER DATABASE (Application Programming APIs)

Altering a Fast Path database:

You can alter an IMS Fast Path DEDB database by changing the description of a database in the IMS catalog. You do this by issuing the ALTER DATABASE statement.

The supported changes include, but are not limited to, changing the encoding, randomizer name and attributes, data capture exit routines.

Related reference:

➡️ ALTER DATABASE (Application Programming APIs)

Changing the data set access method:

You can change the data set access method from VSAM to OSAM or vice versa. IMS does not have the ability to issue a change of a database access type directly from DDL. If a different database access type is needed, create a new database using the CREATE DATABASE statement and specify the desired access type.

Altering IMS table spaces

To make changes to a table space, you can issue an ALTER TABLESPACE statement.

The supported changes include, but are not limited to, changing the output ddname, size changes, blocking factor values. Depending on the database access types, some attributes do not apply so changes for them are not allowed. For a list of supported keyword attributes based on database access type, see the ALTER TABLESPACE (Application Programming APIs) syntax diagrams.

The ALTER TABLESPACE statement cannot be specified for HALDB databases at this time. You may specify the DSGROUP keyword on a CREATE TABLE or ALTER TABLE statement to identify which table space or data set group a table belongs to.

Alternately, you can drop an existing table space and re-create it. To do so, follow these steps.

1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.

2. Issue a DROP TABLESPACE statement for the table space you plan to change.

3. Issue a CREATE TABLESPACE statement to recreate the table space with the desired attributes.
4. Issue a COMMIT DDL statement.

Related reference:

➡ ALTER TABLESPACE (Application Programming APIs)

Creating a table space:

You can define a new table space to an existing database by issuing a CREATE TABLESPACE statement. This can be useful if you find that you need to add another Fast Path DEDB Area or another data set group for a HDAM or HIDAM database.

1. Issue an ALTER DATABASE statement to identify the database in which you plan to include the new table space.
2. Issue a CREATE TABLESPACE statement. Ensure that you also provide the IN keyword to reference it back to the database it belongs to.
3. Issue any ALTER TABLE statements to move any tables you wish to the new table space (data set group). Ensure that you also provide the IN keyword to reference it to the desired table space.
4. Issue any CREATE TABLE statements for any new tables you wish to define to the new table space (data set group).
5. Issue a COMMIT DDL statement.

Related reference:

➡ CREATE TABLESPACE (Application Programming APIs)

Dropping a table space:

You can remove an existing table space by issuing a DROP TABLESPACE statement. This can be useful if you plan to change the table space name or plan to recreate the table space with a different set of attributes.

Ensure that if you are removing the a table space, you do not break the requirements of the database access types. Some database access types require at least one table space. Also, ensure that any tables that are assigned to the table space are not affected.

1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.
2. Issue a DROP TABLESPACE statement. Ensure that you also provide the IN keyword to reference it to the database it belongs to.
3. Issue any ALTER TABLE statements to move any tables you need to a different table space (data set group).
4. Issue a COMMIT DDL statement.

Dropping a table space to recreate it with modified attributes:
1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.
2. Issue a DROP TABLESPACE statement. Ensure that you also provide the IN keyword to reference it to the database it belongs to.
3. Issue a CREATE TABLESPACE statement giving it modified attributes. Ensure that you also provide the IN keyword to reference it back to the database it belongs to.

4. Issue a COMMIT DDL statement.

Dropping a table space to change the table space name:

1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.
2. Issue a DROP TABLESPACE statement. Ensure that you also provide the IN keyword to reference it to the database it belongs to.
3. Issue a CREATE TABLESPACE statement giving it a new desired name. Ensure that you also provide the IN keyword to reference it back to the database it belongs to.
4. Issue any ALTER TABLE statements to move any tables from the old table space over to the new table space (data set group).
5. Issue a COMMIT DDL statement.

Related reference:

➡ DROP TABLESPACE (Application Programming APIs)

Altering IMS tables

To make changes to a table, you can issue an ALTER TABLE statement. Depending on the database access types, some attributes do not apply so changes for them are not allowed.

For a list of supported keyword attributes based on database access type, see the ALTER TABLE (Application Programming APIs) syntax diagrams.

For HALDB databases, you can change the data set group a table belongs to by issuing an ALTER TABLE statement with the DSGROUP keyword.

The supported changes include but are not limited to:
- Assigning it to a different table space.
- Changing the table sizes.
- Changing the EDITPROC (compression exit).
- Changing the change data capture exit.
- Changing the columns definitions.
- Changing the map definitions.

Restriction: If you specify any of the following keywords on the CREATE TABLE statement, you cannot change the keyword and keyword value by using the ALTER TABLE statement. To change the keyword and keyword value, you must first use the DROP TABLE statement to delete the table. Then, you must re-create the table by using the CREATE TABLE statement and specify the keyword and keyword value again.
- The INTERNALNAME*internalname* keyword to specify the internal name of the segment type being defined.
- DIRECT DEPENDENT | SEQUENTIAL DEPENDENT

Related reference:

➡ ALTER TABLE (Application Programming APIs)

Creating a table:

You can define a new table to an existing database by issuing a CREATE TABLE statement. This can be useful if you find that you need to add another child to the hierarchy of your database.

Ensure you do not exceed the maximum number of tables for your database type.

1. Issue an ALTER DATABASE statement to identify the database in which you plan to include the new table.
2. Issue a CREATE TABLE statement. Ensure that you also provide the IN keyword to reference it to the database and table space it belongs to.
3. Specify a FOREIGN KEY in your CREATE TABLE statement to identify its parent.
4. Issue a COMMIT DDL statement.

Related reference:

➡ CREATE TABLE (Application Programming APIs)

Dropping a table:

You can remove an existing table by issuing a DROP TABLE statement.

Ensure that if you are removing a table, you do not break the current hierarchy of the database. Additional modifications to your database may be needed in order for your DROP TABLE to be valid. It is recommended if you are changing the hierarchy of your database that you create a new database with similar attributes instead.

1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.
2. Issue a DROP TABLE statement. Ensure that you also provide the IN keyword to reference it to the database it belongs to.
3. Issue additional DROP TABLE statements for any child tables that you do not want to keep.
4. Issue any ALTER TABLE statements to change the FOREIGN KEYs of any affected tables in the hierarchy.
5. Issue a COMMIT DDL statement.

Related reference:

➡ DROP TABLE (Application Programming APIs)

Adding a column:

When adding a new column to an existing table, you will use the ALTER TABLE statement with the ADD COLUMN clause. By default the column will be added towards the end of the table. You can define the new column to overlap existing columns in the table by specifying the actual starting offset with the START keyword. The physical records are not actually changed until values are inserted in the new columns.

New columns added can be defined to be searched within IMS with an INTERNALNAME. Program views can then be created to have field level sensitivity towards these columns. The program view changes are not necessary if the columns are not searchable within IMS (without the INTERNALNAME).

1. Issue an ALTER DATABASE statement to identify the database in which you plan to modify.
2. Issue an ALTER TABLE statement. Ensure that you also provide the IN keyword to reference it to the database it belongs to. As part of the ALTER TABLE statement, include an ADD COLUMN clause.
3. Issue a COMMIT DDL statement.

Altering a column:

You can change the attributes of a column by using the ALTER TABLE statement with an ALTER COLUMN clause. To change the name of a column, use the RENAME COLUMN clause instead.

Issue the RENAME COLUMN clause.

Dropping a column:

You can drop a column by using the ALTER TABLE statement with a DROP COLUMN clause. Only one DROP COLUMN clause is supported in an ALTER TABLE statement. To drop multiple columns, use multiple ALTER TABLE statements.

When you drop a column, this can produce a range of undefined space in the table if there are no other columns that map the same space. IMS does not shift the remaining columns together. Any subsequent ADD COLUMN clauses will, by default, add the new columns at towards the end of the table after the last column. You may optionally specify an ADD COLUMN, defining a new column in the space of the previous one you dropped by specifying the START keyword on the clause.

Altering IMS indexes

You can add or remove a secondary index from a full-function or a DEDB database.

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

"How secondary indexes restructure the hierarchy of databases" on page 342

Related tasks:

"Dropping an index" on page 749

Altering IMS logical relationships

The steps in the table below outline the steps required to add or modify logical relationships by reorganizing a database.

The table below shows you what you need to do based on the logical relationships and pointers you are starting from in column 1, the databases you are reorganizing in column 2, and the logical relationships and pointers you are adding in columns 3 to 6. Column 1 is the "FROM" column. Columns 3 to 6 are the "TO" columns.

At the intersection between the existing and new logical relationships and pointers, the table provides the following information:
- When a logically related database must be scanned
- When both sides of a logical relationship must be reorganized
- When the Prefix Resolution and Prefix Update utilities must be run

If the Prefix Resolution and Prefix Update utilities do not need to be run, the table cell contains "finished".

The figure applies to reorganizations only. When initially loading databases, you must run the Prefix Resolution and Update utilities whenever work data sets are generated.

The following table covers all reorganization situations, whether or not database pointers are being changed. In using the figure, a bidirectional physically paired relationship should be treated as two unidirectional relationships. Unless otherwise specified, DBR should be specified for the reorganized databases when the Prereorganization utility is run.

The following two examples guide you in use of the table.

Example 1: Reorganizing a database with unidirectional symbolic pointers without changing pointers

Assume your database has unidirectional symbolic pointers and you are not changing pointers. On the left side of the table below, in the FROM column, find unidirectional symbolic pointers and follow the row across to the right until it intersects with the TO column of unidirectional symbolic pointers. The figure tells you what you must do to reorganize with one of the following:

- The database containing the logical parent
- The database containing the logical child
- Both databases, if necessary

In all three situations, it is not necessary to run the Prefix Resolution or Update utilities (this is what is meant by "finished").

Example 2: Changing bidirectional symbolic pointers to bidirectional direct pointers during a reorganization

Assume your database has bidirectional symbolic pointers and you need to change to bidirectional direct pointers. In the FROM column, find the row for bidirectional symbolic pointers. In the TO columns, find bidirectional direct pointers. Where the row and column intersect, the table below shows that:

- Reorganizing only the logical parent database cannot be done, because a logical parent pointer must be created in the logical child segment in the logical child database.
- Reorganizing the logical child database can be done. To scan the logical child database, you must scan the logical parent database. The control statements for the Prereorganization utility must specify DBIL for the logical child database. Also, the Prefix Resolution and Update utilities must be run.
- Reorganizing both databases can also be done. In this case, the control statements for the Prereorganization utility must specify DBIL for the logical child database and DBR for the logical parent database. Again, the Prefix Resolution and Update utilities must be run.

Table 72. Steps required to convert logical relationships and pointers during a database reorganization.

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	**Unidirectional direct pointers**	**Bidirectional symbolic pointers**	**Bidirectional direct pointers**
Unidirectional with symbolic pointers	Logical parent database only	Finished[1]	Not valid, because symbolic LP pointers exist now and direct LP pointers must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** Logical child segment will not contain LT pointers unless it is reorganized.	Not valid, because direct LP and LT pointers must be put in the logical child database.
	Logical child database only	Finished	1. Scan logical parent database. 2. Run prefix resolution and update. Specify DBIL for the logical child database.	Not valid, because a counter exists now and LCF/LCL pointers must be put into the logical parent database.	Not valid, because a counter exists now and LCF/LCL pointers must be put into the logical parent database.
	Both databases	Finished[2]	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBR for both databases.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.
Unidirectional with direct pointers	Logical parent database only	Not valid, because a direct LP pointer exists now and symbolic LP pointers must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update.	Not valid, because a direct LP pointer exists now and symbolic LP pointers must be added to the logical child database. LT pointers must also be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** Logical child segment will not contain LT pointers unless database is reorganized.
	Logical child database only	Finished	Finished	Not valid, because LCF/LCL pointers must be put in the logical parent database.	Not valid, because LCF/LCL pointers must be put in the logical parent database.
	Both databases	Finished[2]	Run prefix resolution and update.	Run prefix resolution and update.	Run prefix resolution and update.

Table 72. Steps required to convert logical relationships and pointers during a database reorganization (continued).

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	Unidirectional direct pointers	Bidirectional symbolic pointers	Bidirectional direct pointers
Bidirectional with symbolic pointers	Logical parent database only	Not valid, because the counter in the logical parent database will not be resolved and LT pointers exist now in the logical child database.	Not valid, because symbolic LP and LT pointers exist now and a direct LP pointer must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** LCF/LCL pointers are not unloaded and reloaded.	Not valid, because a symbolic LP pointer exists now and a direct LP pointer must be added to the logical child database.
	Logical child database only	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	1. Scan logical parent database. 2. Run prefix resolution and update.	1. Scan logical parent database. 2. Run prefix resolution and update. 3. Specify DBIL for the logical child database.
	Both databases	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.

Table 72. Steps required to convert logical relationships and pointers during a database reorganization (continued).

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	**Unidirectional direct pointers**	**Bidirectional symbolic pointers**	**Bidirectional direct pointers**
Bidirectional with direct pointers	Logical parent database only	Not valid, because direct LP and LT pointers exist in the logical child database and symbolic LP pointers must be added.	Not valid, because the counter in the logical parent database will not be resolved and LT pointers will not be removed from the logical child database.	Not valid, because a direct LP pointer exists in the logical child database and the change is to symbolic LP pointers.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** LCF/LCL pointers are not unloaded and reloaded.
	Logical child database only	Not valid, because LCF/LCL pointers exists in the logical parent database and a counter must be added to the logical parent database.	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	1. Scan logical parent database. 2. Run prefix resolution and update.	1. Scan logical parent database. 2. Run prefix resolution and update.
	Both databases	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update.	Run prefix resolution and update.

Note:
1. The Prereorganization utility says to scan the logical child database and the DFSURWF1 records will be produced if scan is run.
2. DFSURWF1 records are produced; however, the prefix resolution and update utilities need *not* be run.

Related concepts:

"Defining sequence fields for logical relationships" on page 271

Adding logical relationships:

You can add logical relationships to an existing database.

This topic contains examples and procedures for adding a logically-related database to an existing database. Not all situations in which you might need to add a logical relationship are described in this topic. However, if the examples do not fit your specific requirements, you should be able to gather enough information from them to decide:

- If adding a logical relationship to your existing database is possible
- How to add the relationship

The examples show the logical parent as a root segment, although this is not a requirement. The examples are still valid when the logical parent is at a lower level in the hierarchy.

When adding logical relationships to existing databases, you should always make the change on a test database. Thoroughly test the change before implementing it using production databases.

In the following examples, these conventions are used:
- Existing databases are shown using solid lines.
- The database being added is shown using dashed lines.
- The logical parent and logical child relationship is labeled for the database being added. They are labeled LP and LC.
- The terms DBX, DBY, and DBZ refer to database 1, database 2, and database 3.

Related concepts:

Chapter 17, "Creating logical relationships," on page 245

Related tasks:

"Estimating the minimum size of the database" on page 559

"Writing a load program" on page 574

Defining program views with DDL

To create or modify an IMS program view (PSB) by using DDL, you code DDL statements and submit them to IMS by using the SQL Batch utility or a product like the IMS Enterprise Suite Explorer for Development.

Related reference:

➥ CREATE PROGRAMVIEW (Application Programming APIs)

High-level procedure for defining program views with DDL

You define new IMS program views by CREATE PROGRAMVIEW DDL statements. You alter IMS program views by using the DROP PROGRAMVIEW and CREATE PROGRAMVIEW DDL statements.

A program view, also known as a program specification block (PSB), must be created before you execute an application program under IMS. The program view is used to describe the program and its use of logical terminals and logical data structure.

A program view cannot be altered. Instead, you can DROP and then ADD a PSB. For example, if you want to add, drop, or delete a schema (PCB) from a program view, you need to drop the PSB, then recreate it with the PCB/schema included that you need.

You define the characteristics of an application program by using the CREATE PROGRAMVIEW statement. At least one is needed for each message, batch, or Fast Path program. The name of the program view and its associated application program must be the same in a telecommunications system.

The CREATE PROGRAMVIEW statements describe the IMS resources to be used through a list of schemas, also known as program communication blocks (PCBs).

If you are creating a new database with a CREATE DATABASE statement, create a program view within the same DDL commit scope, so that the database is useable.

Recommendation: Use the IMS Enterprise Suite Explorer for Development to create or modify database and program view definitions and submit them to IMS. Using IMS Explorer for Development, you can create new database and program view definitions from the existing databases and program views in your IMS system and then modify the definitions as necessary.

To create a new program view:

1. Code your DDL statements.
   a. If you are modifying an existing program view, include a DROP PROGRAMVIEW statement.
   b. Define the program view by coding the CREATE PROGRAMVIEW, CREATE SCHEMA, and CREATE SENSEGVIEW statements.
   c. Include a COMMIT statement as the final statement.
2. Submit the DDL statements to IMS by using either the IMS Enterprise Suite Explorer for Development or the IMS SQL Batch utility. Upon receiving the DDL with a COMMIT statement, IMS takes the following action:
   • Validates the DDL syntax and confirms that the program view does not already exist.
   • Updates the PSB record in the IMS catalog
   • Generates the required application control blocks (ACBs) and stores them in the staging data set of the IMS directory.
   • Updates the resource list indicating this program was updated.
3. Issue the IMPORT DEFN SOURCE(CATALOG) command to activate the program view definitions. IMS activates the pending change in the catalog. If you are adding program views only, you have can activate the program views on a subset of IMS systems. For more information, see Activating PSBs in subsets of systems in an IMSplex (System Administration).

To modify an existing program view:

4. If your installation maintains PSB, DBD, and ACB libraries, update the libraries to reflect the new or modified program views by using the IMS Catalog Library Builder utility (DFS3LU00).

Related tasks:

"Dropping program views" on page 518

Related reference:

⮕ CREATE PROGRAMVIEW (Application Programming APIs)

⮕ IMPORT DEFN SOURCE(CATALOG) command (Commands)

Including a schema in a program view

When defining a schema, also known as a program communication block (PCB), to a program view use the CREATE SCHEMA clause. The CREATE SCHEMA must be specified as a clause within the CREATE PROGRAMVIEW statement.

The schema represents message destinations or database resources used by the application program. One or more are usually included, however are not always required. For example, a message switching program or conversational message program might not require access to a DL/I database.

The maximum number of schemas that can be defined for a given program view is 2500. This is the maximum value for application programs executing in all IMS™ region types (MSG, DL/I, and so on).

Related reference:

➡ CREATE PROGRAMVIEW (Application Programming APIs)

Including segment sensitivity in a program view

The CREATE SENSEGVIEW clause is used to define a hierarchically related set of tables to a schema of a program view. This set represents tables to which a program through a schema is sensitive.

The table identified by this clause can physically exist in one database or can be derived from several physical databases. One or more of these clauses can be included for a schema and can only be specified within a CREATE SCHEMA clause.

There must be one senseg view for each table to which the application program is sensitive. All tables in the hierarchic path to any required table must be specified. A maximum of 30,000 senseg views can be defined in a single program view.

The order in which senseg views are sequenced in a schema determines the logical access order for the tables.

Related reference:

➡ CREATE PROGRAMVIEW (Application Programming APIs)

Including field sensitivity in a program view

The field sensitivity clause is used with the senseg view clause (CREATE SENSEGVIEW) to indicate those columns within a table to which an application program is sensitive.

One or more field sensitivity clause can be included witin a senseg view clause. You can define a maximum of 255 for a given senseg view. You can define a maximum of 10,000 in a single program view.

The same column can be referenced in more than one field sensitivity clause within a senseg view. If the duplicate names participate in a concatenated table and the same column appears in both portions of the concatenation, the first reference will be to the logical child, and all subsequent references will be to the logical parent. This referencing sequence determines the order in which columns will be moved to the user's I/O area.

For retrieve-only processing you can request, using the field sensitivity clause, that the same data be moved to multiple locations in your I/O area, provided that no overlapping occurs, and that field sensitivity of variable-length tables are of the same type.

The following restrictions apply:
- The column defining the length of a variable-length table cannot be referenced through a field sensitivity clause.
- A field sensitivity clause cannot appear within a senseg view with PROCOPT K.
- A field sensitivity clause cannot appear within a senseg view with PROCOPT I or L, if the senseg view refers to a logical child table.
- If field sensitivity clauses are used with PROCOPT I or L, a field sensitivity clause must be included for the primary key.
- Not allowed for DEDB.

Related reference:

➡ CREATE PROGRAMVIEW (Application Programming APIs)

Dropping program views

You can drop an IMS program view by issuing the DROP PROGRAMVIEW statement. When you drop a program view, all of its schemas (PCB), senseg views, and field sensitivity descriptions (SENFLD) are dropped too. Issuing DROP will remove all instances of the program view from the IMS catalog and render it inactive to IMS.

You will need to remove the DRD resources.

IMS does not have an ALTER PROGRAMVIEW statement. If you want to implement a change to an existing program view or schema you must perform the following steps.

1. Open IMS Enterprise Suite Explorer for Development to import the CREATE PROGRAMVIEW statements that make up your program view.
2. Modify the CREATE statements to include your changes.
3. Issue a DROP PROGRAMVIEW statement.
4. Issue the modified CREATE PROGRAMVIEW statement.
5. Issue a COMMIT DDL statement.

Related reference:

➡ DROP PROGRAMVIEW (Application Programming APIs)

Altering a schema in a program view

You cannot alter a schema in a program view. Instead, you must first drop the program view, create a program view, modify the CREATE statement and commit your changes.

For steps on how to perform this task, see "Dropping program views" and "Including a schema in a program view" on page 516.

Altering segment sensitivity in a program view

You cannot alter segment sensitivity in a program view. Instead, you must first drop the program view, create a program view, modify the CREATE statement and commit your changes.

For steps on how to perform this task, see "Dropping program views" and "Including segment sensitivity in a program view" on page 517.

Altering field sensitivity in a program view

You cannot alter field sensitivity in a program view. Instead, you must first drop the program view, create a program view, modify the CREATE statement and commit your changes.

For steps on how to perform this task, see "Dropping program views" and "Including field sensitivity in a program view" on page 517.

Altering processing options

You cannot alter processing options. Instead, you must first drop the program view, create a program view, modify the CREATE statement and commit your changes.

For steps on how to perform this task, see "Dropping program views."

Using the IMS generation utilities to design IMS databases

You can implement your database designs in IMS by coding macro instructions as input to the Database Description (DBD) Generation utility and the Program Specification Block (PSB) Generation utility.

Using the macro instructions, you describe the characteristics of each database and each application program to IMS. By reading the macro instructions, the utilities create DBD and PSB control blocks that are used by IMS and the ACB Maintenance utility. The ACB Maintenance utility generates the application control blocks (ACBs) that are used by IMS during runtime.

In IMS systems that use ACB libraries, the use of macro instructions and the generation utilities is required. In IMS systems that manage ACBs for you, the use of macro instructions and the generation utilities is optional.

If you use macro instructions to implement database designs in IMS systems that manage ACBs, you must populate the IMS catalog before the DBDs, PSBs, and ACBs can be used by IMS, because IMS does not use the DBD, PSB, and ACB libraries that are created by the generation utilities.

Coding database descriptions as input for the DBDGEN utility

One way to define a *database descriptor* (DBD) is with a series of macro instructions that describes such things as a database's organization and access method, the segments and fields in a database record, and the relationships between types of segments.

After you have coded the DBD macro instructions, they are used as input to the DBDGEN utility. This utility is a macro assembler that generates a DBD control block and stores it in the IMS.DBDLIB library for subsequent use during database processing.

If an IMS catalog is enabled, the information you code in the DBD macro instructions also provides much of the database and application program metadata that is stored in the IMS catalog. This metadata includes such things as field data types, application program data structures, date and time formats, and more. The metadata is read into the catalog primarily from the IMS.ACBLIB data set after the ACB members are generated, but in some cases is also read directly from the IMS.DBDLIB data set.

The following figure illustrates the DBD generation process.
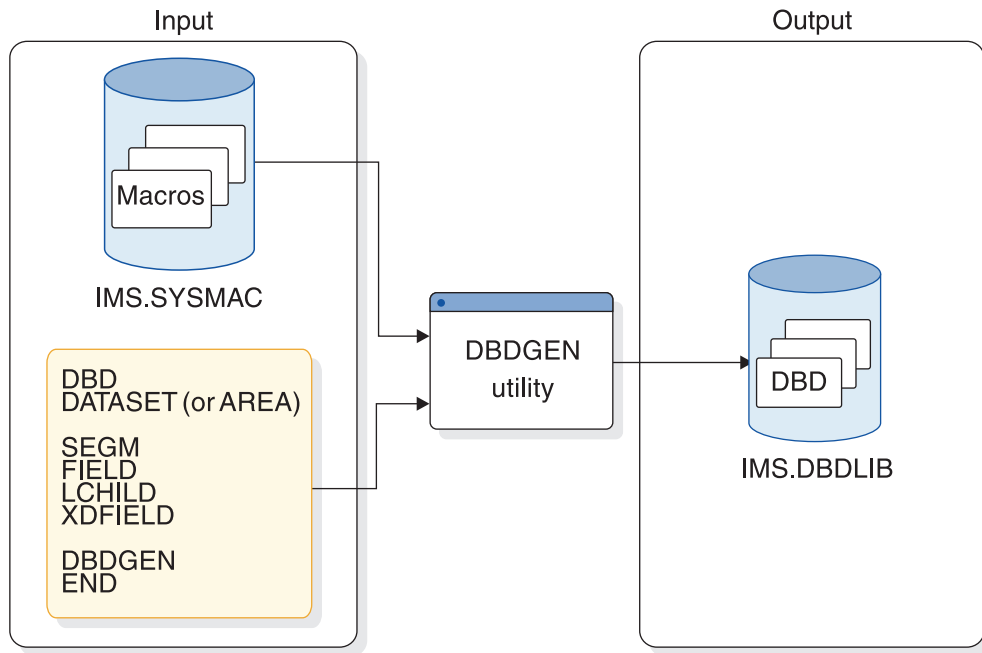
Figure 231. The DBD generation process

The following JCL shows an example of the input to the DBDGEN utility. Separate input is required for each database being defined.

```
//DBDGEN     JOB  MSGLEVEL=1
//                   EXEC        DBDGEN,MBR=APPLPGM1
//C.SYSIN        DD    *

   DBD                required for each DBD generation
   data set(or AREA)   required for each data set group
                       (or AREA in a Fast Path DEDB)
   SEGM               required for each segment type
   FIELD              required for each DBD generation
   LCHILD             required for each secondary index or
                       logical relationship
   XDFLD              required for each secondary index relationship
      .
      .
      .
   DBDGEN             required for each DBD generation
   END                required for each DBD generation
/*
```

In addition to the statements shown in the preceding example, the input to the DBDGEN utility can also include the following macro statements:

- DFSMARSH, which defines data marshalling properties for individual fields
- DFSMAP and DFSCASE, which define alternative fields maps for a segment

Related tasks:

"Creating HALDB databases with the HALDB Partition Definition utility" on page 542

Related reference:

▶ Database Description (DBD) Generation utility (System Utilities)

DBD statement overview

In the input, the DBD statement names the database being described and specifies various attributes including its organization and its exit routines, if any.

Code only one DBD statement in the input deck.

Related reference:

➡ DBD statements (System Utilities)

DATASET statement overview

The DATASET statement defines the physical characteristics of the data sets to be used for the database.

At least one DATASET statement is required for each data set group in the database. Depending on the type of database, up to 10 data set groups can be defined. Each DATASET statement is followed by the SEGM statements for all segments to be placed in that data set group.

The DATASET statement is not allowed for HALDB databases. Use either the HALDB Partition Definition utility to define HALDB partitions or the DBRC commands INIT.DB and INIT.PART

If the database is a DEDB, the AREA statement is used instead of the DATASET statement.

Related reference:

➡ DATASET statements (System Utilities)

➡ AREA statement (System Utilities)

AREA statement overview

The AREA statement defines an area of a Fast Path data entry database (DEDB).

At least one AREA statement is required, but as many as 2,048 AREA statements can be used to define multiple areas.

All AREA statements must be put between the DBD statement and the first SEGM statement.

The AREA statement is used for Fast Path DEDB database types only. Full-function databases use the DATASET statement instead.

Related reference:

"AREA segment type format" on page 63

➡ AREA statement (System Utilities)

➡ DATASET statements (System Utilities)

SEGM statement overview

The SEGM statement defines a segment type in the database, the position of the segment in the hierarchy, the physical characteristics of the segment, and the relationship of the segment to other segments.

SEGM statements are put in the input deck in hierarchical sequence, and a maximum of 15 hierarchical levels can be defined. The number of database statements allowed depends on the type of database. SEGM statements must immediately follow the data set or AREA statements to which they are related.

Related reference:

SEGM statements (System Utilities)

"SEGM segment type format" on page 90

FIELD statement overview

The FIELD statement defines a field within a segment type.

A FIELD statement is required for the following types of fields:
- Sequence field in a segment
- Fields that an application program refers to by name in a segment search argument (SSA) of a DL/I call
- Fields referenced in a SENFLD statement in a PSB
- Fields referenced in an XDFLD statement

In the FIELD statements for each of the preceding field types, you must specify the NAME parameter. When the NAME parameter is specified, the field name is stored in the database data management block (DMB). To save storage in the DMB, specify the EXTERNALNAME parameter only, unless the NAME parameter is required.

You can also code FIELD statements to define data fields and structures to IMS that would otherwise be defined only in application source code or copy books. The advantage to coding these optional FIELD statements is that, when the IMS catalog is enabled, application developers and others can query the IMS catalog for metadata about these data fields and structures, instead of having to locate the application source code or generate the metadata by using IMS Enterprise Suite Explorer for Development. For these types of optional FIELD statements, if you code the EXTERNALNAME parameter, the NAME parameter is not required.

FIELD statements can be put in the input deck in any order, except in the following cases:
- FIELD statements must immediately follow the SEGM statement to which they are related.
- A sequence field, if one is defined, must always be first
- If a FIELD statement specifies the CASENAME parameter, the FIELD statement must follow the DFSCASE statement that is specified on the CASENAME parameter
- If a FIELD statement specifies the DEPENDSON parameter, the FIELD statement must follow the FIELD statement that is specified on the DEPENDSON parameter
- If a FIELD statement specifies the REDEFINES parameter, the FIELD statement must follow the FIELD statement that is specified on the REDEFINES parameter
- If a FIELD statement specifies the STARTAFTER parameter, the FIELD statement must follow the FIELD statement that is specified on the STARTAFTER parameter
- If a FIELD statement specifies the PARENT parameter, the FIELD statement must follow the FIELD statement that is specified on the PARENT parameter

You can define up to 1,000 fields in a database.

Within each segment type, you can include up to 255 FIELD statements that specify the NAME parameter. The NAME parameter enables a field for SSA

searches. If you omit the NAME parameter, there is no limit to the number of FIELD statements that you can define for a segment, as long as the number of fields defined does not exceed the maximum of 1000 fields for the database. However, fields defined without the NAME parameter cannot be specified on an SSA and are not searchable by IMS.

The definition of fields within a segment can overlap. For example, a date "field" within a segment can be defined as three 2-byte fields and also as one 6-byte field as shown in the following figure.
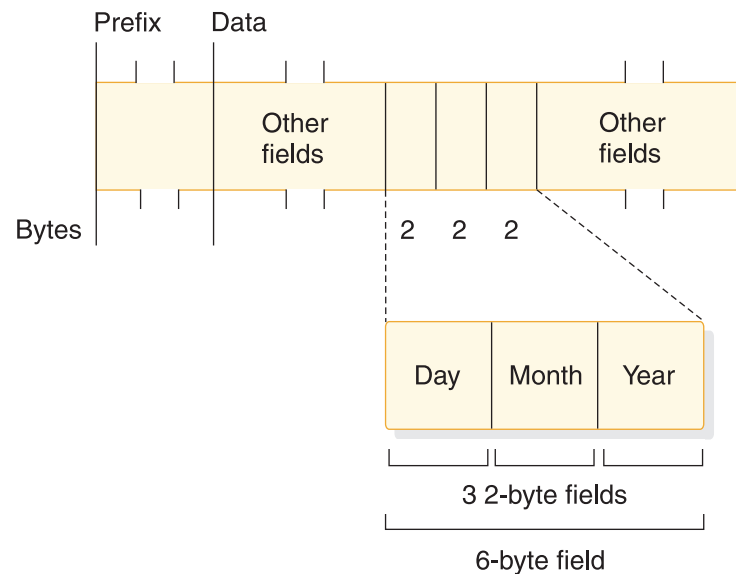
Figure 232. Example of a date field within a segment defined as three 2–byte fields and one 6–byte field

The technique of overlapping field definitions allows application programs to access the same piece of data in various ways. To access the same piece of data in various ways, you code a separate FIELD statement for each field. For the example shown, you would code four FIELD statements, one for the total 6-byte date and three for each 2-byte field in the date.

Related reference:

▶ FIELD statements (System Utilities)

DFSMARSH statement overview
The DFSMARSH statement defines marshalling attributes for field data.

You can use the DFSMARSH statement to specify the following data marshalling attributes for the data contained in a field:
- You can specify the code page or character encoding that defines the character data in a field on the ENCODING parameter. The default is the EBCDIC code page Cp1047.
- You can specify a data-conversion routine for IMS to use when converting field data from the data type that IMS uses to physically store data to a data type expected by an application program. You can specify an IMS-provided routine on the INTERNALTYPECONVERTER parameter or a user-provided routine on the USERTYPECONVERTER parameter.
- You can specify whether a decimal data type is signed or not on the ISSIGNED parameter.

- The pattern to use for dates and times can be specified on the PATTERN parameter.
- You can specify the properties that are used by a user-provided data-conversion routine on the PROPERTIES parameter.

In the input to the DBD Generation utility, the DFSMARSH statement must immediately follow the FIELD statement to which it applies.

Related reference:

➡ DFSMARSH statements (System Utilities)

LCHILD statement overview

The LCHILD statement defines a secondary index or logical relationship between two segment types, or the relationship between a HIDAM (or PHIDAM) index database and the root segment type in the HIDAM (or PHIDAM) database.

LCHILD statements immediately follow the SEGM, FIELD, or XDFLD statement of the segment involved in the relationship. Up to 255 LCHILD statements can be defined for each database.

Restriction: The LCHILD statement cannot be specified for the primary index of a PHIDAM database because the primary index is automatically generated.

Related reference:

➡ LCHILD statements (System Utilities)

XDFLD statement overview

The XDFLD statement is used only when a secondary index exists.

It is associated with the target segment and specifies:
- The name of an indexed field.
- The name of the source segment. The source segment can be the same as the target segment or a dependent of the target segment.
- The field in the source segment that contains the data that is used in the secondary index.

Up to 32 XDFLD statements can be defined per segment. However, the number of XDFLD and FIELD statements combined cannot exceed 255 per segment or 1000 per database.

Restriction: The CONST parameter is not allowed for a HALDB database. Shared secondary indexes are not supported.

Related reference:

➡ XDFLD statements (System Utilities)

DFSMAP statement overview

The DFSMAP statement defines a *map* that references a control field for a set of map cases.

Each map case in a set references the map by the name defined on the NAME parameter in the DFSMAP statement.

The map references the control field by the external name of the field on the DEPENDINGON parameter of the DFSMAP statement.

Related tasks:

"Defining alternative field maps for a segment" on page 538

Related reference:

➡ DFSMAP statements (System Utilities)

DFSCASE statement overview

The DFSCASE statement defines a *map case*, a conditional mapping of a set of fields in a segment.

A map case has a name and an ID.

The fields that make up a map case specify the name of the map case that they belong to on the CASENAME parameter in the FIELD statement.

The ID of a map case is specified on the CASEID parameter. In a segment instance, the ID is inserted in a control field to indicate which map case is in use.

A map case ID can be either Cp1047 character data or hexadecimal data. The data type of the ID is specified on the CASEIDTYPE parameter. The length of the control field must be compatible with the data type selected for the map case ID.

Each map case belongs to a *map*, which is defined by the DFSMAP statement. The map references the control field that determines which map case to use in a given segment instance. The map case specifies the map that it belongs to on the MAPNAME parameter.

Related reference:

➡ DFSCASE statements (System Utilities)

DBDGEN and END statements overview

One DBDGEN statement and one END statement is put at the end of each DBD generation input deck.

These statements specify:
- The end of the statements used to define the DBD (DBDGEN)
- The end of input statements to the assembler (END)

Related reference:

➡ DBDGEN statements (System Utilities)

Coding program specification blocks as input to the PSBGEN utility

A PSB is a series of macro instructions that describes an application program's characteristics, its use of segments and fields within a database, and its use of logical terminals.

A PSB consists of one or more PCBs (program communication blocks). Of the two types of PCBs, one is used for alternate message destinations, the other, for application access and operation definitions.

After you code the PSB macro instructions, they are used as input to the PSBGEN utility. This utility is a macro assembler that generates a PSB control block then stores it in the IMS.PSBLIB library for subsequent use during database processing.

The following figure shows the PSB generation process.
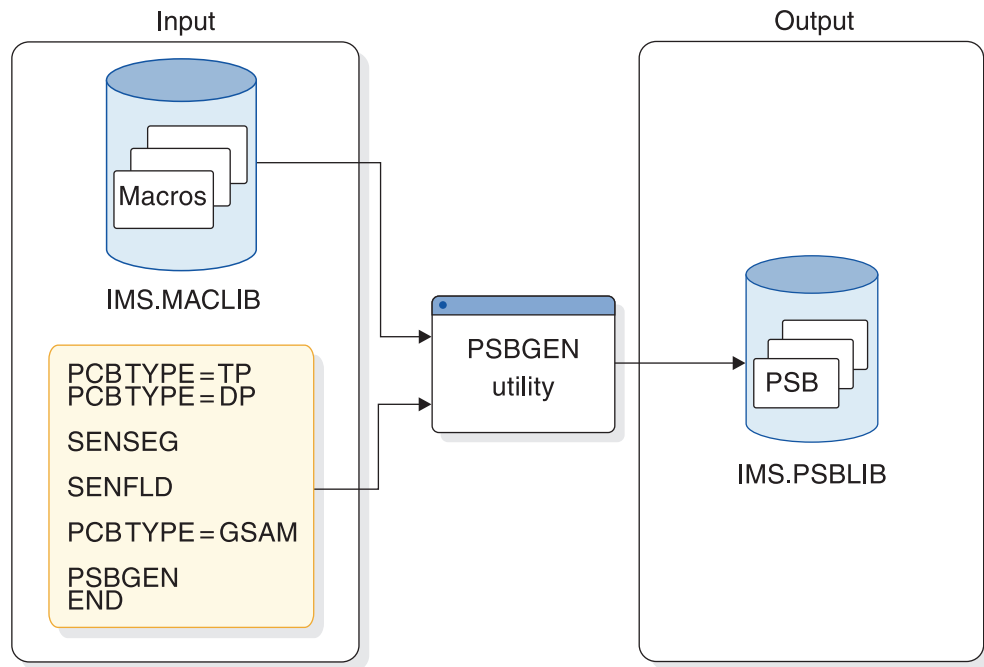
Figure 233. The PSB generation process

The following JCL shows the structure of the deck used as input to the PSBGEN utility.

```
//PSBGEN     JOB  MSGLEVEL=1
//               EXEC        PSBGEN,MBR=APPLPGM1
//C.SYSIN      DD     *

    PCB TYPE=TP        required for output message destinations
    PCB TYPE=DB        required for each database the application program
                        can access
    SENSEG             required for each segment in the database the
                        application program can access
    SENFLD             required for each field in a segment that
                        the application program can access,
                        when field-level sensitivity is specified
    PCB TYPE=GSAM
    .
    .
    .
    PSBGEN             required for each PSB generation
    END                required for each PSB generation
/*
```

Related reference:

The alternate PCB statement

Two types of PCB statements can be placed in the input deck: the *alternate* PCB statement and the *database* PCB statement.

The alternate PCB statement describes where a message can be sent when the message's destination differs from the place where it was entered. Alternate PCB statements must be put at the beginning of the input deck. More information on alternate PCBs is contained in *IMS Version 15 System Administration*.

The database PCB statement

The database PCB statement defines the DBD of the database that the application program will access.

Database PCB statements also define types of operations (such as get, insert, and replace) that the application program can perform on segments in the database. The database can be either physical or logical. A separate database PCB statement is required for each database that the application program accesses. In each PSB generation, up to 2500 database PCBs can be defined, minus the number of alternate PCBs defined in the input deck. The other forms of statements that apply to PSBs are SENSEG, SENFLD, PSBGEN, and END.

Attention: If the PROCOPT values allow a BMP application to insert, replace, or delete segments in databases, ensure that the BMP application does not update a combined total of more than 300 databases and HALDB partitions without committing the changes.

All full-function database types, including logically related databases and secondary indexes, that have uncommitted updates count against the limit of 300. When designing HALDB databases, you should use particular caution because the number of partitions in HALDB databases is the most common reason for approaching the 300 database limit for uncommitted updates.

The SENSEG statement

The SENSEG statement defines a segment type in the database to which the application program is sensitive.

A separate SENSEG statement must exist for each segment type. The segments can physically exist in one database or be derived from several physical databases. If an application program is sensitive to a segment beneath the root segment, it must also be sensitive to all segments in the path from the root segment to the sensitive segment. For example, in the following figure if D is defined as a sensitive segment for an application program, B and A must also be defined as sensitive segments.
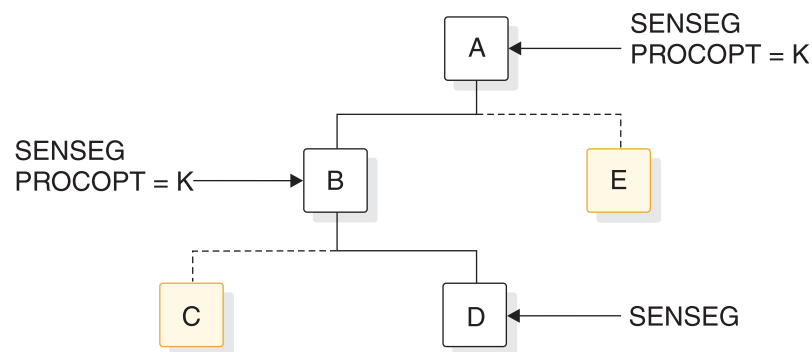
Figure 234. Example of a SENSEG relationship

An application program must be sensitive to all segments in the path to the segment that you actually want to be sensitive. However, you can make the application program sensitive to only the segment key in these other segments. With this option, the application program does not have any access to the segments other than the keys it needs to get to the sensitive segment. To make an application sensitive to only the segment key of a segment, code PROCOPT=K in the SENSEG statement. The application program will not be able to access any

other field in the segment other than the segment key. In the previous example, the application program would be sensitive to the key of segment A and B but not sensitive to A and B's data.

SENSEG statements must immediately follow the PCB statement to which they are related. Up to 30000 SENSEG statements can be defined for each PSB generation.

Related reference:

"SS segment type format" on page 106

The SENFLD statement

The SENFLD statement is used only in parallel with field-level sensitivity and defines the fields in a segment type to which the application program is sensitive.

The SENFLD statement, in conjunction with the SENSEG statement, helps you secure your data. Each SENFLD statement must follow the SENSEG statement to which it is related. Up to 255 sensitive fields can be defined for a given segment type, and a maximum of 10000 can be defined for each PSB generation.

The PSBGEN statement

This statement names the PSB and specifies various characteristics of the application program, such as the language it is written in and the size of the largest I/O area it can use. The input deck can contain only one PSBGEN statement.

The END statement

One END statement is placed at the end of each PSB generation input deck.

The END statement specifies the end of input statements to the assembler.

Detailed instructions for coding PSB statements and examples of PSBs are contained in of *IMS Version 15 System Utilities*.

Building the application control blocks (ACBGEN)

You can create *application control blocks* (ACBs) for each DBD and each PSB that is used in your system by running either the ACB Maintenance utility or the ACB Generation and Catalog Populate utility (DFS3UACB).

Using one of these utilities to generate ACBs is required in IMS systems that use ACB libraries and optional in IMS systems that manage ACBs for you. The ACBMGMT= parameter in the CATALOG section of the DFSDF*xxx* PROCLIB member determines whether an IMS system uses ACB libraries or manages the ACBs for you.

The utilities generate the ACBs from the database definitions (DBDs) and program specification blocks (PSBs) in the IMS.DBDLIB and IMS.PSBLIB data sets that the utility reads as input. Each generated ACB is stored as an *ACB library member* in the IMS.ACBLIB data set.

Depending on which utility you use to generate the ACBs and whether the IMS catalog is enabled, the IMS catalog can be populated either during the ACB generation process or afterward.

The DFS3UACB utility generates the ACB and populates the IMS catalog in the same job step, which ensures that the IMS catalog stays current with the ACB library. If you use the ACB Maintenance utility, which only generates the ACB

members, the IMS catalog can be populated later by using the IMS Catalog Populate utility (DFS3PU00). However, if you populate the IMS catalog later, you risk running IMS with an IMS catalog that is not current to the ACB library.

Recommendation: If the IMS catalog is enabled in your system, use the DFS3UACB utility to build your ACB members and populate the IMS catalog in a single job step.
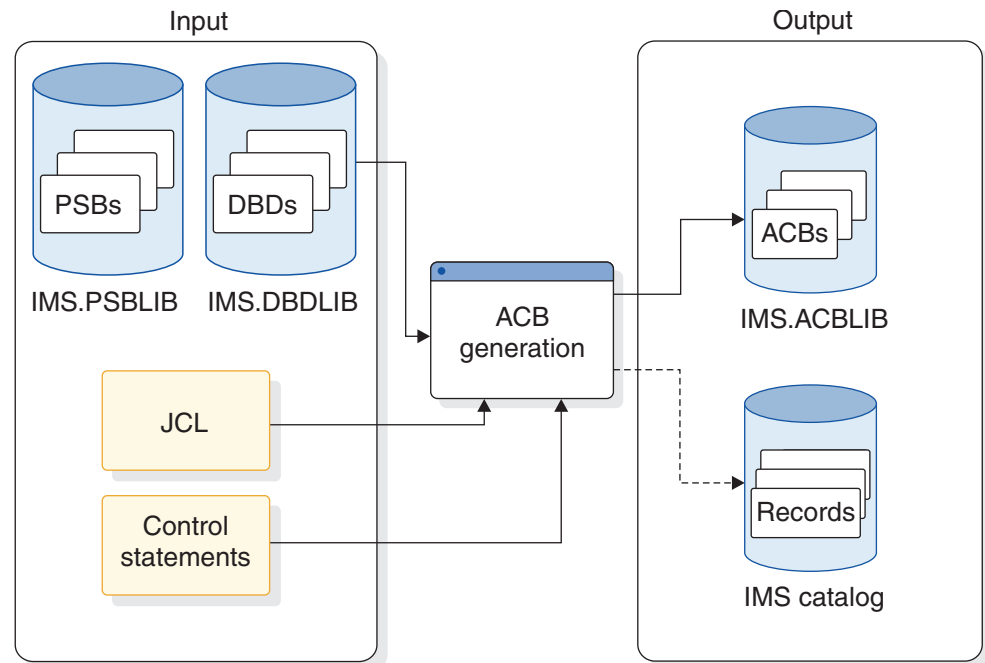
Figure 235. The ACB generation process

ACBs are loaded into storage for online use from the IMS.ACBLIB data set either when the IMS control region is initialized or when an application program that requires the ACB is scheduled, depending on whether the ACB is resident or non-resident.

In online storage, the ACBs for PSBs and the ACBs for DBDs are stored separately in a PSB pool and DMB pool, respectively. The storage for these pools must be allocated by using sizes based on the number and size of each type of ACB member.

If the program or database from which an ACB is built is defined as resident during system definition, IMS loads the ACB into 31-bit storage during the initialization of the IMS control region. Resident ACB members persist in storage until IMS is shut down.

If the program or database from which an ACB is built is not defined as resident during system definition, IMS loads the *non-resident* ACB into 31-bit storage only when an application program that requires it is scheduled. Non-resident ACB members persist in storage only until the storage allocated for the non-resident ACBs is exhausted, at which time IMS frees storage by removing the ACBs that are unused for the longest period.

Non-resident ACB members can also be cached in a 64-bit storage pool to potentially improve the performance of program scheduling and reduce the usage

of 31-bit storage. When 64-bit caching is enabled, when a non-resident ACB is first loaded into the 31-bit storage pool, it is also cached in the 64-bit storage pool. Later, if any non-resident ACBs that were removed from online storage are required by a scheduled application program, the ACBs are retrieved from 64-bit storage instead of from the IMS.ACBLIB data set on DASD.

Caching ACBs in the 64-bit storage is enabled by specifying the ACBIN64 parameter on the DFSDFxxx PROCLIB member. You can display information about the ACBs cached in 64-bit storage by using the type-2 command QUERY POOL TYPE(ACBIN64).

The sizes of the PSB pools and DMB pools are determined by the number and size of the ACB library members. Because PSBs and DMBs are stored together in 64-bit storage pools, the size of a 64-bit storage pool must be large enough to contain all of the non-resident PSBs and DMBs combined. The size of an ACB member is reported by the Application Control Blocks (ACB) Maintenance utility when the ACB members are built.

For execution in a batch environment, IMS can build ACBs dynamically or IMS can use prebuilt ACBs. To build ACBs dynamically, DLI must be specified on the PARM parameter of the batch startup procedure. To use prebuilt ACBs, DBB must be specified on the PARM parameter of the batch startup procedure. Prebuilt ACBs are stored in the IMS.ACBLIB data set pointed to by the IMSACB*x* DD statement in the startup procedure. IMS provides the DLIBATCH procedure to run a batch job that uses dynamically built ACBs and the DBBATCH procedure to run a batch job that uses prebuilt ACBs. For information about these procedures, see *IMS Version 15 System Definition*.

For batch application programs, IMS does not support 64-bit caching of ACBs.

For online application programs, ACBs must be prebuilt.

For GSAM DBDs, ACBs cannot be prebuilt. However, ACBs can be prebuilt for PSBs that reference GSAM databases.

You can prebuild ACBs for all PSBs in IMS.PSBLIB, for a specific PSB, or for all PSBs that reference a particular DBD. Prebuilt ACBs are kept in the IMS.ACBLIB library. The IMS.ACBLIB library is not used if ACBs are not prebuilt. When ACBs are prebuilt and an application program is scheduled, the application program's ACB is read from IMS.ACBLIB directly into storage. This means that less time is required to schedule an application program. In addition, less storage is used if prebuilt ACBs are used.

When the ACBs are prebuilt by a utility, the utility checks for errors in the names that are used in the PSB and the DBDs that are associated with the PSB. If erroneous cross-references are found, the utility prints appropriate error messages.

IMS.ACBLIB must be used exclusively. Because of this, the utility that generates the ACBs can be run only using an IMS.ACBLIB that is not currently allocated to an active IMS system. Also, because IMS.ACBLIB is modified, it cannot be used for any other purpose during execution of the utility that is generating the ACBs.

How you activate new and changed ACBs in an online system depends on whether the IMS management of ACBs is enabled in your IMS system. When the IMS management of ACBs is enabled, you can use either the DFS3UACB utility or the IMS Catalog Populate utility (DFS3PU00) to stage the ACBs to a staging data

set of the IMS catalog and issue the IMPORT DEFN SOURCE(CATALOG) command or, if IMS is shut down, you can have these utilities update the IMS directory data set directly.

If the IMS management of ACBs is not enabled, you can activate ACBs in an online IMS online system by using the online change function to switch the active ACBLIB with an "inactive" copy of the ACBLIB that contains the new and changed ACBs.

Additional information about creating and sizing DMB pools, PSB pools, and 64-bit non-resident ACB storage pools can be found in:
- *IMS Version 15 System Administration*
- *IMS Version 15 System Definition*

Related concepts:

▶ Allocating ACBLIB data sets (System Definition)

Related tasks:

"Steps for altering an online HALDB database" on page 758

"Activating database changes by using the online change function" on page 795

Related reference:

▶ Application Control Blocks Maintenance utility (System Utilities)

▶ Database Description (DBD) Generation utility (System Utilities)

▶ Program Specification Block (PSB) Generation utility (System Utilities)

▶ ACB Generation and Catalog Populate utility (DFS3UACB) (System Utilities)

Defining DBD and PSB metadata to the generation utilities

When the IMS catalog is used, it stores metadata about data types, data structures, access methods, and much more. This metadata is derived from the database descriptors (DBDs) and program specification blocks (PSBs) that you create when you define your databases and application programs views to IMS.

Some of the metadata is *database metadata*, which describes the physical database. Other metadata is *application program metadata*, which describes how the application program views or uses the data in the database. Much of the application program metadata that you code in the DBD comes from the requirements of your application programs and artifacts like COBOL copybooks.

The following elements are examples of things that you can code in a DBD that are described by application program metadata:
- External names for segments and fields, which are specified on the EXTERNALNAME parameter when you define a segment or field
- Complex data elements such as arrays and structures
- Field data types and data marshalling characteristics
- Character encoding
- Alternative field maps for segments

Coding the elements described by application program metadata in a DBD consolidates the data requirements of your application programs in a single, trusted location: the IMS catalog. When the IMS catalog is enabled, the metadata in

the IMS catalog can be queried and analyzed as you develop new application programs or assess the impact of changes to existing databases and application programs.

You define the metadata to IMS either by coding parameters in the DBD and PSB generation macro statements or by submitting DDL to IMS by using a tool such as the IMS Enterprise Suite Explorer for Development.

Related reference:

➡ Database Description (DBD) Generation utility (System Utilities)

➡ Program Specification Block (PSB) Generation utility (System Utilities)

Specifying data types for application programs

You can specify the data type that an application program expects in a given field by specifying the DATATYPE parameter in the FIELD statement that defines the field to the DBD Generation utility.

When the DATATYPE keyword is specified, a data type converter transforms the field data returned by IMS from the binary data type that IMS stores the physical data in to the data type specified on the DATATYPE parameter.

For each DATATYPE specification, IMS automatically selects a compatible IMS-provided data-type converter. If the data-type converter selected by IMS does not fit your needs exactly, you can select the data-type converter yourself, or you can provide your own custom data-type converter. Specify the IMS-provided data-type converters on the INTERNALTYPECONVERTER parameter or a custom data-type converter on the USERTYPECONVERTER parameter. Both parameters are specified in the DFSMARSH statement.

When specifying a data type, you must ensure that the byte length of the field supports the data type. Some data types require a specific field length. For example, if you specify DATATYPE=LONG, you must define the length of the field as eight bytes by specifying BYTES=8.

If no data type is specified, the default data type returned to an application program is determined by the value of the TYPE parameter. If TYPE=C is specified or defaulted to, the default application data type is CHAR. For all other specifications of the TYPE parameter, the default application data type is BINARY.

You can define additional characteristics of your data type selections, such as the character encoding, date and time formats, and decimal types, by specifying additional parameters on the DFSMARSH statement.

Examples of the DFSMARSH statement

The following series of examples show some possible uses of the DFSMARSH statement for various DATATYPE and type converter specifications.

DATATYPE=DATE:

```
FIELD    EXTERNALNAME=XDATE,
         BYTES=8,
         START=84,
         DATATYPE=DATE
DFSMARSH ENCODING=Cp1047,
         INTERNALTYPECONVERTER=CHAR,
         PATTERN='MMddyyyy'
```

DATATYPE=TIME:

```
FIELD    EXTERNALNAME=XTIME,
         BYTES=6,
         START=92,
         DATATYPE=TIME
DFSMARSH ENCODING=Cp1047,
         INTERNALTYPECONVERTER=CHAR,
         PATTERN='HHmmss'
```

DATATYPE=TIMESTAMP:

```
FIELD    EXTERNALNAME=XTIMESTAMP,
         BYTES=16,
         START=84,
         DATATYPE=TIMESTAMP
DFSMARSH ENCODING=Cp1047,
         INTERNALTYPECONVERTER=CHAR,
         PATTERN='MMddyyyyHHmmssff'
```

DATATYPE=ZONEDDECIMAL:

```
FIELD NAME=ORDPRICE,
         BYTES=10,
         START=21,
         DATATYPE=DECIMAL(10,2)
DFSMARSH INTERNALTYPECONVERTER=ZONEDDECIMAL,
         ISSIGNED=Y
```

DATATYPE=PACKEDDECIMAL:

```
FIELD    EXTERNALNAME=XPACKEDDEC1,
         BYTES=4,
         START=60,
         DATATYPE=DECIMAL(7,2)
DFSMARSH INTERNALTYPECONVERTER=PACKEDDECIMAL,
         ISSIGNED=Y
```

USERTYPECONVERTER=:

```
FIELD    EXTERNALNAME=PACKEDDATEFIELD,
         BYTES=5,
         START=40,
         DATATYPE=DATE
DFSMARSH USERTYPECONVERTER=class://com.ibm.ims.dli.types.PackedDateConverter,
         PROPERTIES=(ZONE=PACIFIC,DAYLIGHTSAVINGS=TRUE)
```

Related reference:

➡ FIELD statements (System Utilities)

➡ DFSMARSH statements (System Utilities)

Defining arrays in DBD source statements

You can define to IMS the arrays that are used by your application programs by coding them in your DBD source statements.

An *array* is a data structure that contains an *element* that repeats. You define an array and its elements by using the FIELD statement of the DBD Generation utility.

An array is defined to IMS as a field that has a data type of ARRAY. The elements of an array are defined to IMS as a repeating set of fields. All of the fields in an array element specify the external name of the array field on the PARENT parameter.

The arrays that you can define to IMS can be *static arrays* or *dynamic arrays*. A static array always contains the same number of elements in every instance of a segment type. A dynamic array contains a number of elements that can vary from one instance of a segment type to another. The array type you use is determined by the requirements of your application programs.

The number of times an array element repeats within an array is determined by the MINOCCURS parameter, the MAXOCCURS parameter, and, for dynamic arrays only, a number specified in a separate control field in the segment.

For a static array, the number of array elements is specified on both the MINOCCURS parameter and the MAXOCCURS parameter. The number specified on each parameter must be the same.

For a dynamic array, the value specified on the MINOCCURS parameter defines the minimum possible number of elements that can occur in a given segment instance. The value specified on the MAXOCCURS parameter defines the maximum possible number of elements that can occur in any given segment instance. The actual number of occurrences of the element in a segment instance is determined by the value specified in the control field that is named on the DEPENDSON parameter in the FIELD statement of the array.

Fields defined as an array support only an external name. Consequently, they cannot be specified in an SSA.

Related reference:

↪ FIELD statements (System Utilities)

Defining a static array to IMS:

A static array has a fixed number of array elements in every instance of a segment type.

A static array is defined by coding FIELD statements in the input control statements of the DBD Generation utility. The following steps assume that the corresponding SEGM, DBD, and other utility control statements have been coded correctly:

1. Code the FIELD statement of the static array:
   a. Specify the external name of the array on the EXTERNALNAME parameter. Arrays do not support the NAME parameter.
   b. Specify DATATYPE=ARRAY
   c. Specify the number of elements in the array on the MINOCCURS and MAXOCCURS parameters. The value specified on both parameters must be the same.
   d. Specify the byte size of the array on the BYTES parameter. The byte size specified must be equal to or greater than the sum total of the bytes sizes of all fields that make up the array and array elements.
2. Code the FIELD statements for each field that makes up the repeating array element. In the FIELD statement for each field that is a part of the array element:
   a. Specify the external name of the field on the EXTERNALNAME parameter. Arrays do not support the NAME parameter.
   b. Specify the starting byte offset of the field in the array element on the RELSTART parameter.

c. Specify the external name of the array on the PARENT parameter.

d. Specify the byte size of the field on the BYTES parameter.

The following FIELD statements show an example of a static array definition. In the example, the first FIELD statement defines the array. The array is named COURSES and contains 8 instance of a repeated element, as specified on the MINOCCURS and MAXOCCURS parameters. The repeated array element is defined by the last three FIELD statements. Because each array element is made up of three fields that total 64 bytes and the array is made up of 8 array elements, the array requires a minimum of 512 bytes.

The array starts at byte 5 in the segment and the first field in the first array element starts at byte 1 of the array. The subsequent array elements start at bytes 65, 129, 193, and so on.

```
    FIELD EXTERNALNAME=COURSES,DATATYPE=ARRAY,START=5,                   X
             BYTES=512,MINOCCURS=8,MAXOCCURS=8
   FIELD EXTERNALNAME=COURSE_ID,RELSTART=1,BYTES=8,PARENT=COURSES
   FIELD EXTERNALNAME=COURSE_TITLE,RELSTART=9,BYTES=48,                  X
             PARENT=COURSES
   FIELD EXTERNALNAME=INSTRUCTOR_ID,RELSTART=57,BYTES=8,                 X
             PARENT=COURSES
```

The preceding array definition is based on the following excerpt of an example COBOL copy book:

```
******* COPYBOOK for STUDENT/COURSES (STATIC)01   STUDENT.
      20   COURSES OCCURS 8 TIMES.
        25   COURSE_ID   PIC 9(8).
        25   COURSE_TITLE  PIC X(48).
        25   INSTRUCTOR_ID PIC 9(8).
```

Related reference:

↪ FIELD statements (System Utilities)

Defining a dynamic array to IMS:

A dynamic array is an array in which the number of repeating array elements can vary from one instance of a segment type to another.

A dynamic array is defined by coding FIELD statements in the input control statements of the DBD Generation utility. The following steps assume that the corresponding DBD, DATASET, SEGM, and other utility control statements have been coded correctly:

1. Define the control field that will contain the number of array elements for the array in an instance of the segment.

2. Define the array by coding a FIELD statement:

   a. Specify the external name of the array on the EXTERNALNAME parameter. Arrays do not support the NAME parameter.

   b. Specify DATATYPE=ARRAY

   c. On the MINOCCURS parameter, specify the minimum possible number of elements that the array can have in a segment instance. The value specified on the MINOCCURS parameter must be less than the value specified on the MAXOCCURS parameter.

   d. On the MAXOCCURS parameter, specify the maximum possible number of elements that the array can have in a segment instance. The value specified on the MAXOCCURS parameter must be greater than the value specified on the MINOCCURS parameter.

e. Specify the maximum possible byte size of the array on the MAXBYTES parameter. The byte size specified must be equal to or greater than the sum total of the bytes sizes of all fields that make up the array with the maximum number of array elements. The MAXBYTES parameter is mutually exclusive with the BYTES parameter that is used to define the size of a static array.

f. Specify the name of the control field on the DEPENDSON parameter.

3. Define the repeating array element by coding the FIELD statements for each field that is contained in the element. In the FIELD statement for each field that is a part of the array element:

a. Specify the external name of the field on the EXTERNALNAME parameter. Array elements do not support the NAME parameter.

b. Specify the starting byte offset of the field in the array element on the RELSTART parameter.

c. Specify the external name of the array on the PARENT parameter.

d. Specify the byte size of the field on the BYTES parameter. If the field contains a dynamic array, use the MAXBYTES parameter instead.

The following example shows the FIELD statement definitions for a dynamic array.

In the example, the first FIELD statement defines the control field that will specify the number of array elements in an instance of the array.

The second FIELD statement defines the array. The array is named BOOKS and can contains 1 to 5 array elements, as specified on the MINOCCURS and MAXOCCURS parameters. The field NUMOF_BKS determines the number of array elements in an instance of the array.

The repeating array element is defined by the last three FIELD statements. Because the array element is made up of three fields that together total 40 bytes and the element can repeat up to 5 times, the array must be defined with a MAXBYTES value of at least 200 bytes.

The array starts at byte 5 in the segment and the first field in the first array element starts at byte 1 of the array. The subsequent array elements start at bytes 41, 81, and so on, of the array.

```
FIELD EXTERNALNAME=NUMOF_BKS,DATATYPE=INT,START=1,BYTES=4
FIELD EXTERNALNAME=BOOKS,DATATYPE=ARRAY,START=5,MAXBYTES=200        X
         MINOCCURS=1,MAXOCCURS=5,DEPENDSON=NUMOF_BKS
   FIELD EXTERNALNAME=ISBN,RELSTART=1,BYTES=10,PARENT=BOOKS
   FIELD EXTERNALNAME=BOOK_TITLE,RELSTART=11,BYTES=22,PARENT=BOOKS
   FIELD EXTERNALNAME=RETURN_DATE,RELSTART=33,BYTES=8,PARENT=BOOKS
```

The preceding array definition is based on the following excerpt of an example COBOL copy book:

```
******* COPYBOOK for STUDENT/COURSES (DYNAMIC)01 STUDENT.
     20   NUMOF-BKS      PIC 9(4) COMP.
     20   BOOKS OCCURS 1 TO 5 TIMES DEPENDING ON NUMOF-BKS.
     30   ISBN         PIC X(10).
     30   BOOK-TITLE    PIC X(22).
     30   RETURN-DATE  PIC 9(8).
```

Related reference:

⬛➡ FIELD statements (System Utilities)

Defining a data structure in DBD source statements

Define the data structures that are used by your application programs in your DBD source statements so that IMS stores the data in the same way that your application processes it.

A structure is a fixed set of related data elements. Unlike an array, the elements of a structure do not repeat.

A structure is defined to IMS as a field with a data type of STRUCT. The elements of a structure are fields that specify the structure as a parent.

You can define a name, an external name, or both on a structure. Defining a name allows you to specify the name of the structure field in an SSA. However, if an element of the structure is a dynamic array, you can only specify an external name and not an IMS name. Dynamic arrays and any structures that contain dynamic arrays are not searchable by IMS and therefore cannot be included in an SSA.

1. Define a structure by coding a FIELD statement that specifies the following attributes:
   a. A field name on the NAME parameter or, if the structure contains a dynamic array, the EXTERNALNAME parameter.
   b. DATATYPE=STRUCT
   c. If the structure contains a dynamic array, specify the following:
      - The maximum byte size of the structure on the MAXBYTES parameter. The maximum byte size of the structure must include the MAXBYTES value of the dynamic array and the byte sizes of all other elements contains in the structure.
2. Define each element of the structure by coding a FIELD statement that specifies the name or external name of the structure on the PARENT parameter.

These FIELD statements show an example of a structure definition. In the example, the first FIELD statement defines the structure. The structure is named ADDRESS_INFO. The elements of the structure are defined by the following three FIELD statements that all specify ADDRESS_INFO on the PARENT parameter. Because the structure contains three fields that together total 45 bytes, the structure requires a minimum of 45 bytes.

The structure and the first element of the structure start at byte 5 in the segment. The starting positions of the subsequent elements in the structure are also calculated relative to the beginning of the segment.

```
FIELD NAME=ADDRESS_INFO,DATATYPE=STRUCT,START=5,BYTES=45
FIELD NAME=CITY,DATATYPE=CHAR,START=5,BYTES=15,PARENT=ADDRESS_INFO
FIELD NAME=STREET,DATATYPE=CHAR,START=21,BYTES=25,PARENT=ADDRESS_INFO
FIELD NAME=ZIP,DATATYPE=CHAR,START=46,BYTES=5,PARENT=ADDRESS_INFO
```

The preceding array definition is based on the following excerpt of an example COBOL copy book:

```
        02 ADDRESS-INFO.
        04 CITY PIC X(15).
        04 STREET PIC X(25).
        04 ZIP PIC X(5).
```

Related reference:

➡️ FIELD statements (System Utilities)

Redefining fields

Some application programming languages, such as COBOL, allow application programs to redefine fields for various purposes. IMS is unaware of the redefinition of a field, unless you include corresponding definitions for the redefined fields in the database description (DBD).

A field can be redefined as one or more fields as long as the total length of the fields in the redefinition is equal to or less than the length of the field that is being redefined.

If you are redefining a field as multiple fields, use a structure field to contain the multiple fields of the redefinition.

You cannot redefine a field that specifies DATATYPE=ARRAY.

The following example shows a one-to-one field redefinition in DBD source statements. FLD2 redefines FLD1. FLD0 and FLD3 are not a part of the redefinition example and are included only to show the example in context.

```
FIELD NAME=FLD0,START=1,BYTES=4
FIELD NAME=FLD1,START=5,BYTES=15
FIELD NAME=FLD2,REDEFINES=FLD1,START=5,BYTES=15
FIELD NAME=FLD3,START=20,BYTES=6
```

The following is an example of a field that is redefined as multiple fields in the COBOL programming language. FLD2 redefines FLD1 as two fields, FLD3 and FLD4. FLD0 and FLD5 are not a part of the redefinition, but are included in the example only to show the example in context and help illustrate the relative starting positions of the fields in the redefinition.

```
*    01  FLD0              PIC X(4).
*    01  FLD1              PIC X(15).
*    01  FLD2 REDEFINES FLD1.
*      01 FLD3   PIC X(7).
*      01 FLD4   PIC X(8).
*    01  FLD5              PIC X(1).
```

The following example FIELD statements show the FIELD statement definitions in DBD source that correspond to the preceding copybook. In the example, FLD2 redefines FLD1 and has the same byte length as FLD1. FLD2 specifies DATATYPE=STRUCT. FLD3 and FLD4 both specify PARENT=FLD2. The sum total of the BYTES parameter for FLD3 and FLD4 equals the value of BYTES of both FLD1 and FLD2.

```
FIELD NAME=FLD0,START=1,BYTES=4
FIELD NAME=FLD1,START=5,BYTES=15
FIELD NAME=FLD2,REDEFINES=FLD1,START=5,BYTES=15,
          DATATYPE=STRUCT
  FIELD NAME=FLD3,START=5,BYTES=7,PARENT=FLD2
  FIELD NAME=FLD4,START=12,BYTES=8,PARENT=FLD2
FIELD NAME=FLD5,START=20,BYTES=1
```

Related reference:

➡ FIELD statements (System Utilities)

Defining alternative field maps for a segment

You can define multiple alternative field maps for a single sequence of bytes in a segment definition. When an instance of the segment is created in the database, the application program selects one of the fields maps to map the data and indicates the field map that is in effect by inserting the ID of the field map into a control field.

When an application program accesses a sequence of bytes in a segment instance that uses field mapping, the application program must evaluate the control field to determine which field map is in effect.

Each field map, or *map case*, in a segment definition is defined to IMS by coding a DFSCASE statement. The fields that make up a field map are defined by FIELD statements that specify the name of the map case on the CASENAME parameter.

A set of map cases that map the same sequence of bytes in a segment share the same control field. In the segment definition, a DFSMAP statement links the set of related map cases to the control field. The map case definitions must specify the name of the DFSMAP statement on the MAPNAME parameter in the DFSCASE statement.

The FIELD statements for the fields that are contained in a map must follow the DFSCASE statement that defines the map that they belong to. Each field in the map must specify the map name on the CASENAME parameter.

To define alternative field maps for a sequence of bytes in a segment definition:
1. Define the control field to contain the case ID that determines which map case is in effect in a segment instance. Case IDs can be either a character value or a hexadecimal value. The data type and byte length specified in the control field definition must accommodate the data type and lengths of the case IDs specified in the definitions of the map cases.
2. Define a DFSMAP statement that will group the collection of map cases. The external name of the control field must be specified on the DEPENDINGON parameter.
3. Define a DFSCASE statements for each map case. The length of the control field must support the length and data type of the value that you specify on the CASEID parameter.
4. Define the FIELD statements for the fields that belong to each map case. Specify the name of the map case that each field belongs to in the CASENAME parameter.

Mapping example: DFSMAP and DFSCASE

The following example shows how mapping might be used in the DBD source to define a single segment that is used to store data about three different types of insurance policy: an auto insurance policy, a home insurance policy, and a boat insurance policy. Each policy type requires different fields to hold the information that is unique to that policy type.

In the DBD source, the fields for each policy type are mapped by a different DFSCASE statement. The three map cases in the example are named AUTOMAP, HOMEMAP, and BOATMAP. The fields that make up the map defined by a given DFSCASE statement each specify the name of the DFSCASE statement that they belong to on the CASENAME parameter in their FIELD statement. The DFSCASE statements are grouped by the DFSMAP statement POLICYMAPS in the segment CUSTOMERPOLICY.

The value specified on the CASEID parameter of each map case uniquely identifies the map case and serves as the control field value. When a segment instance is first inserted into the database, the ID of the map case that the segment instance uses is inserted into the control field. In the example, the control field is named POLICYTYPE. At run time, when an application program retrieves the segment

from the database, the application program must evaluate the control field value to
determine the correct mapping of the fields.

```
     DBD       NAME=POLICYDB,                                            C
               ENCODING=CP1047,                                         C
               ACCESS=(DEDB),                                           C
               RMNAME=(RMOD3),                                          C
               PASSWD=NO
       AREA    DD1=PLCYAR01,                                            C
               DEVICE=3330,                                             C
               SIZE=(2048),                                             C
               UOW=(15,10),                                             C
               ROOT=(10,5),                                             C
               REMARKS='AREA NUMBER 1 FOR POLICYDB DATABASE'
     SEGM      NAME=CUSTOMER,                                           C
               PARENT=0,                                                C
               BYTES=(390,20)
       FIELD   NAME=(CUSTKEY,SEQ,U),                                    C
               BYTES=12,                                                C
               START=1,                                                 C
               TYPE=C
     SEGM    NAME=POLICY,                                               C
               EXTERNALNAME=CUSTOMERPOLICY,                             C
               ENCODING=CP1047,                                         C
               PARENT=CUSTOMER,                                         C
               BYTES=(900),                                             C
               TYPE=DIR,                                                C
               RULES=(LLL,HERE)
***********************************************************************
*      CONTROL FIELD:
***********************************************************************
     FIELD   EXTERNALNAME=POLICYTYPE,                                   C
             BYTES=4,                                                   C
             START=1,                                                   C
             DATATYPE=CHAR
***********************************************************************
*      DFSMAP STATEMENT:
***********************************************************************
     DFSMAP  NAME=POLICYMAPS,                                           C
             DEPENDINGON=POLICYTYPE
***********************************************************************
*      DFSCASE STATEMENT 1:
***********************************************************************
     DFSCASE NAME=AUTOMAP,                                              C
             CASEID=AUTO,                                               C
             CASEIDTYPE=C,                                              C
             MAPNAME=POLICYMAPS,                                        C
             REMARKS='DEFINES THE FIELDS OF AN AUTO INSURANCE POLICY'
       FIELD   EXTERNALNAME=AUTOMAKE,                                   C
               CASENAME=AUTOMAP,                                        C
               BYTES=15,                                                C
               START=5,                                                 C
               DATATYPE=CHAR
       FIELD   EXTERNALNAME=MODEL,                                      C
               CASENAME=AUTOMAP,                                        C
               BYTES=15,                                                C
               START=20,                                                C
               DATATYPE=CHAR
       FIELD   EXTERNALNAME=YEAR,                                       C
               CASENAME=AUTOMAP,                                        C
               BYTES=4,                                                 C
               START=35,                                                C
               DATATYPE=CHAR
***********************************************************************
*      DFSCASE STATEMENT 2:
***********************************************************************
     DFSCASE NAME=HOMEMAP,                                              C
```

```
                          CASEID=HOME,                                     C
                          CASEIDTYPE=C,                                    C
                          MAPNAME=POLICYMAPS,                              C
                          REMARKS='DEFINES THE FIELDS OF A HOME INSURANCE POLICY'
            FIELD     EXTERNALNAME=DWELLING_TYPE,                          C
                          CASENAME=HOMEMAP,                                C
                          BYTES=20,                                        C
                          START=5,                                         C
                          DATATYPE=CHAR
            FIELD     EXTERNALNAME=ROOMS,                                  C
                          CASENAME=HOMEMAP,                                C
                          BYTES=5,                                         C
                          START=25,                                        C
                          DATATYPE=CHAR
            FIELD     EXTERNALNAME=SQ_FOOT,                                C
                          CASENAME=HOMEMAP,                                C
                          BYTES=6,                                         C
                          START=30,                                        C
                          DATATYPE=CHAR
***************************************************************************
*       DFSCASE STATEMENT 3:
***************************************************************************
            DFSCASE   NAME=BOATMAP,                                       C
                          CASEID=BOAT,                                     C
                          CASEIDTYPE=C,                                    C
                          MAPNAME=POLICYMAPS,                              C
                          REMARKS='DEFINES THE FIELDS OF A BOAT INSURANCE POLICY'
            FIELD     EXTERNALNAME=CLASS,                                  C
                          CASENAME=BOATMAP,                                C
                          BYTES=10,                                        C
                          START=5,                                         C
                          DATATYPE=CHAR
            FIELD     EXTERNALNAME=LENGTH,                                 C
                          CASENAME=BOATMAP,                                C
                          BYTES=6,                                         C
                          START=15,                                        C
                          DATATYPE=CHAR
            FIELD     EXTERNALNAME=BOATMAKE,                               C
                          CASENAME=BOATMAP,                                C
                          BYTES=10,                                        C
                          START=21,                                        C
                          DATATYPE=CHAR
        DBDGEN
        FINISH
        END
```

Related concepts:

"DFSMAP statement overview" on page 524

Related reference:

➡ FIELD statements (System Utilities)

➡ DFSMAP statements (System Utilities)

➡ DFSCASE statements (System Utilities)

Implementing HALDB design

To create a HALDB database, you perform a two-stage process: first, you define
the data structures by using either the DBD Generation utility or DDL statements
and, second, you define the database and its partitions to DBRC.

You can use either DBRC batch commands or the Partition Definition utility to define HALDB databases and their partitions to DBRC. This topic discusses defining HALDB databases using the Partition Definition utility, as well as allocating an indirect list data set (ILDS).

Related concepts:

"Reorganizing HALDB databases" on page 663

Related tasks:

"Modifying HALDB databases" on page 809

Related reference:

➥ HALDB Partition Definition utility (%DFSHALDB) (Database Utilities)

Creating HALDB databases with the HALDB Partition Definition utility

The HALDB Partition Definition utility (%DFSHALDB) is an ISPF application that allows you to manage IMS HALDB partitions.

Prerequisite: Before you can use the HALDB Partition Definition utility to define the partitions of a HALDB database, you must define the HALDB master database by using the Database Description Generation (DBDGEN) utility.

Creating HALDB partitions with the Partition Definition utility

When you define the first HALDB partition, you must also register the HALDB master database in the DBRC RECON data set. You can use either the HALDB Partition Definition utility or the DBRC INIT.DB and INIT.PART commands to do this.

The HALDB Partition Definition utility does not impact RECON data set contention of online IMS subsystems. The RECON data set is reserved only for the time it takes to process a DBRC request. It is not held for the duration of the utility execution.

When defining HALDB partitions using the Partition Definition utility, you must provide information such as the partition name, data set prefix name, and high key value. Whenever possible, the Partition Definition utility provides default values for required fields.

Automatic and manual HALDB partition definition

You can choose either automatic or manual partition definition by specifying Yes or No in the Automatic Definition field in the Processing Options section of the Partition Default Information panel.

Entering Yes in the Automatic Definition field specifies that the Partition Definition utility automatically defines your HALDB partitions. You must have previously created a data set and it must contain your HALDB partition selection strings. Specify the name of the data set in the Input data set field.

Entering No in the Automatic Definition field specifies that you define your HALDB partitions manually.

You can still use an input data set when you define HALDB partitions manually.

The steps for defining a new HALDB are as follows:

1. Make the dialog data sets available to the TSO user. You can add the data sets to a LOGON procedure or use TSO commands to allocate them. You can use the TSOLIB command to add data sets to the STEPLIB. The following table shows which file names and data sets need to be allocated. Be sure to use your own high level qualifiers.

Table 73. File names and data sets to allocate

File name	Sample data set names	Disposition
STEPLIB	IMS.SDFSRESL	N/A
SYSPROC	IMS.SDFSEXEC	SHR
ISPMLIB	IMS.SDFSMLIB	SHR
ISPPLIB	IMS.SDFSPLIB	SHR
ISPTLIB	IMS.SDFSTLIB	SHR
IMS	IMS.DBDLIB	SHR

If you use a logon procedure, you must log on again and specify logon with the new procedure. If you use allocation commands, they must be issued outside of ISPF. After you allocate the data sets and restart ISPF, restart the Install/IVP dialog, return to this task description, and continue with the remaining steps.

2. Start the HALDB Partition Definition utility from the ISPF command line by issuing the following command: TSO %DFSHALDB

3. Specify the name of the database. Fill in the first partition name as shown in the following example. Fill in the data set name prefix using the data set name for your data set instead of the high level qualifier shown in the example. You should, however, specify the last qualifier as IVPDB1A to match cluster names previously allocated.

   Recommendation: When naming your partitions, use a naming sequence that allows you to add new names later without disrupting the sequence. For example, if you name your partitions xxxx010, xxxx020 and xxxx030 and then later split partition xxxx020 because it has grown too large, you can name the new partition xxxx025 without disrupting the order of your naming sequence.

```
Help
-----------------------------------------------------------------
                  Partition Default Information

Type the field values.  Then press Enter to continue.


Database Name . . . . . . . IVPDB1

                        Processing Options
Automatic Definition . . . .No
Input data set . . . . . .
Use defaults for DS groups .No

                     Defaults for Partitions
Partition Name . . . . . . .IVPDB11
Data set name prefix . . . .IXUEXEHQ.IVPDB1A

Free Space
  Free block freq. factor . 0
  Free space percentage . . 0

                     Defaults for data set groups
Block Size . . . . . . . . .8192

DBRC options
    Max. image copies . . . .2
    Recovery period . . . . .0
    Recovery utility JCL . . RECOVJCL
    Default JCL . . . . . . ._____
    Image Copy JCL . . . . . ICJCL
    Online image copy JCL . .OICJCL
    Receive JCL . . . . . . .RECVJCL
    Reusable? . . . . . . . .No

To exit the application, press F3

Command = = = >
```

Figure 236. Partition Default Information

4. Define your partitions in the Change Partition panel. Make sure that the name
   of the partition and the data set name prefix are correct and then define a high
   key for the partition.

   The high key identifies the highest root key of any record that the partition can
   contain and is represented by a hexadecimal value that you enter directly into
   the Partition High Key field of the Change Partition panel. Press F5 to accept
   the hexadecimal value and display its alphanumeric equivalent in the right
   section of the Partition High Key field.

   You can enter the partition high key value using alphanumeric characters by
   pressing F5 before making any changes in the hexadecimal section of the
   Partition High Key field. This displays the ISPF editing panel. The
   alphanumeric input you enter in the editing panel displays in both hexadecimal
   and alphanumeric formats in the Change Partition Panel when you press F3 to
   save and exit the ISPF editor.

   The last partition you define for a HALDB database should have a high key
   value of X'FF'. This ensures that the keys of all records entered into the HALDB
   database will be lower than the highest high key in the HALDB database. The
   Partition Definition utility fills all remaining bytes in the Partition High Key
   field with hexadecimal X'FF'. If the partition with the highest high key in the
   database has a key value other than X'FF's, any attempt to access or insert a
   database record with a key higher than the high key specified results in an FM
   status code for the call. Application programs written for non-HALDB
   databases are unlikely to expect this status code.

When you finish defining the partition high key, press enter to create the partition. The Change Partition panel remains active so that you can create additional partitions. To create additional partitions, you must change the partition name and the partition high key.

The following figure shows the Change Partition panel. The Partition High Key field includes sample input.

```
Help
-------------------------------------------------------------
                       Change Partition
Type the field values. Then press Enter.

Database name..........IVPDB1
Partition name.........IVPDB11
Partition ID...........1
Data set name prefix...IXUEXEHQ.IVPDB1A
Partition Status......._____


Partition High Key
  +00 57801850 00F7F4F2 40C5A585 99879985 | ...&.742 Evergre |
  +10 859540E3 85999981                    | en Terra         |

Free Space
   Free block freq. factor...0
   Free space percentage.....0

Attributes for data set group A
   Block Size...............8192

DBRC options
   Max. image copies.........2
   Recovery period...........0
   Recovery utility JCL......_____
   Image copy JCL............ICJCL
   Online image copy JCL.....OICJCL
   Receive JCL...............RECVJCL
   Reusable?.................No


Command  = = = >
```

Figure 237. Change Partition panel

5. When you finish defining partitions, press the cancel key (F12) to exit the Change Partition panel. A list of partitions defined in the current session displays.

   To exit the HALDB Partition Definition utility entirely, press F12 again.

Related concepts:

"Options for offline reorganization of HALDB databases" on page 665

"Coding database descriptions as input for the DBDGEN utility" on page 519

▷ DBRC administration (System Administration)

Related tasks:

"Specifying use of multiple data set groups in HD and PHD databases" on page 415

Related reference:

▷ Database Description (DBD) Generation utility (System Utilities)

Allocating an ILDS

The ILDS manages pointers for HALDB partitions that include either logical relationships or a secondary index.

Partitioning a database can complicate the use of pointers between database records because after a partition has been reorganized the following pointers may become invalid:

- Pointers from other database records within this partition
- Pointers from other partitions that point to this partition
- Pointers from secondary indexes

The use of indirect pointers eliminates the need to update pointers throughout other database records when a single partition is reorganized. The Indirect List data set (ILDS) acts as a repository for the indirect pointers. There is one ILDS per partition in PHDAM and PHIDAM databases.

The ILDS contains indirect list entries (ILEs). Each ILE in an ILDS has a 9-byte key that is the indirect list key (ILK) of the target segment appended with the segment code of the target segment. The ILK is a unique token that is assigned to segments when the segments are created.

After a reorganization reload or a migration reload of segments involved in inter-record pointing, the ILE is updated to reflect the changes in location of the target segment of the ILE. Segments involved in inter-record pointing can be one of the following types:

- Physically paired logical children
- Logical parents of unidirectional logical children
- Targets of secondary indexes

The following sample command defines an ILDS. Note that the key size is 9 bytes at offset 0 (zero) into the logical record. Also note that the record size is specified as 50 bytes, the current length of an ILE.

```
DEFINE CLUSTER ( -
        NAME (FFDBPRT1.XABCD010.L00001) -
        TRK(2,1) -
        VOL(IMSQAV) -
        FREESPACE(80,10) -
        REUSE -
        SHAREOPTIONS(3,3) -
        SPEED ) -
      DATA ( -
        NAME(FFDBPRT1.XABCD010.INDEXD) -
        CISZ(8192) -
        KEYS(9,0) -
        RECSZ(50,50) ) -
      INDEX ( -
        NAME(FFDBPRT1.XABCD010.INDEXS) -
        CISZ(2048) )
```

To compute the size of an ILDS, multiply the size of an ILE by the total number of physically paired logical children, logical parents of unidirectional relationships, and secondary index targets.

The inclusion of free space in the ILDS can improve the performance of ILDS processing by reducing CI and CA splits. Both the HD Reorganization Reload utility (DFSURGL0) and the HALDB Index/ILDS Rebuild utility (DFSPREC0)

provide a free space option that uses the VSAM load mode to update or rebuild the ILDS. VSAM load mode adds the free space that is called for by the FREESPACE parameter of the DEFINE CLUSTER command.

Related concepts:

"The HALDB self-healing pointer process" on page 691

⇨ Defining a HALDB indirect list data set (System Definition)

Related reference:

⇨ HALDB Index/ILDS Rebuild utility (DFSPREC0) (Database Utilities)

⇨ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

Defining generated program specification blocks for SQL applications

Generated PSBs (GPSB) are a type of PSB that do not require a PSBGEN or ACBGEN.

A GPSB contains an I/O PCB and a single modifiable alternate PCB. GPSBs are not defined through a PSBGEN. Instead, they are defined by the system definition process through the APPLCTN macro. The GPSB parameter indicates the use of a generated PSB and specifies the name to be associated with it. The LANG parameter specifies the language format of the GPSB. For more information on defining GPSBs refer to the APPLCTN macro topic in *IMS Version 15 System Definition*.

The I/O PCB can be used by the application program to obtain input messages and send output to the inputting terminal. The alternate PCB can be used by the application program to send output to other terminals or programs.

Other than the I/O PCB, an application that makes only Structured Query Language (SQL) calls does not require any PCBs. It does, however, need to define the application program name and language type to IMS. A GPSB can be used for this purpose.

Introducing databases into online systems

Before online IMS systems can recognize databases, the appropriate database control blocks must be created in the online systems.

Online IMS systems require two types of control blocks for databases, application control blocks (ACBs) and control block that store certain runtime attributes. The ACBs for databases are added to an online system by either the online change process or, if the IMS management of ACBs is enabled, the IMPORT DEFN SOURCE(CATALOG) command. For more information about adding ACBs to online systems see, Making online changes (System Administration).

Database runtime attribute controls blocks are created in online systems either by including a DATABASE macro for each online database in the stage 1 system definition or, in systems that have dynamic resource definition enabled, by issuing the type-2 command CREATE DB.

Databases can also be introduced into online systems that have dynamic resource definition enabled by importing the database definition from the resource definition data set (RDDS) or the IMSRSC repository by using the type-2 command IMPORT DEFN.

If IMS is using the IMSRSC repository, databases can be created or updated from the IMS change list in the repository at the end of IMS warm or emergency restart.

The DATABASE system definition macro, the CREATE DB command, and the IMPORT DEFN command all create the database directory (DDIR) control block that is required by IMS to manage databases in the online environment.

To save across cold starts the definitions of databases that are added to online systems by either the CREATE DB or the IMPORT DEFN commands, the database definitions must be either exported to the RDDS or the repository or added to the IMS MODBLKS data set by system definition and then imported during cold start. Across warm starts and emergency restarts, IMS recovers dynamic resource definition changes from the logs.

Database definitions can be exported to an RDDS or the repository by using the EXPORT DEFN command. In addition, IMS can be configured to export the definitions automatically to an RDDS or the repository during system checkpoints. You can enable automatic export by specifying AUTOEXPORT=AUTO, AUTOEXPORT=REPO, or AUTOEXPORT=RDDS in the DYNAMIC_RESOURCES section of the DFSDFxxx member in the IMS PROCLIB data set.

Related concepts:

Resource lists for the IMSRSC repository (System Definition)

Overview of the IMSRSC repository (System Definition)

Related tasks:

"Changing databases dynamically in online systems" on page 791

"Modifying online databases" on page 757

Related reference:

DYNAMIC_RESOURCES section of the DFSDFxxx member (System Definition)

Adding databases dynamically to an online IMS system

In IMS systems that have dynamic resource definition enabled, you can add a database to the online system by issuing the `CREATE DB` command.

The function of the `CREATE DB` command corresponds to the function of the DATABASE system definition macro. Both declare databases to an online IMS system by creating a database directory (DDIR) control block in the IMS control region.

The `CREATE DB` command defines all of the database attributes that can be defined by using the DATABASE system definition macro. The attributes include:

- The recognition of the database by the online IMS system
- The access type
- The resident status of the database

You can issue the `CREATE DB` command either before or after you define the DBD, PSB, and ACBs for the database; however, if you issue the CREATE DB command after the ACBGEN process, IMS can take additional action based on the type of database defined.

If the ACBGEN process is complete and the DMB for the database is in the ACB library, IMS performs the following additional actions when the `CREATE DB` command is issued:

- For full-function databases, IMS loads the DMB for the database into the DMB pool at first schedule and if the resident option is selected, IMS makes the DMB for the database resident at the next restart of IMS.

- For DEDB databases, IMS chains the DMB into the DEDB DMCB chain.

  IMS also checks to see if the CI size for any area in the DMCB is larger than the size defined for the Fast Path global buffer pool. If a CI size is too large, the `CREATE DB` command fails with condition code E3.

  Also, if any DEDB area name already exists in the Fast Path area list (FPAL), the `CREATE DB` command fails with condition code E4.

If a DMB for the database is not already in the ACB library when a `CREATE DB` command is issued, IMS still adds the database to the online system, but the database has a status of NOTINIT. Before the database can be used, a DMB for the database must be added to ACB library and started. Unless you are adding an MSDB database, you can use the online change function for ACB library members to add the DMB.

In addition to using the CREATE DB command, a database can also be added to the online system by either of the following methods:

- Importing the stored resource definitions from the resource definition data set (RDDS) with the automatic import function or the IMPORT command.

- Importing the stored resource definitions from the IMSRSC repository with the automatic import function or the IMPORT DEFN command.

Adding MSDB databases dynamically to an online IMS system

The procedure for adding an MSDB database to an online IMS system by using dynamic resource definition is different than the procedure for adding other database types.

1. Perform the DBDGEN and ACBGEN process for the MSDB database.
2. Copy the ACB staging library to the inactive ACB library.
3. Execute a full ACBLIB online change. The member-level online change function for the ACB library does not support MSDBs.
4. Insert segments into the MSDBINIT data set by using the MSDB Maintenance Utility (DBFDBMA0).
5. Issue the `CREATE DB` command for the MSDB database.

Note that the MSDB database cannot be used until it is loaded into online storage.

To load the MSDB database:

1. Shut down IMS.
2. Warm start IMS with MSDBLOAD keyword. For example, issue the command `/NRE MSDBLOAD` .

Provision a Fast Path DEDB database with z/OSMF

You can provision an IMS Fast Path DEDB database by using the IBM z/OS Management Facility workflow that IMS provides via GitHub.

With the provisioning workflow, IMS also provides a workflow to deprovision a DEDB database.

A workflow consists of several XML files and a properties file.

1. From the IMS-zOSMF-Workflows repository in GitHub, download the compressed folder (.zip file) that contains the workflows you need and extract the contents to a local directory.
2. Review the readme that is included in the .zip file for details about the workflows.
3. Transfer the extracted files to a data set or directory on z/OS where they are available to your z/OSMF instance.
4. From the z/OSMF welcome page, select **Workflows** from the navigation pane on the left.
5. Select **Create a workflow** and specify the location of the workflow file and other initial information, such as your user name.
6. After the steps of the workflow are displayed in the z/OSMF interface, you can begin completing each step by providing the information that each step requires. Some steps require you to begin by expanding the step and starting with the substeps.

Related concepts:

▶ z/OS: IBM z/OS Management Facility

Related reference:

▶ IMS-zOSMF-Workflows

Chapter 25. Developing test databases

Before the application programs accessing your database are transferred to production status, they must be tested. To avoid damaging a production database, you need a test database.

IBM provides various programs that can help you develop your test database, including the DL/I Test Program (DFSDDLT0). For more information on the available IMS tools, go to www.ibm.com/ims and link to the IBM® DB2 and IMS Tools website.

Related concepts:

"Who attends code inspection 1" on page 32

"Design review 4" on page 31

➥ Testing an IMS application program (Application Programming)

➥ Testing the system (System Administration)

Test requirements

Depending on your system configuration, user requirements, and the design characteristics of your database and data communication systems, you should test DL/I call sequences, application decision paths, and performance.

Test for the following:
- That DL/I call sequences execute and the results are correct.
  - This kind of test often requires only a few records, and you can use the DL/I Test Program, DFSDDLT0, to produce these records.
  - If this is part of a unit test, consider extracting records from your existing database. To extract the necessary records, you can use programs such as the IMS DataRefresher™.
- That calls execute through all possible application decision paths.
  - You might need to approximate your production database. To do this, you can use programs such as the IMS DataRefresher and other IMS tools.
- How performance compares with that of a model, for system test or regression tests, for example.
  - For this kind of test, you might need a copy of a subset of the production database. You can use IMS tools to help you.

To test for these capabilities, you need a test database that approximates, as closely as possible, the production database. To design such a test database, you should understand the requirements of the database, the sample data, and the application programs.

To protect your production databases, consider providing the test JCL procedures to those who test application programs. Providing the test JCL helps ensure that the correct libraries are used.

What kind of database?

Often, the test database can be a copy of a subset of the production database, or in some other way, a replica of it. If you have designed the production database, you should have firsthand knowledge of this requirement. Your DBDs for the production database can provide the details. If you have your production database defined in a data dictionary, that definition gives you much of the information you need. The topics in this chapter describe some aids available to help you design and generate your test database.

What kind of sample data?

It is important for the sample data to approximate the real data, because you must test that the system processes data with the same characteristics, such as the range of field values. The kind of sample data needed depends on whether you are testing calls or program logic.

- To test calls, you need values in only those fields that are sequence fields or which are referenced in SSAs.
- To test program logic, you need data in all fields accessed in the program logic such as adds or compares.

Again, you might use a copy of a subset of the real database. However, first determine which fields contain sensitive data and therefore must use fictitious data in the test database.

What kind of application program?

In order to design a test database that effectively tests the operational application programs being developed, you should know certain things about those programs. Much of the information you need is in the application program design documentation, the descriptors such as the PSBs, your project test plan, and in the Data Dictionary.

Disabling DBRC security for the RECON data set in test environments

If you copy the RECON data set from your production environment into your test environment, you can disable DBRC security for the test copy of the RECON data set.

Disabling DBRC security, which secures the RECON data set against the unauthorized use of DBRC commands and API requests, can make working with copies of the production RECON data sets in the test environment easier and can also be useful during problem determination.

Disabling DBRC security does not disable or otherwise affect the security checking performed by other security products, such as RACF.

To disable DBRC security checking for a copy of a RECON data set:

1. Specify a 1- to 44-character substring of the data set name of the secured RECON data set in the *rcnqual* parameter of the CMDAUTH keyword on either an INIT.RECON or CHANGE.RECON DBRC command. Because security checking is still active at this point, you need proper security authorization to issue the INIT.RECON or CHANGE.RECON command.
2. Define a new data set with a name that does not contain the character string specified in the *rcnqual* parameter.

3. Copy the secured RECON data set into the new data set. When the text string in *rcnqual* parameter does not match any portion of the RECON data set name, security checking for the copy of the RECON data set is disabled.

Examples

The following commands provide examples of specifying the *rcnqual* parameter.
* CHANGE.RECON CMDAUTH(SAF,SAFHLQ1,IMSTESTS.DSHR)
* INIT.RECON CMDAUTH(SAF,SAFHLQ1,IMSTESTS.DSHR)

The example below shows the status listing for a RECON data set in which security has been disabled. The string shown in the RCNQUAL field, IMSTESTS.DSHR, which could have been set by either one of the command examples above, does not match exactly any part of the name of the COPY1 RECON data set, IMSTESTS.COPYDSHR.RECON1.

```
RECON  RECOVERY CONTROL DATA SET, IMS V15R1
  DMB#=7                              INIT TOKEN=15122F2233528F
  NOFORCER  LOG DSN CHECK=CHECK17    STARTNEW=NO
  TAPE UNIT=3480     DASD UNIT=SYSDA    TRACEOFF    SSID=**NULL**
  LIST DLOG=NO                 CA/IC/LOG DATA SETS CATALOGED=NO
  MINIMUM VERSION=13.1       CROSS DBRC SERVICE LEVEL ID= 00001

  REORG NUMBER VERIFICATION=NO
  LOG RETENTION PERIOD=00.001 00:00:00.0
  COMMAND AUTH=SAF  HLQ=SAFHLQ1
  RCNQUAL = IMSTESTS.DSHR
  ACCESS=SERIAL      LIST=STATIC
  SIZALERT DSNUM=15      VOLNUM=16     PERCENT= 95
  LOGALERT DSNUM=3       VOLNUM=16

  TIME STAMP INFORMATION:

    TIMEZIN = %SYS

    OUTPUT FORMAT:  DEFAULT = LOCORG NONE    PUNC YY
                    CURRENT = LOCORG NONE    PUNC YY


  IMSPLEX = ** NONE **    GROUP ID = ** NONE **

-DDNAME-       -STATUS-       -DATA SET NAME-
  RECON1         COPY1          IMSTESTS.COPYDSHR.RECON1
  RECON2         COPY2          IMSTESTS.COPYDSHR.RECON2
  RECON3         SPARE          IMSTESTS.COPYDSHR.RECON3

  NUMBER OF REGISTERED DATABASES =       7
```

The example below shows the status listing for a RECON data set in which security is active. In this example, the string shown in the RCNQUAL field, IMSTESTS.DSHR, does match a part of the name of the COPY1 RECON data set, IMSTESTS.DSHR.RECON1.

```
RECON  RECOVERY CONTROL DATA SET, IMS V15R1
  DMB#=7                              INIT TOKEN=15122F2233528F
  NOFORCER  LOG DSN CHECK=CHECK17    STARTNEW=NO
  TAPE UNIT=3480     DASD UNIT=SYSDA    TRACEOFF    SSID=**NULL**
  LIST DLOG=NO                 CA/IC/LOG DATA SETS CATALOGED=NO
  MINIMUM VERSION=13.1       CROSS DBRC SERVICE LEVEL ID= 00001

  REORG NUMBER VERIFICATION=NO
  LOG RETENTION PERIOD=00.001 00:00:00.0
  COMMAND AUTH=SAF  HLQ=SAFHLQ1
  RCNQUAL = IMSTESTS.DSHR
```

```
         ACCESS=SERIAL       LIST=STATIC
         SIZALERT DSNUM=15       VOLNUM=16      PERCENT= 95
         LOGALERT DSNUM=3        VOLNUM=16

         TIME STAMP INFORMATION:

           TIMEZIN = %SYS

           OUTPUT FORMAT:  DEFAULT = LOCORG NONE    PUNC YY
                           CURRENT = LOCORG NONE    PUNC YY


         IMSPLEX = ** NONE **    GROUP ID = ** NONE **

         -DDNAME-        -STATUS-        -DATA SET NAME-
         RECON1          COPY1            IMSTESTS.DSHR.RECON1
         RECON2          COPY2            IMSTESTS.DSHR.RECON2
         RECON3          SPARE            IMSTESTS.DSHR.RECON3

         NUMBER OF REGISTERED DATABASES =        7
```

Designing, creating, and loading a test database

You can develop a test database just as you would develop a production database.

For example, you perform the tasks described throughout the database administration information, keeping in mind the special requirements for test databases. If your installation has testing standards and procedures, you should follow them in developing a test database.

Using testing standards

Testing standards and procedures help you avoid the same kinds of problems for test database development as your IMS development standards do for production databases.

Some of the subjects that might be included in your test system standards and that affect test database design are:

- Objectives of your test system
  - What you test for and at what development stages do you test for it
  - The kinds of testing—offline, online, integrated DB/DC or isolated
- Description of the test organization and definition of responsibilities of each group
- Relationship of test and production modes of operation
- How your test system development process deals with:
  - DB/TM structures
  - Development tools
  - DB/TM features
  - Backup and recovery

Related concepts:

➡ Testing the system (System Administration)

➡ Testing a CICS application program (Application Programming)

➡ Testing an IMS application program (Application Programming)

Using IBM programs to develop a test database

If you use the same development aids to develop the test database that you use to develop your production databases, you will benefit from using familiar tools.

Also, you will avoid problems caused by differences between test and production databases.

IMS Application Development Facility II

The IMS Application Development Facility II (IMSADF II) is designed to increase the productivity of programmers developing applications for IMS systems.

IMSADF II provides ISPF-based dialogs to work with IMS and Db2 for z/OS to reduce the time and effort that is required to develop, maintain, and expand IMS database and data communications applications.

If your installation uses IMSADF II to develop application programs, you can use it to create a simple test database. The interactive nature of the IMSADF II enables you to dynamically add segments to a database. By means of SEGM and FIELD statements, you can define a test database and update it as needed.

Related Reading: For information on how to use the IMS Application Development Facility II, see the *IMS Application Development Facility (ADF) II User's Guide*.

File Manager for z/OS for IMS Data

The IMS component of File Manager (FM/IMS) is an ISPF application with which you can manipulate data stored in IMS databases.

FM/IMS provides you with a number of flexible ways to connect to your IMS databases. For example, with BMP mode you can connect to an online multi-user database and manipulate the data. In DLI mode, you can work with data offline as a single user or you can share the data with others.

In addition, FM/IMS provides two functions that you can use in batch jobs. FM/IMS Edit Batch (IEB) runs a REXX procedure that can insert, update, retrieve, delete or print segments and create views. FM/IMS Batch Print (IPR) can print the entire database in one of several available display formats, or a selected subset of the database, based on a view.

Using the DL/I test program, DFSDDLT0

You can, in some cases, use the DL/I test program (DFSDDLT0) to test DL/I call sequences or insert segments.

For example, if you need a test database with relatively few database records, you can use DFSDDLT0 to test DL/I call sequences. If you have no machine-readable database to begin with, you can define a PCB, then use DFSDDLT0 to insert segments. This step eliminates the need for a load program to generate your test database.

The DL/I Test Program cannot be used by CICS, but can be used for stand-alone batch programs. If used for stand-alone batch programs, it is useful to interpret the database performance as it might be implemented for online or shared database programs.

Related concepts:

➡ Testing DL/I call sequences (DFSDDLT0) before testing your IMS program (Application Programming)

Part 5. Database administrative tasks

This section discusses the primary tasks associated with administering IMS databases, including loading databases, backing up and recovering databases, monitoring databases, tuning databases, and modifying databases.

Chapter 26. Loading databases

After you implement your database design, you are ready to write and load your database. However, before writing a load program, you must estimate the minimum size of the database and allocate data sets.

Related concepts:

"Who attends code inspection 1" on page 32

Related tasks:

"Adjusting VSAM options specified in the Access Method Services DEFINE CLUSTER command" on page 710

Estimating the minimum size of the database

When you estimate the size of your database, you estimate how much space you need to initially load your data.

Unless you do not plan to insert segments into your database after it is loaded, allocate more space for your database than you actually estimate for the initial load.

This topic contains the step-by-step procedure for estimating minimum database space. To estimate the minimum size needed for your database, you must already have made certain design decisions about the physical implementation of your database. Because these decisions are different for each DL/I access method, they are discussed under the appropriate access method in step 3 of the procedure.

If you plan to reorganize your HALDB partitions online, consider the extra space reorganization requires. Although online reorganization does not need any additional space when you first load a HALDB partition, the process does require additional space at the time of reorganization.

Related concepts:

Chapter 32, "Converting database types," on page 841

"Choosing a logical record length for a HISAM database" on page 435

"HALDB online reorganization" on page 669

"Adding logical relationships" on page 514

Related tasks:

"Deleting segment types" on page 723

"Changing the amount of space allocated" on page 711

"Changing segment size" on page 724

"Changing the number of data set groups" on page 749

"Adjusting HDAM and PHDAM options" on page 700

"Adjusting VSAM options specified in the Access Method Services DEFINE CLUSTER command" on page 710

"Unloading and reloading using the reorganization utilities" on page 722

Step 1. Calculate the size of an average database record

Determine the size, then the average number of occurrences of each segment type in a database record. By multiplying these two numbers together, you get the size of an average database record.

Determining segment size

Segment size here is physical segment size, and it includes both the prefix and data portion of the segment.

You define the size of the data portion. It can include unused space for future use. The size of the data portion of the segment is the number you specified in the BYTES= operand in the SEGM statement in the DBD or, if you are using DDL, the MAXBYTES parameter in the table options on the CREATE or ALTER TABLE statement.

The prefix portion of the segment depends on the segment type and on the options you are using. The following table helps you determine, by segment type, the size of the prefix. Using the chart, add up the number of bytes required for necessary prefix information and for extra fields and pointers generated in the prefix for the options you have chosen. Segments can have more than one 4-byte pointer in their prefix. You need to factor all extra pointers of this type into your calculations.

Table 74. Required fields and pointers in a segment's prefix

Type of segment	Fields and pointers used in the segment's prefix	Size of the field or pointer (in bytes)
All types	Segment code (not present in a SHSAM, SHISAM, GSAM, or secondary index pointer segment)	1
	Delete byte (not present in a SHSAM, SHISAM, or GSAM segment)	1
HDAM, PHDAM, HIDAM, and PHIDAM	PCF pointer	4
	PCL pointer	4
	PP pointer	4
	PTF pointer	4
	PTB pointer	4
HDAM and HIDAM only	HF pointer	4
	HB pointer	4
DEDB	PCF pointer	4
	PCL pointer	4
	Subset pointer	4
Logical parent (for HDAM and HIDAM)	LCF pointer	4
	LCL pointer	4
	Logical child counter	4
Logical parent (for PHDAM and PHIDAM)	Logical child counter (only present for unidirectional logical parents)	4
Logical child	LTF pointer	4
	LTB pointer	4
	LP pointer	4

Table 74. Required fields and pointers in a segment's prefix (continued)

Type of segment	Fields and pointers used in the segment's prefix	Size of the field or pointer (in bytes)
Logical child (PHDAM and PHIDAM)	EPS	28
Secondary index	Symbolic or direct-address pointer to the target segment	4
PSINDEX	EPS plus the target segment root key	28 + length of the target-segment root key
All segments in PHDAM and PHIDAM	ILK	8

Related concepts:

"Mixing pointers" on page 157

Determining segment frequency

After you have determined the total size of a segment type, you need to determine segment frequency.

Segment frequency is the average number of occurrences of a particular segment type in the database record. To determine segment frequency, first determine the average number of times a segment occurs under its immediate physical parent.

For example, in the database record in the following figure, the ITEMS segment occurs an average of 10 times for each DEPOSITS segment. The DEPOSITS segment occurs an average of four times for each CUSTOMER root segment. The frequency of a root segment is always one.
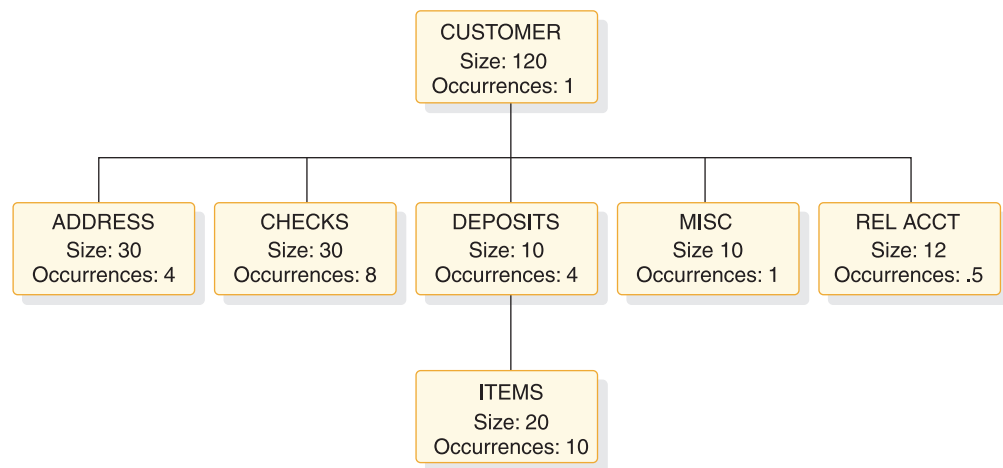
Figure 238. Segment sizes and average segment occurrences

To determine the average number of occurrences of a particular segment type in the database record, multiply together the segment frequencies of each segment in the path from the given segment back up to the root. For the ITEMS segment type, the path includes the ITEMS segment and the DEPOSITS segment. The segment frequency of ITEMS is 10, and the segment frequency of DEPOSITS is 4. Therefore, the average number of occurrences of the ITEMS segment in the database record is

40 (10 x 4). Another way of expressing this idea is that each customer has an average of 4 DEPOSITS, and each DEPOSIT has an average of 10 ITEMS. Therefore, for each customer, an average of 40 (10 x 4) ITEMS exist in the database record.

Determining average database record size

After you determine segment size and segment frequency, you can determine the average size of a database record.

To determine average database record size for a HISAM database, multiply segment size and segment frequency together for each segment type in the database record, then add the results. For example, for the database record shown in "Determining segment frequency" on page 561, the average database record size is calculated as shown in the following table.

Table 75. Calculating the average database record size

Segment type	Segment size	Average occurrences	Total size
CUSTOMER	120	1	120
ADDRESS	30	4	120
CHECKS	30	8	240
DEPOSITS	10	4	40
ITEMS	20	40 (10x4)	800
MISC	10	1	10
REL ACCT	12	.5	6
Record Total			**1336**

Step 2. Determine overhead needed for CI resources

If you are using VSAM, you need to determine how much overhead is needed for a CI before you can do the remaining space calculations.

If you are not using VSAM, you can skip this step.

Overhead is space used in a CI for two control fields. VSAM uses the control fields to manage space in the CI. The control fields and their sizes are shown in the following table.

Table 76. VSAM control fields

Field	Size in bytes
CIDF (Control interval definition field)	4
RDF (Record definition field)	3

If one logical record exists for each CI, CI overhead consists of one CIDF and one RDF (for a total of 7 bytes). HDAM and HIDAM databases and PHDAM and PHIDAM partitions use one logical record for each CI.

If more than one logical record exists for each CI, CI overhead consists of one CIDF and two RDFs (for a total of 10 bytes). HISAM (KSDS and ESDS), HIDAM and PHIDAM index, and secondary index databases can all use more than one logical record for each CI.

Step 3 tells you when to factor CI overhead into your space calculations.

Step 3. Determine the number of CIs or blocks needed

The calculations in this step are done by database type.

To determine how many CIs or blocks are needed to hold your database records, go to the topic in this step that applies to the database type you are using. If you are using VSAM, the first CI in the database is reserved for VSAM.

HISAM: determining the number of CIs or blocks needed

A CI in HISAM can contain one or more logical records. In the primary data set a logical record can only contain one database record (or part of one database record). In the overflow data set a logical record can only contain segments of the same database record, but more than one logical record can be used for the overflow segments of a single database record.

In HISAM, you should remember how logical records work, because you need to factor logical record overhead into your calculations before you can determine how many CIs (control intervals) are needed to hold your database records. Logical record overhead is a combination of the overhead that is always required for a logical record and the overhead that exists because of the way in which database records are stored in logical records (that is, storage of segments almost always results in residual or unused space).

Because some overhead is associated with each logical record, you need to calculate the amount of space that is available after factoring in logical record overhead. Once you know the amount of space in a logical record available for data, you can determine how many logical records are needed to hold your database records. If you know how many logical records are required, you can determine how many CIs or blocks are needed.

For example, assume you need to load 500 database records using VSAM, and to use a CI size of 2048 bytes for both the KSDS and ESDS. Also, assume you need to store four logical records in each KSDS CI and two logical records in each ESDS CI.

1. First factor in CI overhead by subtracting the overhead from the CI size: 2048 - 10 = 2038 bytes for both the KSDS and the ESDS. The 10 bytes of overhead consists of a 4-byte CIDF and two 3-byte RDFs.
2. Then, calculate logical record size by dividing the available CI space by the number of logical records per CI: 2038/4 = 509 bytes for the KSDS and 2038/2 = 1019 bytes for the ESDS. Because logical record size in HISAM must be an even value, use 508 bytes for the KSDS and 1018 bytes for the ESDS.
3. Finally, factor in logical record overhead by subtracting the overhead from logical record size: 508 - 5 = 503 bytes for the KSDS and 1018 - 5 bytes for the ESDS. HISAM logical record overhead consists of 5 bytes for VSAM (a 4-byte RBA pointer for chaining logical records and a 1-byte end-of-data indicator).

   This means if you specify a logical record size of 508 bytes for the KSDS, you have 503 bytes available in it for storing data. If you specify a logical record size of 1018 bytes for the ESDS, you have 1013 bytes available in it for storing data.

Refer to the previous example. Because the average size of a database record is 1336 bytes, the space available for data in the KSDS is not large enough to contain it. It takes the available space in one KSDS logical record plus one ESDS logical record to hold the average database record (503 + 1013 = 1516 bytes of available

space). This record size is greater than an average database record of 1336 bytes. Because you need to load 500 database records, you need 500 logical records in both the KSDS and ESDS.

- To store four logical records per CI in the KSDS, you need a minimum of 500/4 = 125 CIs of 2048 bytes each for the KSDS.
- To store two logical records per CI in the ESDS, you need a minimum of 500/2 = 250 CIs of 2048 bytes each for the ESDS.

HIDAM or PHIDAM: determining the number of CIs or blocks needed

With HIDAM or PHIDAM, one VSAM logical record exists per CI or block. In this context, logical record is the unit of transfer when invoking an access method (such as VSAM), to get or put records. Logical record overhead consists of an FSEAP (4 bytes). If you are using RAPs (HIDAM only), the logical record overhead consists of one RAP (4 bytes). For example, assume you need to load 500 database records using VSAM and to use a CI size of 2048 bytes and no RAP (specify PTR=TB on the root to suppress the RAP for HIDAM).

1. First, determine the size of a logical record by subtracting CI overhead from CI size: 2048 - 7 = 2041 bytes for the ESDS logical record size. The 7 bytes of overhead consists of a 4-byte CIDF and a 3-byte RDF.
2. Then, determine the amount of logical record space available for data by factoring in logical record overhead. In this example, logical record overhead consists of an FSEAP: 2041 - 4 = 2037 bytes. This means you have 2037 bytes available to store data in each logical record.

HIDAM or PHIDAM index: calculating the space needed

Calculating space for a HIDAM or PHIDAM index is similar to calculating space for a HISAM KSDS. The difference is that no logical record overhead exists. One index record is stored in one logical record, and multiple logical records can be stored in one CI or block.

HDAM or PHDAM: determining the amount of space needed

Because of the many variables in HDAM or PHDAM, no exact formula exists for estimating database space requirements. Therefore, you should use a space calculation aid to help determine the amount of space needed for HDAM or PHDAM databases.

If you are using VSAM, and you decide to estimate, without use of an aid, the amount of space to allocate for the database, the first CI in the database is reserved for VSAM. Because of this, the bitmap is in the second CI.

With HDAM or PHDAM, logical record overhead depends on the database design options you have selected. You must choose the number of CIs or blocks in the root addressable area and the number of RAPS for each CI or block. These choices are based on your knowledge of the database.

A perfect randomizer requires as many RAPs as there are database records. Because a perfect randomizer does not exist, plan for approximately 20% more RAPs than you have database records. The extra RAPs reduces the likelihood of synonym chains. For example, assume you need to store 500 database records. Then, for the root addressable area, if you use:

- One RAP per CI or block, you need 600 CIs or blocks

- Two RAPs per CI or block, you need 300 CIs or blocks
- Three RAPs per CI or block, you need 200 CIs or blocks

Because of the way your randomizer works, you decide 300 CIs or blocks with two RAPs each works best. Assume you need to store 500 database records using VSAM, and you have chosen to use 300 CIs in the root addressable area and two RAPs for each CI. This decision influences your choice of CI size. Because you are using two RAPs per CI, you expect two database records to be stored in each CI. You know that a 2048-byte CI is not large enough to hold two database records (2 x 1336 = 2672 bytes). And you know that a 3072-byte CI is too large for two database records of average size. Therefore, you would probably use 2048-byte CIs and the byte limit count to ensure that on average you would store two database records in the CI.

To determine the byte limit count:

1. First, determine the size of a logical record by subtracting CI overhead from CI size: 2048 - 7 = 2041 bytes for the ESDS logical record size.
2. Then, determine the amount of logical record space available for data by factoring in logical record overhead. (Remember only one logical record exists per CI in HDAM or PHDAM.) In this example, logical record overhead consists of a 4-byte FSEAP and two 4-byte RAPs: 2041 - 4 - (2 x 4) = 2029 bytes. This means you have 2029 bytes available for storing data in each logical record in the root addressable area.
3. Finally, determine the available space per RAP by dividing the available logical record space by the number of RAPs per CI: 2029/2 = 1014 bytes. Therefore, you must use a byte limit count of about 1000 bytes.

Continuing our example, you know you need 300 CIs of 2048 bytes each in the root addressable area. Now you need to calculate how many CIs you need in the overflow area. To do this:

- Determine the average number of bytes that will not fit in the root addressable area. Assume a byte limit count of 1000 bytes. Subtract the byte limit count from the average database record size: 1336 - 1000 = 336 bytes. Multiply the average number of overflow bytes by the number of database records: 500 x 336 = 168000 bytes needed in the non-root addressable area.
- Determine the number of CIs needed in the non-root addressable area by dividing the number of overflow bytes by the bytes in a CI available for data. Determine the bytes in a CI available for data by subtracting CI and logical record overhead from CI size: 2048 - 7 - 4 = 2037 (7 bytes of CI overhead and 4 bytes for the FSEAP). Overflow bytes divided by CI data bytes is 168000/2037 = 83 CIs for the overflow area.

You have estimated you need a minimum of 300 CIs in the root addressable area and a minimum of 83 CIs in the non-root addressable area.

Secondary index: determining the amount of space needed

Calculating space for a secondary index is similar to calculating space for a HISAM KSDS. The difference is that no logical record overhead exists in which factor.

One index record is stored in one logical record, and multiple logical records can be stored in one CI or block.

Step 4. Determine the number of blocks or CIs needed for free space

In HDAM, HIDAM, PHDAM, and PHIDAM databases, you can allocate free space when your database is initially loaded.

Free space can only be allocated for an HD VSAM ESDS or OSAM data set. Do not confuse the free space discussed here with the free space you can allocate for a VSAM KSDS using the DEFINE CLUSTER command.

To calculate the total number of CIs or blocks you need to allocate in the database, you can use the following formula:

`A = B x (fbff / (fbff - 1)) x (100 / (100 - fspf))`

Where the values are:

A   The total number of CIs or blocks needed including free space.

B   The number of blocks or CIs in your database.

fbff   How often you are leaving a block or CI in the database empty for free space.

For an HDAM or HIDAM database, this is the *fbff* value that is specified on the FRSPC= parameter of the DBD.
For a PHDAM or PHIDAM database, this is the value that is specified either on the FBFF= parameter of the INIT.PART or CHANGE.PART DBRC command or in the `Free block freq. factor` field of the Partition Default Information panel of the HALDB Partition Definition utility (%DFSHALDB).
If you are using DDL, this is the value that is specified on the FREEBLOCK parameter of the CREATE or ALTER TABLESPACE statement.

fspf   The minimum percentage of each block or CI you are leaving as free space.

For an HDAM or HIDAM database, this is the *fspf* value that is specified on the FRSPC= parameter of the DBD.
For a PHDAM or PHIDAM database, this is the value that is specified either on the FSPF= parameter of the INIT.PART or CHANGE.PART DBRC command or in the `Free space percentage` field of the Partition Default Information panel of the HALDB Partition Definition utility (%DFSHALDB).
If you are using DDL, this is the value that is specified on the FREESPACE parameter of the CREATE or ALTER TABLESPACE statement.

Related tasks:

"Specifying free space (HDAM, PHDAM, HIDAM, and PHIDAM only)" on page 431

Step 5. Determine the amount of space needed for bitmaps

In HDAM, HIDAM, PHDAM, and PHIDAM databases, you need to add the amount of space required for bitmaps to your calculations.

To calculate the number of bytes needed for bitmaps in your database, you can use the following formula:

`A = D / ((B - C) x 8)`

Where the values are:

A   The number of bitmap blocks or CIs you need for the database.

B   The CI or block size you have specified, in bytes, minus 4.

Four is subtracted from the CI or block size because each CI or block has a 4-byte FSEAP.

C   The number of RAPs you specified for a CI or block, times 4.

The number of RAPs is multiplied by 4 because each RAP is four bytes long. (B - C) is multiplied by 8 in the formula to arrive at the total number of bits that will be available in the CI or block for the bitmap.

D   The number of CIs or blocks in your database.

You need to add the number of CIs or blocks needed for bitmaps to your space calculations.

Related concepts:

"General format of HD databases and use of special fields" on page 160

Allocating database data sets

After you have determined how much space you will need for your database, you can allocate data sets and then load your database.

You can allocate VSAM data sets by using the DEFINE CLUSTER command. You must specify the REUSE parameter when allocating HALDB data sets. Use of this command is described in z/OS DFSMS Access Method Services for Catalogs.

Attention: If you use the Database Image Copy 2 utility to take image copies of your database, the data sets must be allocated on hardware that supports either the DFSMS concurrent copy function or the DFSMS fast replication function.

When loading databases (excluding HALDB databases) that contain logical relationships or secondary indexes, DL/I writes a control record to a work file (DFSURWF1). This work file must also be allocated and in the JCL.

All other data sets are allocated using normal z/OS JCL. You can use the z/OS program IEFBR14 to preallocate data sets, except when the database is an MSDB. For MSDBs, you should use the z/OS program IEHPROGM.

For more information about the standard DFSMS methods for allocating data sets and about data sets in general, see:
- *z/OS DFSMS: Using Data Sets*
- *z/OS DFSMS Access Method Services for Catalogs*

Related tasks:

"Changing the number of data set groups" on page 749

Using OSAM as the access method

OSAM is a special access method supplied with IMS.

This topic contains Product-sensitive Programming Interface information.

You need to know the following information about OSAM if your database is using OSAM as an access method:

- IMS communicates with OSAM using OPEN, CLOSE, READ, and WRITE macros.
- OSAM communicates with the I/O supervisor using the I/O driver interface.
- An OSAM data set can be read using either the BSAM or QSAM access method.
- You can define sequential OSAM data sets that use the BSAM access method as z/OS large format data sets by specifying DSNTYPE=LARGE on the DD statement. Large format data sets can exceed 65,535 tracks.
- The maximum number of extents in an OSAM data set is 120. However, the maximum number might be less if the length of the z/OS data extent block (DEB) that is associated with the OSAM data set increases.

  The length of a DEB is represented in a single byte that is expressed as the number of double words. The format and length (expressed in double words) of a DEB are shown in the following table.

Table 77. Length and format of an OSAM DEB

Format	Length in double words
Appendage sector table	5
Basic DEB	4
Access method dependent section	2
Subroutine name section	4

- An OSAM data set can be opened for update in place and extension to the end through one data control block (DCB). The phrase "extension to the end" means that records can be added to the end of the data set and that new direct-access extents can be obtained.
- An OSAM data set does not need to be formatted before use.
- An OSAM data set can use fixed-length blocked or unblocked records.
- The maximum size of an OSAM data set depends on the block size of the data set and whether it is a HALDB OSAM data set. The size limits for OSAM data sets are:
  - For a non-HALDB database, 4 GB (if the data set has an odd-length block size) or 8 GB (if the data set has an even-length block size)
  - For a HALDB database partition, 4 GB or 8 GB (if the HALDB is registered to DBRC as supporting 8-GB OSAM data sets)
- File mark definition is always used to define the current end of the data set. When new blocks are added to the end of the data set, they replace dummy pre-formatted (by OSAM) blocks that exist on a logical cylinder basis. A file mark is written at the beginning of the next cylinder, if one exists, during a format logical cylinder operation. This technique is used as a reliability aid while the OSAM data set is open.
- OSAM EXCP counts are accumulated during OSAM End of Volume (EOV) and close processing.
- Migrating OSAM data sets utilizing ADRDSSU and the DFSMSdss component of z/OS DFSMS: DFSMSdss will migrate the tracks of a data set up to the last block written value (DS1LSTAR) as specified by the DSCB for the volume being migrated. If the OSAM data set spans multiple volumes that have not been pre-allocated, the DS1LSTAR field for each DSCB will be valid and DFSMSdss can correctly migrate the data.

  If the OSAM data set spans multiple volumes that have been pre-allocated, the DS1LSTAR field in the DSCB for each volume (except the last) can be zero. This condition will occur during the loading operation of a pre-allocated,

multi-volume data set. The use of pre-allocated volumes precludes EOV processing when moving from one volume to another, thereby allowing the DSCBs for these volumes not to be updated. The DSCB for the last volume loaded is updated during close processing of the data set.

DFSMSdss physical DUMP or RESTORE commands with the parameters ALLEXCP or ALLDATA must be used when migrating OSAM data sets that span multiple pre-allocated volumes. These parameters will allow DFSMSdss to correctly migrate OSAM data sets.

Related Reading: For more information on the z/OS DFSMSdss component of DFSMS and the ALLEXCP and ALLDATA parameters of the DUMP and RESTORE commands, see the *z/OS DFSMS Storage Administration Reference (for DFSMSdfp, DFSMSdss, and DFSMShsm)*.

- You can enable OSAM data sets to reside in the extended addressing space of extended address volumes (EAVs) that are available in z/OS V1.12 or later. To enable an OSAM data set to reside in the extended addressing space of EAVs, specify an EAV volume on the VOL=SER= parameter when you allocate the data set. In addition, specify the attribute EATTR=OPT to indicate that the data set supports the extended attributes needed for use of the extended addressing area.

  Restriction: Data sets with EATTR=OPT specified on them cannot be shared with an IMS Version 10 or IMS Version 11 system because those IMS versions do not support extended attributes.

Other z/OS access methods (VSAM and SAM) are used in addition to OSAM for physical storage of data.

Related concepts:

"Maximum sizes of HD databases" on page 146

➡ OSAM subpool definition (System Definition)

Related reference:

➡ High-speed sequential processing control statements (System Definition)

Allocating OSAM data sets

For databases other than HALDB databases, you should use JCL to allocate OSAM data sets at the time the data set is loaded. This mode of allocation can be for single or multiple volumes, using the SPACE parameter.

OSAM also supports large format sequential data sets, which can be used as an alternative to multi-volume data sets.

Related concepts:

"Recovery and data sets" on page 609

Allocating single-volume OSAM data sets

You can allocate single-volume OSAM data sets by using JCL or z/OS DFSMS Access Method Services (AMS) IDCAMS.

- To allocate a single-volume OSAM data set by using JCL, you can model your JCL after the following example JCL.

```
//ALLOC1  EXEC  PGM=IEFBR14
//DD1     DD    DSN=HIGHNODE.MIDNODE.LOWNODE,
//              DISP=(NEW,CATLG),
//              SPACE=(CYLS,(200,100)),
```

```
//*  add UNIT and VOLSER if needed
//*  add SMS parameters if needed
//*  do not code DCB parameters
/*
```

- To allocate a single-volume OSAM data set by using AMS IDCAMS, you can model your JCL after the following example JCL.

```
//ALLOC1   EXEC  PGM=IDCAMS,REGION=4096K
//SYSPRINT DD    SYSOUT=*
//SYSIN    DD    *
  /*                                        */
  /* DELETE OLD DATA SET IF IT EXISTS       */
  /*                                        */
  DELETE 'HIGHNODE.MIDNODE.ENDNODE'

  IF LASTCC <= 8 THEN SET MAXCC = 0

  /*                                        */
  /*  ALLOCATE - WITH SMS PARAMETERS        */
  /*                                        */

  ALLOCATE DSN('HIGHNODE.MIDNODE.ENDNODE') -
           NEW CATALOG -
           DATACLAS(DCLAS) -
           MGMTCLAS(MCLAS) -
           STORCLAS(SCLAS) -
           SPACE(200 100) CYLINDERS
 /*
```

- To allocate an OSAM data set capable of residing in the extended addressing space on an extended address volume on non-SMS managed DASD, you can model your JCL after the following example JCL.

```
//OSAMALLO   JOB  A,OSAMEXAMPLE
//S1         EXEC  PGM=IEFBR14
//EAVD            DD  VOL=SER=EAV001,SPACE=(CYL,(20,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP),EATTR=OPT
```

Allocating multi-volume OSAM data sets

For multi-volume HALDB database data sets, you should preallocate your OSAM data sets. For single volume HALDB OSAM data sets, the allocation can be done either before or during the load step, although doing it in the load step is recommended.

If the installation control of direct-access storage space and volumes require that the OSAM data sets be preallocated, or if a message queue data set requires more than one volume, the OSAM data sets might be preallocated.

Observe the following restrictions when you preallocate with any of the accepted methods:

- DCB parameters should not be specified.
- Secondary allocation must be specified for all volumes if the data set will be extended beyond the primary allocation.
- Secondary allocation must be specified for all volumes in order to write to volumes preallocated but not written to by initial load or reload processing.
- Secondary allocation is not allowed for queue data sets because queue data sets are not extended beyond their initial or preallocated space quantity. However, queue data sets can have multi-volume allocation.
- The secondary allocation size defined on the first volume will be used for all secondary allocations on all volumes regardless of the secondary allocation size

specified on the other volumes. All volumes should be defined with the same secondary allocation size to avoid confusion.
- If the OSAM data set will be cataloged, use IEHPROGM or Access Method Services to ensure that all volumes are included in the catalog entry.

When a multi-volume data set is preallocated, you should allocate extents on all the volumes to be used. The suggested method of allocation is to have one IEFBR14 utility step for each volume on which space is desired.

Restriction:
- **Do not** use IEFBR14 and specify a DD card with a multi-volume data set, because this allocates an extent on only the first volume.
- **Do not** use a separate IEFBR14 utility step to allocate extents on each volume, as described above, if you plan to use the Database Image Copy 2 utility (DFSUDMT0). Using a separate IEFBR14 step for each volume creates extents that have the same identifier (volume 1). The Database Image Copy 2 utility requires that you allocate your multi-volume data sets using the standard DFSMS methods.

Example of allocating multi-volume OSAM data sets on non-SMS managed DASD:

The JCL in the following example allocates a multi-volume OSAM data set on a non-SMS managed direct-access storage device (DASD).

The data sets allocated with this JCL are not compatible with the Database Image Copy 2 utility.

JCL allocating a multi-volume OSAM data set on non-SMS managed DASD
```
//OSAMALLO   JOB  A,OSAMEXAMPLE
//S1        EXEC  PGM=IEFBR14
//EXTENT1    DD   VOL=SER=AAAAAA,SPACE=(CYL,(20,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
//S2        EXEC  PGM=IEFBR14
//EXTENT2    DD   VOL=SER=BBBBBB,SPACE=(CYL,(30,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
     .
     .
     .
//LAST      EXEC  PGM=IEFBR14
//EXTENTL    DD   VOL=SER=LLLLLL,SPACE=(CYL,(30,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
//          EXEC  PGM=IEHPROGM
//SYSPRINT   DD   SYSOUT=*
//DD1        DD   UNIT=3390,VOL=SER=AAAAAA,DISP=SHR
//DD2        DD   UNIT=3390,VOL=SER=BBBBBB,DISP=SHR
     .
     .
     .
//DDL        DD   UNIT=3390,VOL=SER=LLLLLL,DISP=SHR
//SYSIN      DD   *
  CATLG DSNAME=OSAM.SPACE,VOL=3390=(AAAAAA,BBBBBB,LLLLLL)
/*
```
Related concepts:
"Cautions when allocating multi-volume OSAM data sets" on page 572

Example of allocating multi-volume OSAM data sets on SMS-managed DASD:

The following examples show alternate methods for allocating multi-volume OSAM data sets on SMS-managed DASD.

When allocating data sets using the following JCL, you must specify an SMS storage class that has been defined by your installation with guaranteed space (GUARANTEED_SPACE=YES) and specify the volume serial numbers. Without the guaranteed space attribute and the volume serial numbers, only the first volume will get a primary extent and the other volumes will get secondary extents.

The following JCL is an example of the JCL that you can use to allocate multi-volume OSAM data sets. These data sets are compatible with the Database Image Copy 2 utility. In the example, *gtdstcls* is a storage class that is defined with the guaranteed space attribute.

```
//OSAMALLO   JOB  A,OSAMEXAMPLE
//         EXEC  PGM=IEFBR14
//DD123      DD DSN=HIGHNODE.MIDNODE.ENDNODE,
//           DISP=(NEW,CATLG),
//           SPACE=(CYL,(200,100)),
//           UNIT=(3390)
//           VOL=SER=(VOL111,VOL222,VOL333),
//           STORCLAS=gtdstcls
/*
```

The following JCL is an example of using the DFSMS Access Method Services ALLOCATE command to allocate multi-volume OSAM data sets. These data sets are compatible with the Database Image Copy 2 utility. In the example, *gtdstcls* is a storage class that is defined with the guaranteed space attribute.

```
//OSAMALLO   JOB  A,OSAMEXAMPLE
//         EXEC  PGM=IDCAMS,REGION=4096K
//SYSPRINT   DD  SYSOUT=A
//SYSIN      DD *
         DELETE 'HIGHNODE.MIDNODE.ENDNODE'

         IF LASTCC<=8 THEN SET MAXCC=0

         ALLOCATE DSN('HIGHNODE.MIDNODE.ENDNODE') -
         NEW CATALOG -
         SPACE(200,100) CYLINDERS -
         UNIT(3390) -
         VOL(VOL111,VOL222,VOL333) -
         STORCLAS(gtdstcls)
/*
```

For more information about allocating space for data sets on SMS-managed DASD, see *z/OS DFSMS Storage Administration Reference (for DFSMSdfp, DFSMSdss, and DFSMShsm)*.

Related concepts:

"Cautions when allocating multi-volume OSAM data sets"

Cautions when allocating multi-volume OSAM data sets:

Keep in mind the following advisories when allocating multi-volume OSAM data sets.

1. Preallocating more volumes for OSAM data set extents than are used during initial load or reload processing might cause an abend if you attempt to extend the data set beyond the last volume written to at initial load or reload time under the following circumstances: the initial load or reload step did not result

in the data being written to the last volume of the preallocated data set, and secondary allocation was not specified during data set preallocation.

2. When loading a database, the volumes of the data sets are loaded in the order that their volume serial numbers are listed in the CATLG statement.

3. Do not reuse a multivolume OSAM data set without first scratching the data set and then reallocating the space. This also applies to the output data sets for the HALDB Online Reorganization function.

   Failure to scratch and reallocate the data set might cause an invalid EOF mark to be left in the DSCB of the last volume of the data set when the data set is:

   a. First reused by an IMS utility (such as the Unload/Reload utility used in database reorganization).

   b. Then opened by OSAM for normal processing.

   For example, a data set might initially be allocated on three volumes, with the EOF mark on the third volume. However, after the reorganization utility is run, the data set might need only the first two volumes. Therefore, the new EOF mark is placed on the second volume. After reorganization, when the data set is opened by OSAM for normal processing, OSAM checks the last volume's DSCB for an EOF mark. When OSAM finds the EOF in the third volume, it inserts new data after the old EOF mark in the third volume instead of inserting data after the EOF mark created by the reorganization utility in the second volume.

   Subsequent processing by another utility such as the Image Copy utility uses the EOF mark set by the reorganization utility on the second volume and ignores new data inserted by OSAM on volume three.

4. If you intend to use the Image Copy 2 utility (DFSUDMT0) to back up and restore multi-volume databases, they MUST be allocated using the standard DFSMS techniques.

   For more information about the DFSMS methods of data set allocation, see *z/OS DFSMS: Using Data Sets*.

Related concepts:

"Example of allocating multi-volume OSAM data sets on non-SMS managed DASD" on page 571

"Example of allocating multi-volume OSAM data sets on SMS-managed DASD" on page 572

Allocating an OSAM large format sequential data set

OSAM supports large format sequential data sets. Large format sequential data sets can exceed 65 535 tracks or 3.4 GB on a single volume.

The following JCL is an example of allocating an OSAM large format sequential data set. The data sets allocated with this JCL are compatible with the Database Image Copy 2 utility.

```
//OSAMALBG   JOB  A,OSAMEXAMPLE
//S1        EXEC  PGM=IEFBR14
//AJOSAMDB DD  DSN=IMSTESTL.AJOSAMDB,UNIT=SYSDA,
//           DISP=(NEW,CATLG),DSNTYPE=LARGE,
//           SPACE=(CYL,(4500,100)),VOL=SER=LRGVS1
```

The following JCL is an example of allocating an OSAM large format sequential data set on an extended address volume (EAV). The data sets allocated with this JCL are compatible with the Database Image Copy 2 utility.

```
//OSAMALBG   JOB  A,OSAMEXAMPLE
//S1        EXEC  PGM=IEFBR14
//AJOSAMDB DD  DSN=IMSTESTL.AJOSAMDB,UNIT=SYSDA,
//              DISP=(NEW,CATLG),DSNTYPE=LARGE,
//              SPACE=(CYL,(4500,100)),VOL=SER=EAV001,EATTR=OPT
```

Writing a load program

After you have determined how much space your database requires and allocated data sets for it, you can load the database.

The load process

Loading the database is done using an initial load program. Initial load programs must be batch programs, since you cannot load a database with an online application program. It is your responsibility to write this program.

Basically, an initial load program reads an existing file containing your database records. Using the DBD, which defines the physical characteristics of the database, and the load PSBs, the load program builds segments for a database record and inserts them into the database in hierarchical order.

The following figure shows the load process.
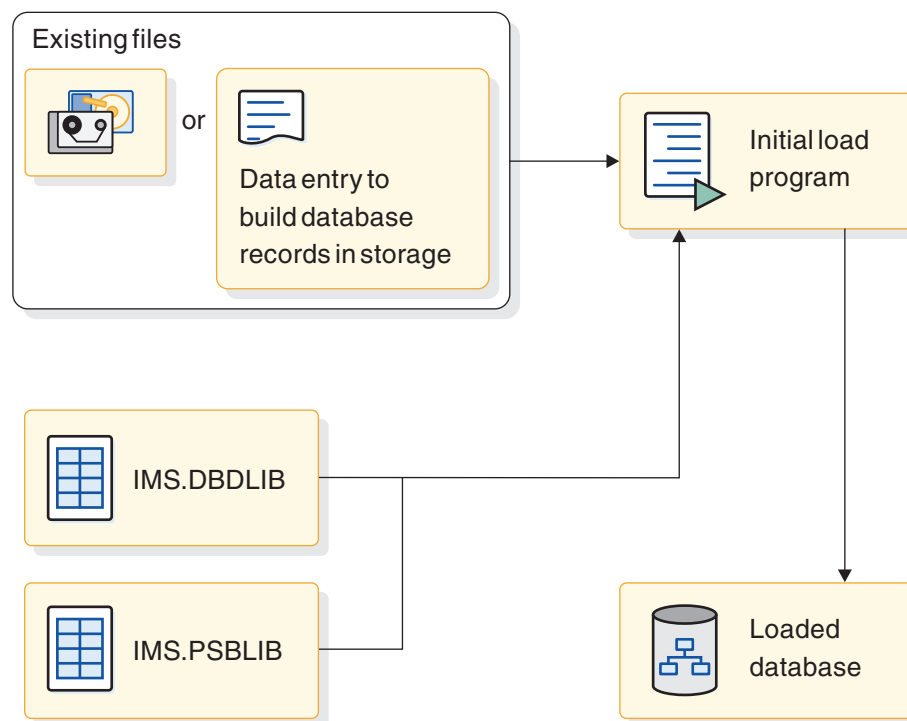
Figure 239. The load process

If the data to be loaded into the database already exists in one or more files, merge and sort the data, if necessary, so that it is presented to the load program in correct sequence. Also, if you plan to merge existing files containing redundant data into one database, delete the redundant data, if necessary, and correct any data that is wrong.

The following figure illustrates loading a database using existing files.
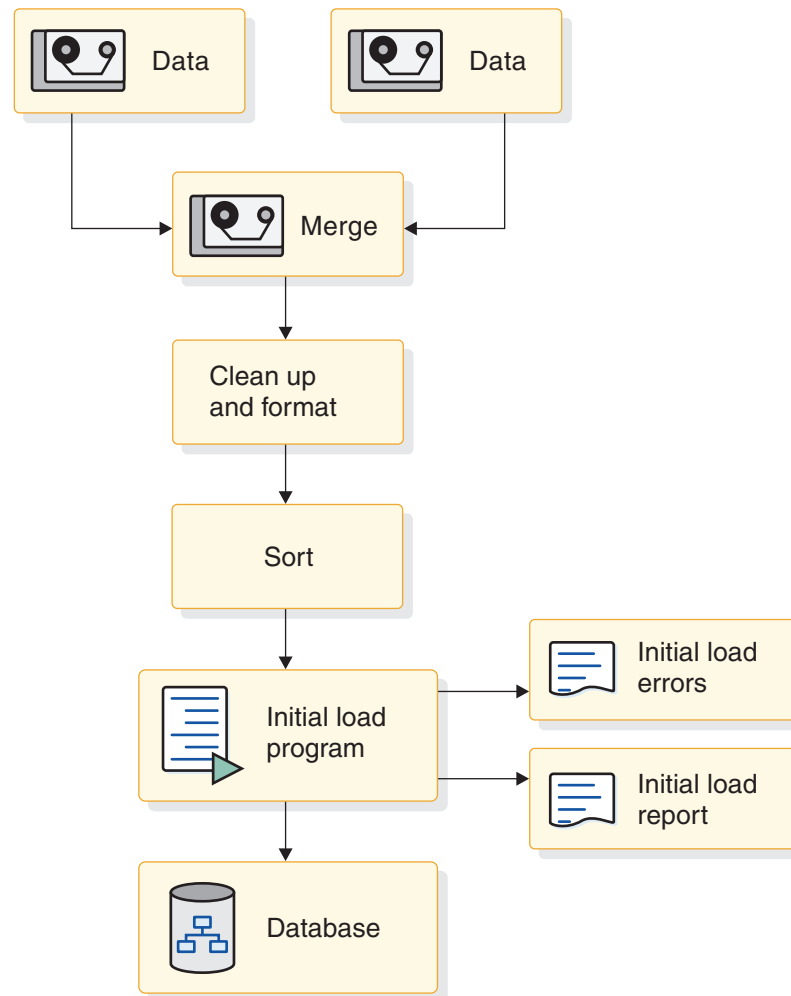
Figure 240. Loading a database using existing files

After you have defined the database, you load it by writing an application program that uses the ISRT call. An initial load program builds each segment in the program's I/O area, then loads it into the database by issuing an ISRT call for it. ISRT calls are the only DL/I requests allowed when you specify PROCOPT=L in the PCB. The only time you use the "L" option is when you initially load a database. This option is valid only for batch programs.

Restriction: A PSB that includes a PCB statement that specifies PROCOPT=L cannot contain other PCB statements that specify PROCOPT values of A, D, G, I, or R.

If the database being loaded is registered with DBRC, DBRC authorization is also required for all databases that are logically related to the database being loaded. If DBRC is active when the database is loaded, DBRC sets the image copy status for this database to IC NEEDED in the DBDS record in the RECON data set.

The FIRST, LAST, and HERE insert rules do not apply when you are loading a database, unless you are loading an HDAM database. When you are loading a

HDAM database, the rules determine how root segments with non-unique sequence fields are ordered. If you are loading a database using HSAM, the same rules apply.

Recommendation: Load programs do not need to issue checkpoints.

Most comprehensive databases are loaded in stages by segment type or by groups of segment types. Because there are usually too many segments to load using only one application program, you need several programs to do the loading. Each load program after the first load program is technically an "add" program, not a load program. Do not specify "L" as the processing option in the PCB for add programs. You should review any add type of load program written to load a database to ensure that the program's performance will be acceptable; it usually takes longer to add a group of segments than to load them.

For HSAM, HISAM, HIDAM, and PHIDAM, the root segments that the application program inserts must be pre-sorted by the key fields of the root segments. The dependents of each root segment must follow the root segment in hierarchical sequence, and must follow key values within segment types. In other words, you insert the segments in the same sequence in which your program would retrieve them if it retrieved in hierarchical sequence (children after their parents, database records in order of their key fields).

If you are loading an HDAM or PHDAM database, you do not need to pre-sort root segments by their key fields.

When you load a database:
- If a loaded segment has a key, the key value must be in the correct location in the I/O area.
- When you load a logical child segment, the I/O area must contain the logical parent's concatenated key, followed by the logical child segment to be inserted.
- After issuing an ISRT call, the current position is just before the next available space following the last segment successfully loaded. The next segment you load will be placed in that space.

Recommendation: You should always create an image copy immediately after you load, reload, or reorganize the database.

Related concepts:

"Adding logical relationships" on page 514

Status codes for load programs

If the ISRT call is successful, DL/I returns a blank status code for the program. If not, DL/I returns one of several possible status codes.

If an ISRT call is unsuccessful, DL/I returns one of the following status codes:

LB The segment you are trying to load already exists in the database. DL/I only returns this status code for segments with key fields.

In a call-level program, you should transfer control to an error routine.

LC The segment you are trying to load is out of key sequence.

LD No parent exists for this segment. This status code usually means that the segment types you are loading are out of sequence.

LE In an ISRT call with multiple SSAs, the segments named in the SSAs are not in their correct hierarchical sequence.

LF Initial load of PHDAM or PHIDAM attempted ISRT of a logical child segment.

V1 You have supplied a variable-length segment whose length is invalid.

Using SSAs in a load program

When you are loading segments into the database, you do not need to worry about position, because DL/I inserts one segment after another. The most important part of loading a database is the order in which you build and insert the segments.

The only SSA you must supply is the unqualified SSA giving the name of the segment type you are inserting.

Because you do not need to worry about position, you need not use SSAs for the parents of the segment you are inserting. If you do use them, be sure they contain only the equal (EQ, =b, or b=) relational operator. You must also use the key field of the segment as the comparative value.

For HISAM, HIDAM, and PHIDAM, the key X'FFFF' is reserved for IMS. IMS returns a status code of LB if you try to insert a segment with this key.

Loading a sequence of segments with the D command code

You can load a sequence of segments in one call by concatenating the segments in the I/O area and supplying DL/I with a list of unqualified SSAs.

You must include the D command code with the first SSA. The sequence that the SSAs define must lead down the hierarchy, with each segment in the I/O area being the child of the previous segment.

Two types of initial load program

Two types of initial load programs exist: *basic* and *restartable*.

The basic program must be restarted from the beginning if problems occur during execution. The restartable program can be restarted at the last checkpoint taken before problems occurred. Restartable load programs must be run under control of the Utility Control Facility (UCF) and require VSAM as the access method. The following topics describe both types of load programs:

Basic initial load program

You should write a basic initial load program (one that is not restartable) when the volume of data you need to load is not so great that you would be seriously set back if problems occurred during program execution.

If problems do occur, the basic initial load program must be rerun from the beginning.

Fast Path Data Entry Databases (DEDBs) cannot be loaded in a batch job as can DL/I databases. DEDBs are first initialized by the DEDB Initialization Utility and then loaded by a user-written Fast Path application program that executes typically in a BMP region.

Fast Path Main Storage Databases (MSDBs) are not loaded until the IMS control region is initialized. These databases are then loaded by the IMS start-up procedure when the following requirements are met:

- The MSDB= parameter on the EXEC Statement of Member Name IMS specifies a one-character suffix to DBFMSDB in IMS.PROCLIB.
- The member contains a record for each MSDB to be loaded.

  The record contains a record for each MSDB, the number of segments to be loaded, and an optional "F" which indicates that the MSDB is to be fixed in storage.

- A sequential data set, part of a generation data group (GDG) with dsname IMS.MSDBINIT(0), is generated.

  This data set can be created by a user-written program or by using the INSERT function of the MSDB Maintenance utility. Records in the data set are sequenced by MSDB name, and within MSDBs by key.

The following figure shows the logic for developing a basic initial load program.
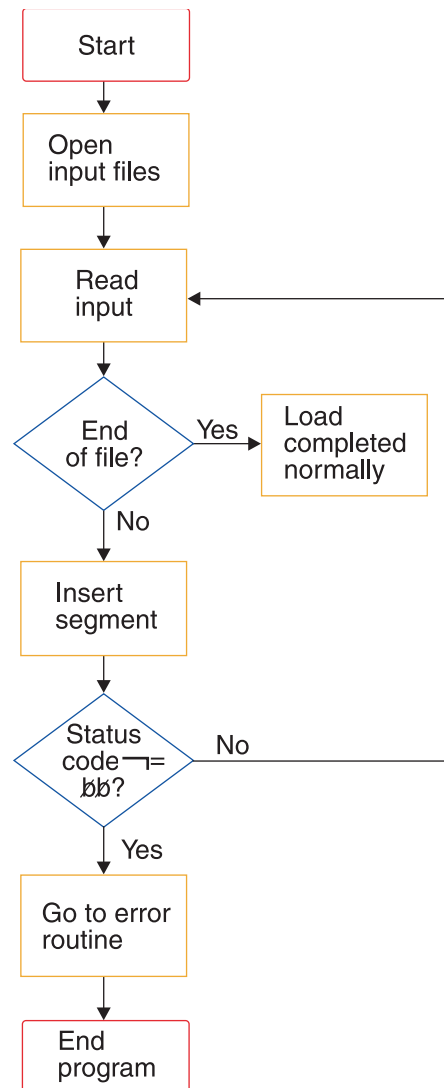
Figure 241. Basic initial load program logic

The following code is a sample load program that satisfies the basic IMS database loading requirements. A sample program showing how this can be done with the Utility Control Facility is also provided.

```
DLITCBL  START
         PRINT   NOGEN
         SAVE    (14,12),,LOAD1.PROGRAM    SAVE REGISTERS
         USING   DLITCBL,10                DEFINE BASE REGISTER
         LR      10,15                     LOAD BASE REGISTER
         LA      11,SAVEAREA               PERFORM
         ST      13,4(11)                   SAVE
         ST      11,8(13)                    AREA
         LR      13,11                        MAINT
         L       4,0(1)                    LOAD PCB BASE REGISTER
         STCM    4,7,PCBADDR+1             STORE PCB ADDRESS IN CALL LIST
         USING   DLIPCB,4                  DEFINE PCB BASE REGISTER
         OPEN    (LOAD,(INPUT))            OPEN LOAD DATA SOURCE FILE
LOOP     GET     LOAD,CARDAREA             GET SEGMENT TO BE INSERTED
INSERT   CALL    CBLTDLI,MF=(E,DLILINK)    INSERT THE SEGMENT
         AP      SEGCOUNT,=P'1'            INCREMENT SEGMENT COUNT
         CLC     DLISTAT,=CL2' '           WAS COMPLETION NORMAL?
         BE      LOOP                      YES - KEEP GOING
ABEND    ABEND   8,DUMP                    INVALID STATUS
EOF      WTO     'DATABASE 1 LOAD COMPLETED NORMALLY'
         UNPK    COUNTMSG,SEGCOUNT         UNPACK SEGMENT COUNT FOR WTO
         OI      COUNTMSG+4,X'F0'          MAKE SIGN PRINTABLE
         WTO     MF=(E,WTOLIST)            WRITE SEGMENT COUNT
         CLOSE   (LOAD)                    CLOSE INPUT FILE
         L       13,4(13)                  UNCHAIN SAVE AREA
         RETURN  (14,12),RC=0              RETURN NORMALLY
         LTORG
SEGCOUNT DC      PL3'0'
         DS      0F
WTOLIST  DC      AL2(LSTLENGT)
         DC      AL2(0)
COUNTMSG DS      CL5
         DC      C' SEGMENTS PROCESSED'
LSTLENGT EQU     (*-WTOLIST)
DLIFUNC  DC      CL4'ISRT'                 FUNCTION CODE
DLILINK  DC      A(DLIFUNC)                DL/I CALL LIST
PCBADDR  DC      A(0)
         DC      A(DATAAREA)
         DC      X'80',AL3(SEGNAME)
CARDAREA DS      0CL80                     I/O AREA
SEGNAME  DS      CL9
SEGKEY   DS      0CL4
DATAAREA DS      CL71
SAVEAREA DC      18F'0'
LOAD     DCB     DDNAME=LOAD1,DSORG=PS,EODAD=EOF,MACRF=(GM),RECFM=FB
DLIPCB   DSECT   ,                         DATABASE PCB
DLIDBNAM DS      CL8
DLISGLEV DS      CL2
DLISTAT  DS      CL2
DLIPROC  DS      CL4
DLIRESV  DS      F
DLISEGFB DS      CL8
DLIKEYLN DS      CL4
DLINUMSG DS      CL4
DLIKEYFB DS      CL12
         END
```

Related concepts:

Tailoring the IMS system to your environment (System Definition)

Related reference:

Definition and initialization utilities (Database Utilities)

➡ MSDB Maintenance utility (DBFDBMA0) (Database Utilities)

Restartable initial load program

You should write a restartable initial load program (one that can be restarted from the last checkpoint taken) when the volume of data you need to load is great enough that you would be seriously set back if problems occurred during program execution.

If problems occur and your program is not restartable, the entire load program has to be rerun from the beginning.

Restartable load programs differ from basic load programs in their logic. If you already have a basic load program, usually only minor changes are required to make it restartable. The basic program must be modified to recognize when restart is taking place, when WTOR requests to stop processing have been made, and when checkpoints have been taken.

To make your initial database load program restartable under UCF, consider the following points when you are planning and writing the program:

- If a program is being restarted, the PCB status code will contain a UR prior to the issuance of the first DL/I call. The key feedback area will contain the fully concatenated key of the last segment inserted prior to the last UCF checkpoint taken. (If no checkpoints were taken prior to the failure, this area will contain binary zeros.)
- The UCF does not checkpoint or reposition user files. When restarting, it is the user's responsibility to reposition all such files.
- When restarting, the first DL/I call issued must be an insert of a root segment.

  For HISAM and HIDAM Index databases, the restart will begin with a GN and a VSAM ERASE sequence to reinsert the higher keys. The resume operation then takes place. Space in the KSDS is reused (recovered) but not in the ESDS.

  For HDAM, the data will be compared if the root sequence field is unique and a root segment insert is done for a segment that already exists in the database because of segments inserted after the checkpoint. If the segment data is the same, the old segment will be overlaid and the dependent segments will be dropped since they will be reinserted by a subsequent user/reload insert. This occurs only until a non-duplicate root is found. Once a segment with a new key or with different data is encountered, LB status codes will be returned for any subsequent duplicates. Therefore, space is reused for the roots, but lost for the dependent segments.

  For HDAM with non-unique keys, any root segments that were inserted after the checkpoint at which the restart was made will remain in the database. This is also true for their dependent segments.
- When the stop request is received, UCF will take a checkpoint just prior to inserting the next root. If the application program fails to terminate, it will be presented the same status code at each of the following root inserts until normal termination of the program.
- For HISAM databases, the RECOVERY option must be specified. For HD organizations, either RECOVERY or SPEED can be defined to Access Method Services.
- UCF checkpoints are taken when the checkpoint count (CKPNT=) has expired and a root insert has been requested. The count refers to the number of root segments inserted and the checkpoint is taken immediately prior to the insertion of the root.

The following figure shows the logic for developing a restartable initial load program.
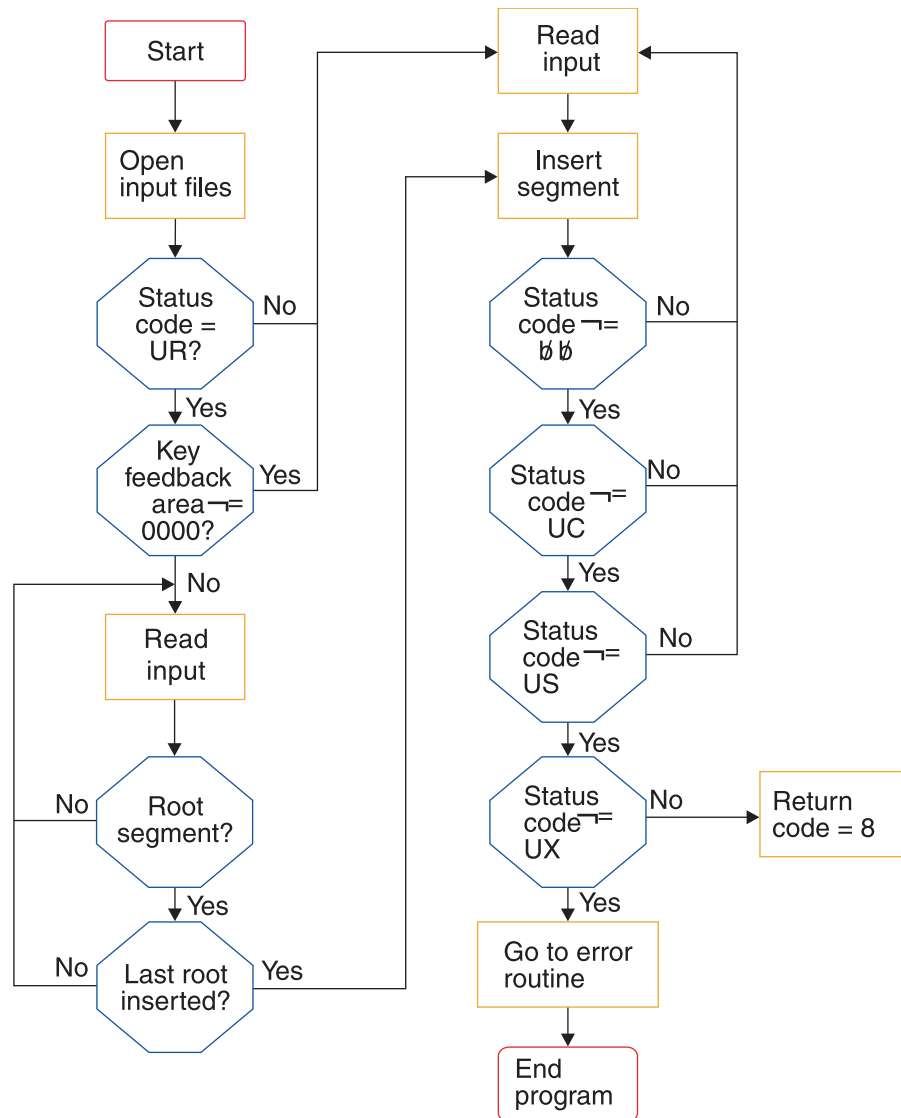
Figure 242. Restartable initial load program logic

The following lists explains the status codes shown in the preceding figure:

UR   Load program being restarted under control of UCF

UC   Checkpoint record written to UCF journal data set

US   Initial load program prepared to stop processing

UX   Checkpoint record was written and processing stopped

The following code is for a sample restartable initial load program:

```
DLITCBL  START
         PRINT   NOGEN
         SAVE    (14,12),,LOAD1.PROGRAM     SAVE REGISTERS
         USING   DLITCBL,10                 DEFINE BASE REGISTER
         LR      10,15                      LOAD BASE REGISTER
         LA      11,SAVEAREA                PERFORM
         ST      13,4(11)                    SAVE
```

```
                ST      11,8(13)                 AREA
                LR      13,11                     MAINT
                L       4,0(1)                   LOAD PCB BASE REGISTER
                STCM    4,7,PCBADDR+1            STORE PCB ADDRESS IN CALL LIST
                USING   DLIPCB,4                 DEFINE PCB BASE REGISTER
                OPEN    (LOAD,(INPUT))           OPEN LOAD DATA SOURCE FILE
                CLC     DLISTAT,=C'UR'           IS THIS A RESTART?
                BNE     NORMAL                   NO - BRANCH
                CLC     DLIKEYFB(4),=X'00000000' IS KEY FEEDBACK AREA ZERO?
                BE      NORMAL                   YES - BRANCH
RESTART  WTO     'RESTART LOAD PROCESSING FOR DATABASE 1 IS IN PROCESS'
RLOOP    GET     LOAD,CARDAREA            GET A LOAD RECORD
                CLC     SEGNAME(8),=CL8'SEGMA'   IS THIS A ROOT SEGMENT RECORD?
                BNE     RLOOP                    NO - KEEP LOOKING
                CLC     DLIKEYFB(4),SEGKEY       IS THIS THE LAST ROOT INSERTED?
                BNE     RLOOP                    NO - KEEP LOOKING
                B       INSERT                   GO DO IT
NORMAL   WTO     'INITIAL LOAD PROCESSING FOR DATABASE 1 IS IN PROCESS'
LOOP     GET     LOAD,CARDAREA            GET SEGMENT TO BE INSERTED
INSERT   CALL    CBLTDLI,MF=(E,DLILINK)   INSERT THE SEGMENT
                AP      SEGCOUNT,=P'1'           INCREMENT SEGMENT COUNT
                CLC     DLISTAT,=CL2' '          WAS COMPLETION NORMAL?
                BE      LOOP                     YES - KEEP GOING
                CLC     DLISTAT,=CL2'UC'         HAS CHECKPOINT BEEN TAKEN?
                BNE     POINT1                   NO - KEEP CHECKING
POINT0   WTO     'UCF CHECKPOINT TAKEN FOR LOAD 1 PROGRAM'
                UNPK    COUNTMSG,SEGCOUNT        UNPACK SEGMENT COUNT FOR WTO
                OI      COUNTMSG+4,X'F0'         MAKE SIGN PRINTABLE
                WTO     MF=(E,WTOLIST)           WRITE SEGMENT COUNT
                B       LOOP                     NO - KEEP GOING
POINT1   CLC     DLISTAT,=CL2'US'         HAS OPERATOR REQUESTED STOP?
                BNE     POINT2                   NO - KEEP CHECKING
                B       LOOP                     KEEP GOING
POINT2   CLC     DLISTAT,=CL2'UX'         COMBINED CHECKPOINT AND STOP?
                BNE     ABEND                    NO - GIVE UP
                WTO     'LOAD1 PROGRAM STOPPING PER OPERATOR REQUEST'
                B       RETURN8
ABEND    ABEND   8,DUMP                   INVALID STATUS
EOF      WTO     'DATABASE 1 LOAD COMPLETED NORMALLY'
                UNPK    COUNTMSG,SEGCOUNT        UNPACK SEGMENT COUNT FOR WTO
                OI      COUNTMSG+4,X'F0'         BLAST SIGN
                WTO     MF=(E,WTOLIST)           WRITE SEGMENT COUNT
                CLOSE   (LOAD)                   CLOSE INPUT FILE
                L       13,4(13)                 UNCHAIN SAVE AREA
                RETURN  (14,12),RC=0             RETURN NORMALLY
RETURN8  WTO     'DATABASE 1 LOAD STOPPING FOR RESTART'
                UNPK    COUNTMSG,SEGCOUNT        UNPACK SEGMENT COUNT FOR WTO
                OI      COUNTMSG+4,X'F0'         BLAST SIGN
                WTO     MF=(E,WTOLIST)           WRITE SEGMENT COUNT
                CLOSE   (LOAD)                   CLOSE INPUT FILE
                L       13,4(13)                 UNCHAIN SAVE AREA
                RETURN  (14,12),RC=8             RETURN AS RESTARTABLE
                LTORG
SEGCOUNT DC      PL3'0'
                DS      0F
WTOLIST  DC      AL2(LSTLENGT)
                DC      AL2(0)
COUNTMSG DS      CL5
                DC      C' SEGMENTS PROCESSED'
LSTLENGT EQU     (*-WTOLIST)
DLIFUNC  DC      CL4'ISRT'                FUNCTION CODE
DLILINK  DC      A(DLIFUNC)               DL/I CALL LIST
PCBADDR  DC      A(0)
                DC      A(DATAAREA)
                DC      X'80',A13(SEGNAME)
CARDAREA DS      0CL80                    I/O AREA
SEGNAME  DS      CL9
```

```
SEGKEY   DS     0CL4
DATAAREA DS     CL71
SAVEAREA DC     18F'0'
STOPNDG  DC     X'00'
LOAD     DCB    DDNAME=LOAD1,DSORG=PS,EODAD=EOF,MACRF=(GM),RECFM=FB
DLIPCB   DSECT                         DATABASE PCB
DLIDBNAM DS     CL8
DLISGLEV DS     CL2
DLISTAT  DS     CL2
DLIPROC  DS     CL4
DLIRESV  DS     F
DLISEGFB DS     CL8
DLIKEYLN DS     CL4
DLINUMSG DS     CL4
DLIKEYFB DS     CL12
         END
```

Related reference:

➡ Utility Control Facility (DFSUCF00) (Database Utilities)

JCL for the initial load program

The JCL is an example of the JCL that you use to initially load your database.

The //DFSURWF1 DD statement is present only if a logical relationship or secondary index exists.

JCL used to initially load a database

```
//         EXEC PGM=DFSRRC00,PARM='DLI,your initial load program name,
//              your PSB name'
//DFSRESLB  DD   references an authorized library that contains IMS
//               SVC modules
//STEPLIB   DD   references library that contains your load program
//          DD   DSN=IMS.SDFSRESL
//IMS       DD   DSN=IMS.PSBLIB,DISP=SHR
//          DD   DSN=IMS.DBDLIB,DISP=SHR
//DFSURWF1  DD   DCB=(RECFM=VB,LRECL=300,
//               BLKSIZE=(you must specify),
//               DSN=WF1,DISP=(MOD,PASS)
//DBNAME    DD   references the database data set to be
//               initially loaded or referenced by
//               the initial load program
//INPUT     DD   input to your initial load program
//DFSVSAMP  DD   input for VSAM and OSAM buffers and options
.
.
.
//*
```

Loading a HISAM database

Segments in a HISAM database are stored in the order in which you present them to the load program.

You must present all occurrences of the root segment in ascending key sequence and all dependent segments of each root in hierarchical sequence. PROCOPT=L (for load) must be specified in the PCB.

Loading a SHISAM database

Segments in a SHISAM database are stored in the order in which you present them to the load program.

You must present all occurrences of the root segment in ascending key sequence. PROCOPT=L (for load) must be specified in the PCB.

Loading a GSAM database

GSAM databases use logical records, not segments or database records. GSAM logical records are stored in the order in which you present them to the load program.

Loading an HDAM or a PHDAM database

In an HDAM or a PHDAM database, the user randomizing module determines where a database record is stored, so the sequence in which root segments are presented to the load program does not matter.

All dependents of a root should follow the root in hierarchical sequence. PROCOPT=L (for load) or PROCOPT=LS (for load segments in ascending sequence) must be specified in the PCB.

Loading a HIDAM or a PHIDAM database

To load a HIDAM or a PHIDAM database, you must present root segments in ascending key sequence and all dependents of a root should follow the root in hierarchical sequence.

PROCOPT=LS (for load segments in ascending sequence) must be specified in the PCB.

Restriction: Load programs for PHIDAM databases must run in a DLI region type. Load programs for HIDAM databases do not have this restriction.

Loading a database with logical relationships or secondary indexes

If you are loading a database with logical relationships or secondary indexes, you will need to run, in addition to your load program, some combination of the reorganization utilities.

You need to run the reorganization utilities to put the correct pointer information in each segment's prefix.

Related concepts:

"Reorganization utilities" on page 644

Loading Fast Path databases

This topic describes how to load MSDBs, DEDBs, and sequential dependent segments.

Loading an MSDB

Because MSDBs reside in main storage they are read from a data set and loaded during system initialization.

You do not load them as you do other IMS databases, that is, by means of a load program that you provide. You first build this data set either by using a program you provide or by running the MSDB Maintenance utility (DBFDBMA0).

Related concepts:

"MSDBs storage" on page 219

Related reference:

⬛ MSDB Maintenance utility (DBFDBMA0) (Database Utilities)

Loading a DEDB

You load data into a DEDB database with a load program similar to that used for loading other IMS databases. Unlike other load programs, this program runs as a batch message program.

The following five steps are necessary to load a DEDB:

1. Calculate space requirements

   The following example assures that root and sequential dependent segment types are loaded in one area.

   Assume all root segments are 200 bytes long (198 bytes of data plus 2 bytes for the length field) and that there are 850 root segments in the area. On the average, there are 30 SDEP segments per record. Each is 150 bytes long (148 bytes of data and a 2-byte length field). The CI size is 1024 bytes.

   A. Calculate the minimum space required to hold root segments:

   ```
   1024              CI length minus
   - 21              CI control fields
                     equals amount of space for root segments
   ────
   1003              and their prefixes.

   1003 / 214 = 4.6  Amount of root and root prefix space
                     divided by length of one root with its
                     prefix equals the number of segments
                     that will fit in one CI.
                     DEDB segments do not span CIs.
                     Therefore, only four
                     roots will fit in a CI.

    850 / 4 = 212.5  The minimum amount of space to hold
                     the defined number of roots to be
                     inserted in this area (850)
                     requires 213 CIs.
   ```

   After choosing a UOW size, you can determine the DBD specifications for the root addressable and independent overflow parts using the result of the above calculation as a base.

   B. Calculate the minimum space required to hold the sequential dependent segments:

   ```
   1024              CI length minus
   - 17              CI control fields
                     equals amount of space for sequential
   ────
   1007              dependents and their prefixes.

   1007 / 160 = 6.2  Amount of sequential dependent and
                     prefix space divided by length of one
                     sequential dependent plus its prefix
                     equals the number of segments that
                     will fit in one CI.
                     Six SDEP segments will fit in a
                     CI.

   30 / 6 = 5 CIs    Minimum amount of space required to
                     hold 30 sequential dependent
   ```

```
                        segments from one root.  For 850
                        roots, the minimum amount of space
                        required is 850 * 5 = 4250 CIs.
```

C. Factor into your calculations additional space to take into account:

- The "reorganization UOW", which is the same size as a regular UOW
- Two control data CIs allocated at the beginning of the root addressable part
- One control data CI for each 120 CIs in the independent overflow part

  Assuming a UOW size of 20 CIs, the minimum amount of space to be allocated is: 213 + 4250 + 20 + 2 + 1 = 4486 CIs.

2. Set up the DBD specifications according to the above results, and execute the DBD generation.

3. Allocate the VSAM cluster using VSAM Access Method Services.

   The following example shows how to allocate an area that would later be referred to as AREA1 in a DBDGEN:

```
DEFINE -
   CLUSTER -
     (NAME (AREA1) -
     VOLUMES (SER123) -
     NONINDEXED -
     CYLINDERS (22) -
     CONTROLINTERVALSIZE (1024) -
     RECORDSIZE (1017) -
     SPEED) -
   DATA -
     (NAME(DATA1)) -
     CATALOG(USERCATLG)
```

   The following keywords have special significance when defining an area:

   NAME
   > The name supplied for the cluster is the name subsequently referred to as the area name. The name for the data component is optional.

   NONINDEXED
   > DEDB areas are non-indexed clusters.

   CONTROLINTERVALSIZE
   > The value supplied, because of a VSAM ICIP requirement, must be 512, 1024, 2048, or 4096.

   RECORDSIZE
   > The record size is 7 less than the CI size. These 7 bytes are used for VSAM control information at the end of each CI.

   SPEED
   > This keyword is recommended for performance reasons.

   CATALOG
   > This optional parameter can be used to specify a user catalog.

4. Run the DEDB initialization utility (DBFUMIN0).

   This offline utility must be run to format each area to DBD specifications. Root-addressable and independent-overflow parts are allocated accordingly. The space left in the VSAM cluster is reserved for the sequential-dependent part. Up to 2048 areas can be specified in one utility run; however, the area initializations are serialized. After the run, check the statistical information report against the space calculation results.

5. Run the user DEDB load program

A BMP program is used to load the DEDB. The randomizing routine used during the loading of the DEDB might have been tailored to direct specific ranges of data to specific areas of the DEDB.

If the load operation fails, the area must be scratched, reallocated, and initialized.

Loading sequential dependent segments

If the order of sequential dependent segments is important, you must consider the way sequential dependents might be loaded in a DEDB.

The two alternatives are:

* Add a root and its sequential dependents.

  All the sequential dependents of a root are physically written together, but their physical order does not reflect the original data entry sequence. This reflection is not necessarily the way the application needs to view the dependent segments if they are being used primarily as a journal of transactions.

* Add all roots and then the sequential dependents.

  This technique restores the SDEP segments to their original entry sequence order. However, it requires a longer process, because the addition of each SDEP segment causes the root to be accessed.

Loading HALDBs that have secondary indexes

By default, when you perform an initial load of a HALDB database that has a secondary index, the secondary index is created as a part of the load process.

When source segments are loaded in HALDB databases, their secondary index entries are inserted in the secondary index partitions. The buffer pools that are used by the load program must include buffers for the secondary indexes. The secondary index partitions must be initialized before loading the database.

Initial loads do not create entries in the indirect list data set (ILDS), because all of the secondary index entries have accurate RBA pointers in their extended pointer sets (EPSs).

You can prevent secondary indexes from being created during an initial load of a HALDB database by specifying BLDSNDX=NO on the OPTIONS control statement in either the DFSVSAMP data set or the DFSVSM*xx* member of the IMS.PROCLIB data set. For example:

```
DFSVSAMP DD *
OPTIONS,BLDSNDX=NO
VSRBF=4096,500
IOBF=(8192,200)
/*
```

If you specify BLDSNDX=NO, you must build your secondary index by other means. You can use a tool, such as the IBM IMS Index Builder for z/OS.

When you use BLDSNDX=NO, the secondary index partitions are not authorized and their data sets are not allocated.

The use of BLDSNDX=NO and an index builder tool can shorten the load and index creation process, especially if you have many secondary index entries. When HALDB secondary indexes are created during the load process, the inserts in the secondary indexes are random, which can lengthen the load process considerably.

Index builder tools like the IMS Index Builder read the loaded database, create index entries without writing them to the secondary indexes, sort the entries in secondary index key sequence, and then write them to the secondary indexes in a sequential process.

Related tasks:

"Adjusting VSAM options specified in the OPTIONS control statement" on page 709

Related reference:

Defining VSAM performance options (System Definition)

Chapter 27. Database backup and recovery

The successful recovery of a database after a failure depends on the planning and preparation you do before an error ever occurs. Making database backup copies is a critical part of that preparation.

Related concepts:

"Backup and recovery of HIDAM and PHIDAM primary indexes" on page 181

"DL/I calls that can be issued against HD databases" on page 146

"Image-copy option" on page 475

"HALDB partition data sets and recovery" on page 187

Database failures

Every time IMS encounters a DL/I I/O error, IMS creates an extended error queue element (EEQE) that identifies the block or VSAM control interval in error.

IMS issues message DFS0451I or DFS0451A when there is an error in an IMS database.

When IMS closes a database, it automatically retries read and write errors on DL/I databases. If successful, forward recovery of the database is not required. Otherwise, forward recovery is eventually required. It might be possible to defer recovery to a more convenient time. Deferring recovery does not inhibit scheduling access or updating.

Using DEDB multiple area data sets also allows application programs to continue when I/O errors exist. For DEDB I/O errors, IMS issues messages DFS2571, DFS2572, DFS3712, and DFS3713. If a DEDB area is not available, the application receives an FH status code.

IMS maintains I/O information and buffer images across restarts. IMS does this by recording the EEQEs in DBRC, notifying all systems that are using the IRLM, and, during initialization and checkpoint, logging EEQEs and virtual buffers to the OLDS.

Related concepts:

"DL/I I/O errors and recovery" on page 625

➡ DFS messages (Messages and Codes)

Related tasks:

➡ DB - Database service aids (Diagnosis)

Database write errors

An IMS application program is unaware of a database write error. IMS does not pass a return code to the application program after a write operation completes or fails to complete.

Instead, when a write error occurs, IMS creates an extended error queue element (EEQE), allocates a buffer for the block or control interval in error, and writes a log record with information about the buffer.

IMS uses the buffer for all subsequent I/O to the block or CI, including database updates and read requests. When IMS closes the database, it retries the original write operation that failed; if it is successful, IMS frees the buffer, removes the corresponding EEQE from DBRC, and notifies all subsystems to disregard their corresponding EEQEs.

There is a limit to the number of EEQEs that IMS can create. For Fast Path areas, an area is stopped after 100 EEQEs. For full-function databases, the limit is 32,767 EEQEs. Before encountering this limit, you would likely encounter virtual storage limitations for the buffers needed to write that many EEQEs.

You can defer recovery for weeks or longer, but deferring recovery too long can increase the time that IMS requires to perform the recovery.

IMS notifies DBRC of each EEQE; DBRC records each EEQE in the DBDS record of the RECON data set.

Database read errors

When a read error occurs, IMS returns an A0 status code to the application program. IMS also creates an EEQE for the data in error but does not create a buffer for it.

Database quiesce

By quiescing a database, DEDB area, HALDB partition, or a database group, you can create a recovery point across databases and logs, create clean database image copies without taking a database offline, and improve the performance of the Database Change Accumulation utility (DFSUCUM0).

When a database is quiesced, no updates to the database are in progress, all prior updates have been committed and written to DASD, and application programs with new updates for the database are held in a wait state until the database is released from its quiesce state. The database is fully quiesced when the last in-progress update is committed in the database.

After the quiesced state is reached, DBRC records a deallocation (DEALLOC) time stamp in the allocation (ALLOC) record for the database, area, or partition in the RECON data set. The DEALLOC time stamp is used as the recovery point for recoveries and the recovery utilities, such as the Database Recovery utility (DFSURDB0) and the DFSUCUM0 utility.

Quiescing a database does not alter the database data sets. The quiesce function leaves the database data sets in the same state that they were in when the quiesce function starts.

Unlike other methods for creating recovery points, such as issuing the /DBRECOVERY command, quiescing a database requires only the issuance of a single command. IMS automatically places application programs in a wait state after any in-progress updates are committed and automatically restores access to the database when the quiesce is released. Creating a recovery point by using the /DBRECOVERY command requires stopping all database activity, unauthorizing and taking the database offline, and then restarting everything afterward.

The relative ease and speed of the database quiesce function, as well as its minimal impact on database availability, makes creating frequent recovery points less

expensive. Creating frequent points of recovery shortens the time required for recoveries and improves the performance of the Database Change Accumulation utility (DFSUCUM0) by reducing the number of log records the utility must merge.

Database types that support the quiesce function

You can quiesce Fast Path DEDB databases, full-function databases, database groups, DEDB areas, and HALDB partitions.

Specifically, the following types of databases support the quiesce function:
- DEDB
- HDAM
- HIDAM
- HISAM
- HSAM
- PHDAM
- PHIDAM
- PSINDEX
- SHISAM
- SHSAM

GSAM and MSDB databases do not support the quiesce function.

For DEDB databases, you can quiesce the entire database, which quiesces all of the areas in the DEDB, or you can quiesce a subset of one or more areas directly. Similarly, for HALDB PHDAM, PHIDAM, and PSINDEX databases, you can quiesce the HALDB master database, which quiesces all of the partitions in the database, or you can quiesce a subset of one or more partitions directly. When either a DEDB database or a HALDB database is quiesced at the database level, the quiesce status is maintained in the records for each area or partition rather than in the database record.

Because DBRC is required to manage access to the database and to record the time stamp used for recovery, quiescing a database that is not registered, although technically possible, is not recommended.

Quiesce options

When quiescing a database, you have the following options:
- You can quiesce a database just long enough to create a recovery point.
- You can hold a database in the quiesce state indefinitely to run either an image copy utility or the Database Change Accumulation utility (DFSUCUM0). To release a database that is held in the quiesce state, issue the appropriate UPDATE command with the STOP(QUIESCE) keyword specified.

  The UPDATE commands on which you can specify the STOP(QUIESCE) keyword include:
  - UPDATE AREA
  - UPDATE DATAGRP
  - UPDATE DB
- You can specify a timeout interval for IMS to wait for application programs to commit their updates before IMS cancels the quiesce process.

When you quiesce a database to create a recovery point, as soon as the database reaches a point of consistency, IMS immediately releases the database from the quiesce state and application programs can resume updating the database.

To momentarily quiesce a database, area, partition, or database group, for the purposes of creating a recovery point, specify the START(QUIESCE) keyword on the appropriate UPDATE command. For example: UPDATE DB NAME(DBXYZ) START(QUIESCE).

When you hold a database in the quiesce state, the database is held in the quiesce state until you release it by issuing the appropriate UPDATE command with the STOP(QUIESCE) keyword. While databases are held in the quiesce state, you can create clean image copies without having to take the database offline by running either an online image copy utility, such as the Database Image Copy 2 utility (DFSUDMT0), or a batch image copy utility, such as the Database Image Copy utility (DFSUDMP0).

To quiesce a database and hold it in the quiesce state, specify both the START(QUIESCE) and OPTION(HOLD) keywords on the appropriate UPDATE command. For example: UPDATE DB NAME(DBXYZ) START(QUIESCE) OPTION(HOLD)

The timeout interval for the quiesce function defines the amount of time that the quiesce function waits for application programs to commit any updates to the database that are in progress when the quiesce function is initiated. If updates remain in progress when the time interval expires, the quiesce of the database is canceled and the appropriate UPDATE command must be reissued with the START(QUIESCE) option to quiesce the database.

The default timeout value for the quiesce function is 30 seconds. You can define a different timeout value for the quiesce function by using the DBQUIESCETO parameter in the DFSCGxxx PROCLIB member. You can override the timeout value when starting the quiesce function by including the SET(TIMEOUT(nnn) parameter on the appropriate UPDATE command.

Impact of quiescing databases on application programs

While a quiesce of a database, area, or partition is in progress or is being held, application programs can be scheduled, but can access the database, area, or partition for reads only. Application programs that need to access the database for an update are held in a wait state until the quiesce is released.

If an application program is updating the database when a quiesce is initiated, the quiesce function waits until the application program commits the updates in progress and the updates are written to DASD. If, after committing the in-progress updates, the application program has more updates for the database, the application program is placed in a wait state until the quiesce is released. If the application program does not commit its updates before the timeout interval specified for the quiesce function expires, the quiesce of the database is canceled.

Holding a database in a quiesce state can cause the number of application programs waiting in dependent regions to increase, potentially causing the number of dependent regions to reach the maximum number allowed by your installation.

To avoid problems associated with a large number of waiting application programs, hold databases in a quiesce state only during periods of low activity

and only for as long as is needed to perform required tasks, such as creating an image copy.

DBRC, the RECON data set, and the quiesce function

To ensure the coordination of the quiesce function in an IMSplex, the DBRC instances that belong to the IMS systems participating in the quiesce function must be in the same DBRC group, as defined by a unique DBRC group ID, and the DBRC instances must be registered with the Structured Call Interface (SCI) of the Common Service Layer (CSL).

DBRC records the quiesce status of a database, DEDB area, or HALDB partition in the RECON data set. When the quiesce function is invoked, DBRC updates the RECON data set to indicate that a quiesce of a database, area, or partition is in progress. When a database, area, or partition is quiesced and then held in the quiesce state, DBRC updates the RECON data set to indicate both that a quiesce of the database, area, or partition is in progress and that the quiesce state is being held.

While a database, area, or partition is being quiesced, a listing of its record in the RECON data set shows `QUIESCE IN PROGRESS`. While a database, area, or partition is held in a quiesce state, a listing of the record in the RECON data set shows both `QUIESCE IN PROGRESS` and `QUIESCE HELD`.

While a quiesce of a database, area, or partition is in progress, but the quiesce state is not being held, DBRC manages authorization and access to the database, area, or partition as follows:
- Not authorized:
  - Utilities that require an access intent of update or exclusive
  - Image copy utilities
  - Batch application programs with update access
- Authorized, but cannot access the database, area, or partition:
  - IMS systems that are not participating in the quiesce
  - Online IMS systems that have application programs that have update privileges
- Authorized and can access the database, area, or partition:
  - Batch application programs with read-only intent
  - Online IMS systems that have application programs that have read-only privileges
  - Utilities that require read-only intent (excluding image copies)

When a database, area, or partition is being held in the quiesce state, DBRC manages database, area, and partition authorization and access as follows:
- Not authorized:
  - Utilities that require an access intent of update or exclusive
  - Batch application programs with update authority
- Authorized, but cannot access the database, area, or partition:
  - IMS systems that are not participating in the quiesce
  - Online IMS systems that have application programs that have update authority
- Authorized and can access the database, area, or partition:

- Utilities that require read-only access authority
- Image copy utilities
- Batch application programs with read-only authority, as specified by a value of G, GO, or GOx on the PROCOPT parameter of the PCB statement for the application program.
- Online IMS systems that have application programs that have read-only authority

When the database is quiesced, if the database has open ALLOC records in the RECON data set, the ALLOC records are closed with a DEALLOC time stamp that corresponds to the point the quiesce was achieved across the IMSplex. The ALLOC record indicates that the database was deallocated as a result of being quiesced. The DEALLOC time stamp in the closed ALLOC record can then be used as a recovery point for the Database Recovery utility (DFSURDB0) and time stamp recoveries.

When a new ALLOC record is created for the database, area, or partition, a new update set ID (USID) and data set sequence number (DSSN) are assigned.

When the quiesce is released, the database can be accessed again by DL/I calls. For DEDB areas, a new ALLOC record is created in the RECON data set. For full-function databases, a new ALLOC record is not created until the database is accessed by the first update.

Requirements and restrictions for the quiesce function

The database quiesce function requires an IMSplex environment with the following CSL managers enabled.
- Operations Manager (OM)
- Resource Manager (RM), unless the IMSplex environment includes only a single IMS system and RMENV=N is specified in either the DFSCGxxx PROCLIB member or the CSL section of the DFSDFxxx PROCLIB member.
- Structured Call Interface (SCI)

The minimum version specified for the RECON data set must be 11.1 or later by using the MINVERS keyword on the DBRC command CHANGE.RECON.

A quiesce fails to start in any of the following circumstances:
- A batch application program with update authority is accessing the database when the quiesce function is started; however, a batch application program that has read-only authority does not cause the quiesce function to fail.
- The database needs to be recovered or backed out.
- A reorganization intent flag is set in the database, area, or partition record in the RECON data set.
- The HALDB online reorganization function has been initiated and either the target HALDB partition is in cursor-active state, or an IMS system owns the Online Reorganization function for a partition being quiesced.
- A quiesce of the database is already in progress.
- The HALDB partition being quiesced needs to be initialized.
- Any one of the following commands were issued against the database and have not completed processing.
  - /DBD DB

- – /DBR AREA
- – /DBR DB
- – /INIT OLREORG
- – INIT OLREORG
- – /STA AREA
- – /STA DB
- – /STO AREA
- – /STO DB
- – UPDATE AREA
- – UPDATE DATAGRP
- – UPDATE DB

Related concepts:

"Non-concurrent image copies" on page 598

Related tasks:

"Using database change accumulation input for recovery" on page 610

Related reference:

↪ UPDATE commands (Commands)

Making database backup copies

This topic explains how to make backup copies of your databases. It describes the utilities and how they affect IMS operations.

Related concepts:

↪ Message queue backup copies (System Administration)

↪ System data set backup copies (System Administration)

Image copies and the IMS image copy utilities

Backup copies of databases are called image copies. You can create image copies by using one of the image copy utilities provided by IMS. Depending on which utility you use, you can create image copies while databases are online, offline, or quiesced.

IMS provides three utilities that you can use to make image copies:
- Database Image Copy utility (DFSUDMP0)
- Online Database Image Copy utility (DFSUICP0)
- Database Image Copy 2 utility (DFSUDMT0)

Restriction: You cannot run these utilities against a HALDB partition that is being reorganized by HALDB Online Reorganization (OLR).

These utilities create image copies of recoverable and nonrecoverable databases. The image copy utilities operate on data sets.

The frequency of creating image copies is determined by your recovery requirements. The minimum requirement is that an image copy be created immediately after a database is reorganized, reloaded, or initially loaded. Because database recovery is done on a physical replacement basis, a reloaded data set is not physically the same as it was before unload.

With the Database Image Copy utility (DFSUDMP0) and the Online Database Image Copy utility (DFSUICP0), if a database includes multiple data sets or areas, you must supply the utility with multiple specifications. With the Database Image Copy 2 utility, however, you can copy multiple database data sets in one execution of the utility. You can specify a group name to represent the collection of database data sets that are to be copied in a single execution.

Use the DBRC commands INIT.DB and CHANGE.DB to identify recoverable databases to DBRC. DBRC works similarly with all of the image copy utilities.

The output data sets from the Database Image Copy utility (DFSUDMP0) and the Online Database Image Copy utility (DFSUICP0) have the same format. The rules for predefinition and reuse of image copy data sets apply to the data sets that are used by both of these utilities.

If you use the Database Image Copy 2 utility, the format of the output data sets depends on which image copy option you use. Image copies that are produced by using the concurrent copy function of the Database Image Copy 2 utility are in DFSMSdss dump format. Image copies that are produced by using the fast replication function of the Database Image Copy 2 utility are exact copies of the original data set.

If you use the Database Image Copy 2 utility, the DBRC rules for predefinition and reuse of image copy data sets apply only to the data sets that are used for the concurrent copy function. The fast replication function of the Database Image Copy 2 utility does not support the predefinition and reuse of image copy data sets.

All of these utilities call DBRC to verify the input (DBRC allows them to run only if the input is valid), and they call DBRC to record information in the RECON data set about the image copy data sets that they create. An image copy record in the RECON data set has the same format regardless of which utility created its corresponding image copy data set. The Online Database Image Copy utility, however, has its own PDS member of skeletal JCL, and you use a separate DBRC command, GENJCL.OIC, to generate the job for the Online Database Image Copy utility.

Recommendation: Copy all data sets or areas for a database at the same time. When you recover a database to a prior state, you must recover all data sets that belong to the database, as well as all logically related databases (including those related by application processing) to the same point to avoid data integrity problems.

When using the image copy utilities, you have the option of creating one or more output image copies, unless you are using the fast replication function of DFSUDMT0. Advantages of making two or more copies are:
* If an I/O error occurs on one copy, the utility continues to completion on the others.
* If one copy cannot be read, you can perform recovery using another.

The trade-off in deciding whether to make more than one copy is that the performance of the image copy utility is degraded by the time required to write the other copies.

If you are using the fast replication option of the Database Image Copy 2 utility, the utility can make only a single output image copy of each source database data set during each execution of the utility.

The utilities enforce a minimum output record length of 64 bytes; thus, it is possible that an image copy of a database with a very short logical record length can require more space than the original database.

Recommendation: Take an image copy immediately after running batch jobs that update the database without IMS logging. Even when pool sizes are identical and your system uses VSAM background write, updates made by batch jobs are not bit-for-bit repeatable. Taking an image copy allows you to maintain the integrity of your database if a recovery is required.

If you recover using log tapes up to the start of a batch job, and then reprocess the batch job, the resulting database might not be bit-for-bit identical to the database after the previous batch run, although it is logically identical. If the database is not bit-by-bit identical, log tapes created after the previous execution of the batch jobs would not be valid after the reprocessing. Therefore, do not attempt a recovery by starting with an image copy, applying log tapes, reprocessing unlogged batch executions, and then applying more log tapes.

Image copies that are not created by one of the IMS image copy utilities are called *nonstandard* image copies or *user* image copies.

Related tasks:

"Nonstandard image copy data sets" on page 604

Concurrent image copies

IMS allows you to create image copies of databases that are registered with DBRC while the database is being updated.

An image copy taken while a database is being updated is called a *concurrent image copy* or, because the copy represents the state of the database over a period of time rather than at an instant in time, a *fuzzy image copy*. When you create a concurrent image copy, your final copy might include some, all, or none of the updates made to a database during the copy process.

The IMS Database Image Copy 2 utility (DFSUDMT0) provides a concurrent copy option, which you should not confuse with a concurrent image copy. The concurrent copy option of the DFSUDMT0 utility can create either concurrent image copies or clean image copies. The concurrent copy option of the DFSUDMT0 utility takes its name from the CONCURRENT keyword of the z/OS DFSMSdss DUMP command and does not refer to the type of image copy the option creates. DFSMS and the Concurrent Copy feature of 3990 hardware can make copies of a data set while the data set is offline or online.

The fast replication option of the DFSUDMT0 utility can also create either concurrent or clean image copies.

Restrictions:

- The IMS image copy utilities can make a concurrent image copies only of databases that are registered with DBRC.
- You can make copies of nonrecoverable databases, but they must be stopped before you run the utility to make the image copies. You cannot take concurrent image copies of nonrecoverable databases while they are online because IMS

does not log changes to them. The database itself becomes fuzzy if a fuzzy image copy of a nonrecoverable database is used to recover the database.

- Using the Database Image Copy utility (DFSUDMP0), you can only make concurrent image copies for OSAM and VSAM Entry Sequenced Data Set (ESDS) DBDSs; VSAM Key Sequenced Data Set (KSDS) DBDSs are not supported for concurrent image copy. If you use either of the other two image copy utilities, you can create image copies of ESDSs or KSDSs.

Related tasks:

"Concurrent image copy recovery" on page 625

Non-concurrent image copies

Non-concurrent image copies are image copies that are taken while no updates are being made to the database.

Non-concurrent image copies are also referred to as batch image copies, because they are often created when the database is offline, or clean image copies because they do not contain any fuzzy data. All of the data reflected in the image copy has been committed and the image copy by itself can be used as the starting point for recovery. Only log records for updates to the data set that occur after the image copy is made are needed to recover the data set.

You can create non-concurrent image copies in two ways:

- By quiescing the database, area, or partition with the quiesce and hold option specified and running one of the image copy utilities. Databases, areas, and partitions are held in the quiesce state by issuing the appropriate UPDATE command with the START(QUIESCE) OPTION(HOLD) keywords. If a batch image copy utility is used, the DD statement for the database must specify DISP=SHR. After the image copy is complete, the quiesce must be released with another UPDATE command with the STOP(QUIESCE) keyword specified.
- By taking the database offline and running a batch image copy utility. This method requires stopping access to the database, deallocating it, and unauthorizing it with DBRC by issuing the appropriate UPDATE command with the STOP(ACCESS) keyword specified.

Of the two methods for creating non-concurrent image copies, using the quiesce function is the easier and faster method; however, because application programs are held in their dependent regions in a wait state, databases should be held in the quiesce state only during periods of low database activity.

Related concepts:

"Database quiesce" on page 590

Fast replication image copies

You can create fast replication image copies by using the fast replication option of the Database Image Copy 2 utility (DFSUDMT0).

The DFSUDMT0 utility uses the z/OS DFSMSdss `COPY` command with the `FASTREPLICATION` keyword.

By using the fast replication option of the DFSUDMT0 utility, you can create image copies more quickly than other image copy methods. You can create clean or concurrent fast replication image copies and register them with DBRC for use by the Database Recovery utility (DFSURDB0) during recovery procedures.

Fast replication image copies are exact copies of the input data sets and are not formatted like image copies taken by the Image Copy 2 utility using the DFSMSdss `DUMP` command with the `CONCURRENT` keyword.

DFSMSdss fast replication requires hardware support of either the FlashCopy® feature of the IBM Enterprise Storage Server® (ESS) or the SnapShot feature of the IBM RAMAC Virtual Array (RVA) storage system.

The Image Copy 2 utility is the only IMS image copy utility that supports fast replication image copies.

Related reference:

▶ Database Image Copy 2 utility (DFSUDMT0) (Database Utilities)

Recovery after image copy

The Database Image Copy utility (DFSUDMP0) and the Database Image Copy 2 utility (DFSUDMT0) copy data sets for HISAM, HIDAM, HDAM, PHDAM, PHIDAM databases, and areas for DEDBs.

When you perform a subsequent recovery, what you need as input to the recovery depends on whether the copies are concurrent or not:

- For non-concurrent copies, you need only the image copy and those logs created after the database is restored to the online system.
- For concurrent copies, you need the image copy and logs created before and after the database is restored to the online system. DBRC helps you decide which logs you need.

Recommendation: When you run the Database Image Copy utility (DFSUDMP0) for databases and areas (without specifying the `CIC` on the `EXEC` statement for the utility), no other subsystem should update the databases. You can prevent updates to full function databases by issuing the /DBDUMP DB or UPDATE DB STOP(ACCESS) command. You can prevent updates to DEDB areas by issuing either the /STOP AREA or UPDATE AREA STOP(SCHD) command.

The Online Database Image Copy utility (DFSUICP0) runs as a BMP program. You can use it for HISAM, HIDAM, and HDAM databases only. If IMS updates these databases while the utility is running, IMS requires all logs for any subsequent recovery, including the log in use when you started the utility. IMS requires the logs because the image copy is not an image of the database at any one time.

HSSP image copies

If you use the image copy option of HSSP, IMS creates image copies of DEDB areas for you.

The image copy contains the "after images" of the HSSP PCB, and is a fuzzy copy. IMS logs all other PCB database changes as usual. During database recovery, you must apply any concurrent updates since the start of the image copy process. A fuzzy image copy can be created if a non-HSSP region updated the same DEDB area during the image copy process.

Definition: *Fuzzy* means that there might have been concurrent updates during the image copy process.

If, however, the image copy process does not complete successfully, this data set is simply returned to the set of available HSSP image copy data sets.

IMS uses the QSAM access method to create HSSP image copies. The primary allocation of image copy data sets must be larger than (or equal to) the DEDB area data set size.

IMS treats HSSP image copies like concurrent image copies, so you can use the Database Recovery utility (DFSURDB0) without telling it that the image copy is an HSSP image copy.

Restriction: You can only make an HSSP image copy if a database is registered with DBRC. Furthermore, you must initialize the image copy data sets using the INIT.IC command.

Related tasks:

"HSSP image copy recovery" on page 625

Creating image copy data sets for future use

You can allocate image copy data sets before you need them.

IMS automatically selects these data sets for use as output data sets when you use either of the DBRC commands GENJCL.IC or GENJCL.OIC to generate a job for the Database Image Copy utility or Online Database Image Copy utility.

Restriction: The fast replication function of the Database Image Copy 2 utility (DFSUDMT0) does not support the REUSE parameter of the INIT.DBDS command. Therefore, you cannot create data sets for the fast replication function of the Database Image Copy 2 utility before you need them.

When you use the DBRC command INIT.DBDS to identify a DBDS or area in the RECON data set, you can specify any of the following keywords for that DBDS or area:

GENMAX
> Use this keyword to specify how many image copy data sets you want DBRC to maintain information about for the DBDS or area.
>
> You can maintain a certain number of image copy data sets for physical recovery. DBRC keeps a record of a specified number of the most recent image copy data sets. This number is the value you specify for the GENMAX keyword for this DBDS; duplicate image copy data sets are not included in this number.

RECOVPD
> Use this keyword to maintain data for a certain period.

REUSE
> Use this keyword to inform DBRC that you want to define image copy data sets and record them in the RECON data set for future use. DBRC records these available image copy data sets, with their names, unit types, and volume serial information, in the RECON data set, selects them during the processing of a GENJCL.IC command, and uses this information in the appropriate DD name of the job for the Database Image Copy utility.

NOREUSE
> Use this keyword to inform DBRC that you do not want to use an existing data set, and that you want to provide the data set name for the output image copy data set that the Database Image Copy utility is to use. Specify the data set name in either the JCL partitioned data set (PDS) member that the DBRC uses to process the GENJCL.IC command or in the job you produce. When you specify NOREUSE, DBRC dynamically sets the unit type of the output image

copy data set to the default unit type for the device (as specified in the INIT.RECON and CHANGE.RECON commands).

When the Database Image Copy utility uses an available image copy data set, DBRC adds a current time stamp to its record in the RECON data set.

Related tasks:

"Recovery period of image copy data sets"

Recovery period of image copy data sets

The *recovery period* is the amount of time before the current date for which DBRC maintains recovery information in the RECON data set.

For example, if the recovery period of a DBDS or area is 14 days, DBRC maintains sufficient recovery-generation information for at least 14 days.

You specify the recovery period using the GENMAX and RECOVPD keywords of the INIT.DBDS and CHANGE.DBDS commands. The examples in these topics describe the effects of various DBRC keywords on the recovery period.

Related tasks:

"Creating image copy data sets for future use" on page 600

Example 1: recovery period of image copy data sets

Example 1 describes what happens when the recovery period specified by the RECOVPD parameter has not been exceeded, but the number of image copy data sets has reached the GENMAX value, and REUSE=Y.

The following table shows the DBRC keywords and their values for this example:

REUSE	GENMAX	RECOVPD
Y	Reached	Not Exceeded

If there are available image copies:
- IMS uses an available image copy.
- IMS issues message DSP0065I, indicating that the predefined image copy has been used.

If the number of predefined data sets equals the maximum number of generations:
- IMS cannot reuse the oldest image copy.
- IMS issues message DSP0063I, indicating that the image copy within the recovery period cannot be reused.
- IMS stops processing.

Example 2: recovery period of image copy data sets

Example 2 describes what happens when the recovery period specified by the RECOVPD parameter has been exceeded, the number of image copy data sets is under the GENMAX value, and REUSE=Y.

The following table shows the DBRC keywords and their values for this example:

REUSE	GENMAX	RECOVPD
Y	Not Reached	Exceeded

Processing continues as described in "Reusing image copy data sets" on page 603, except that IMS uses the available image copy data set.

Example 3: recovery period of image copy data sets

Example 3 describes what happens when the recovery period specified by the RECOVPD parameter has been exceeded, the number of image copy data sets has reached the GENMAX value, and REUSE=N.

The following table shows the DBRC keywords and their values for this example:

REUSE	GENMAX	RECOVPD
N	Reached	Exceeded

IMS deletes the oldest image copy data set record that exceeds the RECOVPD value.

Example 4: recovery period of image copy data sets

Example 4 describes what happens when the recovery period specified by the RECOVPD parameter has not been exceeded, the number of image copy data sets has reached the GENMAX value, and REUSE=N.

The following table shows the DBRC keywords and their values for this example:

REUSE	GENMAX	RECOVPD
N	Reached	Not Exceeded

If the number of image copy data sets used has reached the GENMAX value, IMS cannot delete the oldest image copy data sets within the recovery period. In this case:

- IMS issues message DSP0064I, indicating that an image copy data set within the recovery period cannot be deleted.
- Processing continues and DBRC records a new image copy data set in the RECON data set.

Example 5: recovery period of image copy data sets

Example 5 describes what happens when the recovery period specified by the RECOVPD parameter has been exceeded, the number of image copy data sets is under the GENMAX value, and REUSE=N.

The following table shows the DBRC keywords and their values for this example:

REUSE	GENMAX	RECOVPD
N	Not Reached	Exceeded

DBRC records a new image copy data set in the RECON data set. Even though the oldest image copy is beyond the recovery period, it will not be deleted because the GENMAX has not been reached.

Other recovery period considerations

If you issue a CHANGE.DBDS command and specify a new value for GENMAX that is smaller than the existing value, IMS will record the value, regardless of whether the oldest image copies cannot be deleted because they are within the recovery period (RECOVPD).

When you issue the DELETE.IC command, IMS deletes any specified image copy data sets, regardless of whether the RECOVPD value has been exceeded.

Reusing image copy data sets

DBRC allows you to reuse old image copy data sets.

The REUSE keyword of the INIT.DBDS command, in addition to allowing you to define image copy data sets for future use, allows DBRC to reuse image copy data sets. To reuse the image copy data set means that DBRC uses the same name, volume, physical space, and record in the RECON data set for the new image copy data set as for the old one.

Restriction: You cannot use the REUSE keyword for data sets that will be copied using the fast replication function of the Database Image Copy 2 utility. The fast replication function does not support the reuse of data sets.

When you run one of the image copy utilities, IMS automatically reuses the oldest image copy data set for a DBDS or area, if it can, when both of the following conditions are met:

- The RECON data set has records for a number of image copy data sets equal to the current GENMAX value. To see the current GENMAX value, use the LIST.DBDS command.
- The oldest image copy is beyond the recovery period.

When you use the GENJCL.IC command to generate the job for the Database Image Copy utility or Database Image Copy 2 utility, IMS automatically selects the image copy data set to be reused. If the number of image copy data sets is less than the GENMAX value, and all image copy data sets have been used, you must define more image copy data sets for the DBDS or area before running the Database Image Copy utility or Database Image Copy 2 utility. The number of image copy data sets should be greater than the GENMAX value if you want to use a recovery period.

If you do not allow IMS to reuse image copy data sets, but the GENMAX value has been reached and the RECOVPD has been exceeded, when you run the Database Image Copy utility or Database Image Copy 2 utility, DBRC selects a new image copy data set and deletes the record in the RECON data set with the oldest time stamp. IMS does not scratch the image copy data set itself. You must scratch the data set yourself or keep track of it, because DBRC is no longer aware of its existence.

HISAM copies (DFSURUL0 and DFSURRL0)

The HISAM Reorganization Unload utility (DFSURUL0) can make backup copies of an entire HISAM database while it reorganizes the database, all in one pass.

Because the unload utility (DFSURUL0) reorganizes the database, you must, before resuming normal online operations, reload the data set using the HISAM Reorganization Reload utility (DFSURRL0), as shown in the following figure. If you do not reload the data set, the logging done after unload but before reload reflects the old organization of the data set. Therefore, if you need to use that log to recover the data set in the future, the organizations will not match, and the data set's integrity will be destroyed.
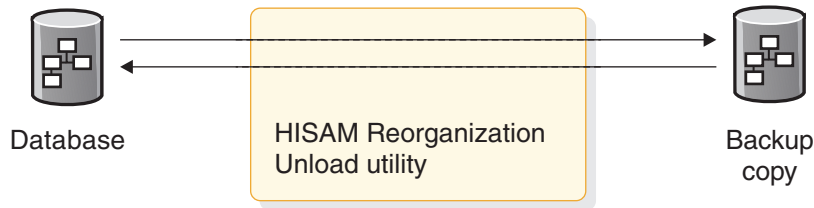
Figure 243. Making a backup copy with HISAM unload

When using the HISAM utility to make a backup copy, you must reload immediately, or the actual database will not match the backup database. The reload utility (DFSURRL0) uses the output of its reorganization step as input to the reload step as if it were an image copy data set. After reorganizing a database, you must make an image copy of it before you can authorize the DBDS.

Before reorganizing a shared database, you must prevent other subsystems from being authorized during the reorganization process by:

- Issuing a global /DBRECOVERY DB or UPDATE DB STOP(ACCESS) command for the database to be reorganized. The /DBRECOVERY DB GLOBAL command prevents further authorizations except for reorganization and recovery utilities.
- Issuing the CHANGE.DB command with the NOAUTH keyword to manually update the RECON data set. This command prevents future authorizations except for the reorganization and recovery utilities. After the reorganization is complete, manually update the RECON data set by issuing the CHANGE.DB command with the AUTH keyword for the database that was just reorganized.

Recommendation: Ensure that recovery utilities do not run during the reorganization.

Nonstandard image copy data sets

Image copy data sets that are not created by using one of the IMS image copy utilities are referred to as *nonstandard image copy data sets* or *user image copy data sets*.

For example, you can use the IBM Device Support Facilities (ICKDSF) product to make a copy of the volume on which a DBDS resides.

If you do not use one of the IMS utilities to make image copies, you must keep track of the data set names of the image copies and the tools that are used to make the copies because IMS does not track this information about nonstandard image copies.

Failure to copy the data set correctly can result in the failure of the database recovery process. For a KSDS, any utility which preserves the KSDS data content can be used to create the copy (IMS accesses a KSDS by key, and therefore has no dependency upon RBA sequence within the KSDS). For all other data set types, the physical sequence of the content of the data set must be preserved. Be sure to use only utilities that copy the data set correctly, or use the IMS provided Image Copy utilities to create a standard image copy.

DBRC does not automatically record the existence of nonstandard image copy data sets in the RECON data set; you must do so by using the NOTIFY.UIC command. If you do not record this information in the RECON data set, DBRC might

misinterpret subsequent information about changes to the DBDS. The NOTIFY.UIC command supports both batch and concurrent nonstandard image copies.

Restriction: When you use the NOTIFY.UIC command, you cannot specify the REUSE keyword of the INIT.DBDS command.

Before you recover a DBDS or DEDB area by using a nonstandard clean image copy data set, you must restore the DBDS or DEDB area from the nonstandard image copy data set. However, the process for completing recovery varies depending on whether the nonstandard image copy is a batch image copy or a concurrent image copy.

Related concepts:

"Image copies and the IMS image copy utilities" on page 595

Recovery from a batch nonstandard image copy

If you are using a batch nonstandard user image copy, before you can run the Database Recovery utility, but after you restore the DBDS or DEDB area from the image copy, you must issue the NOTIFY.RECOV command to notify DBRC that you restored the data set or area.

If the time stamp of the nonstandard image copy is within the range of an existing time stamp recovery, the NOTIFY.RECOV command fails. If the NOTIFY.RECOV command is successful, you can then run the Database Recovery utility to add changes that occurred since the time stamp of the nonstandard image copy data set. DBRC provides and verifies JCL only for the sources of change records to be applied to the already-restored DBDS or DEDB area.

If you are using a batch nonstandard image copy and you use the GENJCL.RECOV command to generate the recovery JCL, include the USEDBDS parameter to indicate that no image copy data set is to be included in the generated JCL.

Recovery from a concurrent nonstandard image copy

Before you recover the DBDS or DEDB area with a concurrent nonstandard image copy data set, issue the CHANGE.DBDS command with the RECOV parameter to set 'recovery needed' status to indicate that the DBDS or DEDB area is unavailable for use.

After the DBDS or DEDB area has been restored from the image copy, run the Database Recovery utility to add the changes that occurred since the time stamp of the nonstandard image copy data set. DBRC provides and verifies JCL only for the sources of change records to be applied to the already-restored DBDS or DEDB area.

If you are using a concurrent nonstandard image copy and you use the GENJCL.RECOV command to generate the recovery JCL, do not include the USEDBDS parameter; in this case, a flag in the DBRC RECON data set indicates that the image copy data set is to not to be included in the generated JCL.

When you recover the database data set, you must indicate the runtime when the nonstandard concurrent image copy was taken. You can do this by specifying either the USERIC(*time_stamp*) keyword or the LASTUIC keyword of the GENJCL.RECOV command, which generates a JCL stream for the Database Recovery utility that adds the changes since the nonstandard concurrent image copy data set was created. DBRC generates and verifies JCL only for the sources of change records to be applied to the already-restored DBDS or DEDB area.

Because the Database Recovery utility does not use an image copy for processing the change records, DBRC does not allow the Database Recovery utility to process any log that contains changes outside the recovery range. The recovery range is defined by the time-stamp recovery record's RECOV TO (image copy time) and RUNTIME values.

Recommendation: Close the database by using the /DBRECOVERY command (without the NOFEOV keyword) or the UPDATE DB STOP(ACCESS) OPTION(FEOV) command before running the image copy utility.

Frequency and retention for backup copies

When developing a backup strategy for your databases you need to consider how frequently to make new copies and how long to keep old, back-level copies.

There are no precise answers to these questions. Generally, the more frequently you copy, the less time recovery takes. The farther back in time your old copies go, the farther back in time you can recover; remember that program logic errors are sometimes not discovered for weeks. However, making each new copy requires work, and each old copy you save uses additional resources.

The only firm guidelines are these:
* If you do an initial load of a database, immediately make a backup copy of it.
* If a database is composed of several data sets, be sure to copy all data sets at the same time.
* If you reorganize a database, immediately make a new backup copy of it.

  Exception: It is not necessary to make backup copies after DEDB online reorganizations.

You can reduce the amount of work required to create non-concurrent image copies by taking the image copies while databases are quiesced instead of while databases are offline. Quiescing a database leaves the database online, allocated, and authorized with DBRC. Application programs are placed in a wait state. Because quiescing databases requires less work, it is possible that you can increase the frequency with which you create image copies.

If you take databases offline to create non-concurrent image copies, the databases are taken offline, deallocated, and unauthorized with DBRC. Application programs encounter an unavailable database.

Related tasks:

Recovery of databases

If a database is physically lost or damaged in such a way that records in it become inaccessible, you can reconstruct the database from the information you have been keeping: image copies, logs, and so forth. This type of recovery is known as forward recovery.

Definition: *Forward recovery* involves reconstructing information and reapplying it to a backup copy of the database. It is based on the notion that if you knew what the data was like at one time and you know what changes have been made to it since then, you can process the data to return the database to the state it was in just before it was lost.

The following figure illustrates this concept. You know what the database was like at one time, because you made a backup copy of it. You know what changes (additions, deletions, alterations) have been made to the database since the backup copy was made, because IMS has been recording these changes on the log. Therefore, you only need to combine the two, and create a new data set to replace the database you have lost.

| Backup copy | Changes since backup copy was made (recorded on SLDS) | Reconstructed database |
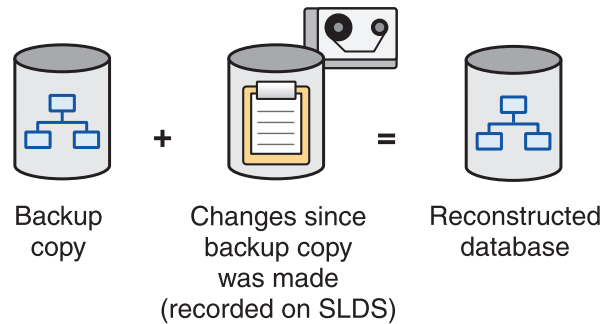
Figure 244. Database reconstruction (forward recovery)

IMS provides the Database Recovery utility (DFSURDB0) to perform forward recovery. The following figure illustrates how the utility works.

Additionally, if you have the IBM IMS Database Recovery Facility for z/OS product installed, IMS provides the /RECOVER command to invoke the tool from within IMS.

SLDS (possibly accumulated),
RLDS (possibly accumulated), and/or
DSLOG (DBRC use only)

Most recent
image copy

DBD
library

Control
statement

Database
Recovery
utility

RECON
data set
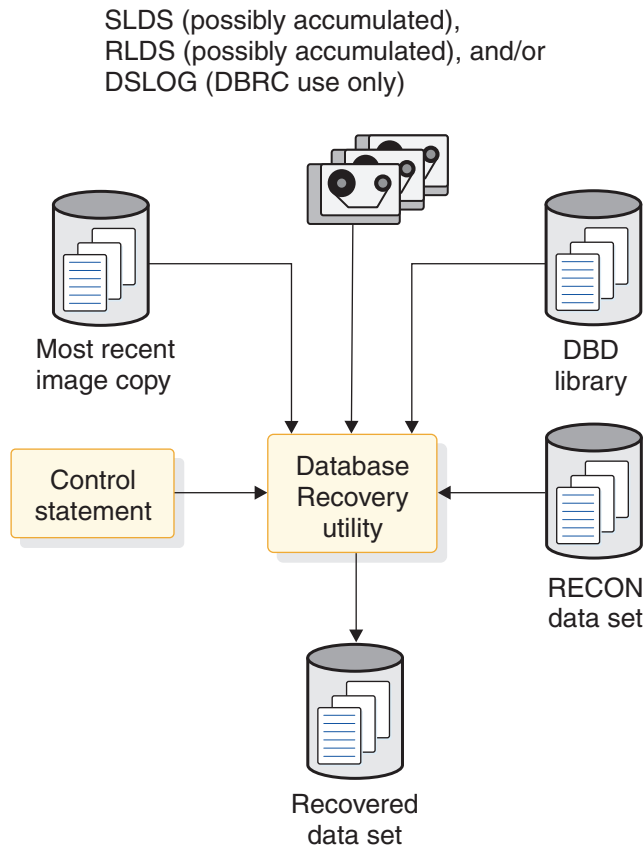
Recovered
data set

Figure 245. Recovering a database using the IMS Database Recovery utility (DFSURDB0)

Restriction: DBRC is unable to generate JCL for recovering databases that are not registered with DBRC. You can use the `LIST.LOG` command (or `DSPAPI FUNC=QUERY` API request) to produce a listing of current OLDSs, RLDSs, and SLDSs. Using this command lets you check manually that the JCL you are using for change accumulation or database recovery contains DD statements for all active log data sets. You can also use the `LIST.LOG` command (or `DSPAPI FUNC=QUERY` API request) when you use DBRC batch log control.

Recommendation: Do not supply logs or change accumulation input when running the Database Recovery utility for nonrecoverable DBDSs.

The Database Recovery utility and database recovery service perform database recovery at the data set level. Usually, only a single data set of the database requires recovery. If more than one data set has been lost or damaged, you need to recover each one separately if you are using the Database Recovery utility. If you are using the database recovery service, all the lost or damaged data sets can be recovered in one recovery operation.

With DEDBs, using multiple area data sets reduces the need for recovery. If one copy has an I/O error, you can create another copy online and drop the error copy.

Related tasks:

"Recovering HALDB databases when enabling partitions" on page 827

 Recovering databases (System Administration)

Related reference:

 Database Recovery utility (DFSURDB0) (Database Utilities)

Recovery and data sets

The Database Recovery utility (DFSURDB0) performs database recovery at the data set level. If more than one data set has been lost or damaged and you are using the Database Recovery utility, you need to recover each data set separately.

If you are using the IMS Database Recovery Facility for z/OS, all the lost or damaged data sets can be recovered in one recovery operation.

With DEDBs, using multiple area data sets reduces the need for recovery. If one copy of the area data set has an I/O error, you can create another copy online and drop the error copy.

If your database data sets are multi-volume OSAM data sets, make certain you are using the appropriate allocation specifications to ensure that all volumes get a primary extent.

In some cases, you do not need to delete and redefine data sets prior to recovering them. In fact, in cases where the deletion and redefinition of data sets is optional, doing so unnecessarily slows the recovery process.

You are required to delete and define data sets during the recovery process in the following cases:
- You are recovering VSAM database data sets by using the Database Recovery utility.
- You are recovering the output data sets of the integrated HALDB Online Reorganization function and the data sets are in a cursor active state.
- You are rebuilding a HALDB indirect list data set (ILDS) by using the HALDB Index/ILDS Rebuild utility.

You are not required to delete and define data sets during the recovery process in the following cases:
- You are recovering OSAM database data sets by using the Database Recovery utility and you include image copies as input to the utility.
- You are rebuilding a HALDB PHIDAM primary index by using the HALDB Index/ILDS Rebuild utility.

In cases where you are required to delete and redefine the data sets, when you must execute the delete and define step in the recovery process depends on the circumstances of the recovery. For example:
- If you are including an image copy as input to the Database recovery utility, delete and define the data set before you run the utility.
- If you are not including an image copy as input to the Database recovery utility, such as when you apply an image copy to the database in a prior step, delete and define the data set before applying the image copy.
- If you are executing the recovery JCL automatically when the GENJCL.RECOV command is issued, delete and define the data set before you issue the GENJCL.RECOV command.

Related tasks:

"Allocating OSAM data sets" on page 569

Planning your database recovery strategy

You can recover databases in many different ways.

The simplest, although possibly the most time-consuming, approach is to use an image copy plus all log data sets created since that image copy. You can also:

- Use the database recovery service
- Use the Change Accumulation utility to reduce the amount of log input to the Database Recovery utility
- Create an RLDS with the Log Archive utility and use the RLDS instead of an SLDS

The Database Recovery utility always processes log tapes **after** it processes the change accumulation data set. If, therefore, you must include log tapes that were created **before** the change accumulation data set, you must execute DFSURDB0 twice—once with the image copy and log tapes, and again with the change accumulation data set.

If you register your databases with DBRC, DBRC ensures the correct input is provided to the Database Recovery utility, and optionally generates the necessary JCL. For more detailed information on DBRC, see *IMS Version 15 System Administration*.

Some of the advantages and disadvantages of the different recovery techniques are described in this topic.

Related concepts:

"Frequency and retention for backup copies" on page 606

Using database change accumulation input for recovery

If you use the Database Change Accumulation utility (DFSUCUM0) periodically to consolidate your old log volumes, you can use these accumulated volumes as input to the database recovery service and the Database Recovery utility.

You must include any log volumes recorded since the last time you ran the Database Change Accumulation utility as input to the recovery—either in their current format or after being passed through the Database Change Accumulation utility.

When you run the Database Change Accumulation utility, you can specify the DBDSs for which you want to accumulate change records. For each DBDS, you can either ignore change records or write them to the optional output log data set. By specifying a subset of the DBDSs (called a change accumulation group), you can reduce the size of the change accumulation data set, and improve performance of the Database Recovery utility. However, you also must run the Database Change Accumulation utility (and pass the log data set) for each change accumulation group.

If you select only one DBDS each time you run the Database Change Accumulation utility, you must run the utility (and pass the log data set) once for each DBDS. With this technique, each output change accumulation file contains only those records needed for the recovery of a single DBDS.

You might improve the performance of the Database Change Accumulation utility by performing frequent quiesces of your databases. Quiescing a database creates a point of consistency across databases and logs which reduces the number of log records that need to be accumulated. When creating a point of consistency, new updates are held and all in-progress updates are committed and written to DASD. After the last in-progress update is written to DASD a deallocation (DEALLOC)

time stamp is added to the database allocation (ALLOC) record in the RECON data set to create a new recovery point and updates to the database resume.

The following are some common recovery techniques:

- Define a single change accumulation group for all DBDSs.

  In this case, the Database Change Accumulation utility reads the log data set only once, but the Database Recovery utility must read records in the output change accumulation data set for potentially unnecessary databases. This technique improves the performance of the Database Change Accumulation utility, but degrades performance of the database recovery.

- Define several change accumulation groups.

  In this case, you divide your databases into groups of several databases, often by volume or by application. If you take image copies frequently, you can keep the number of records scanned by the Database Recovery utility to a minimum.

- Define no change accumulation groups. Run the Database Change Accumulation utility, but not as a regular job.

  When you need to recover a database, specify that the utility accumulate changes for the affected DBDS only. Provide all log data sets created since the latest image copy of the data set to be recovered. If you use one of the image copy utilities, you must include the log created when the image copy was made as input to the Database Change Accumulation utility.

  If you are considering this approach, you should make some test runs to see whether the sum of the execution times of the Database Change Accumulation and Database Recovery utilities is less than the time needed to run the Database Recovery utility with unaccumulated log data sets.

Related concepts:

"Database quiesce" on page 590

Using log data sets for recovery

If you decide not to use the Database Change Accumulation utility, consider using a recovery log data set (RLDS) produced by the Log Archive utility (DFSUARC0).

RLDSs are smaller than SLDSs because they contain only a subset of the recovery-related log records from the corresponding SLDSs. The Log Archive utility writes to the RLDS only those records that are needed for recovery. The frequency of image copies has a major impact on recovery time in this case, depending on the volume of database updates.

You must submit the log volumes to the recovery utility in chronological sequence: from the oldest to the most recent.

If you back out a database using the Database Batch Backout utility after making an image copy of it, you must use both the log created by the Batch Backout utility and the original log as input to the Database Recovery utility. This is because the Batch Backout utility backs out some of the changes recorded on the original log.

Supervising recovery using DBRC

Use DBRC to supervise database recovery. By using DBRC, your task of recovery is greatly simplified.

During daily operations, DBRC keeps track of the activity of the database: backup copies taken, relevant log volumes recorded, change accumulations made, and so forth. Thus, DBRC knows what to supply to the recovery utility to recover the

database: image copies and log volumes. DBRC generates the necessary JCL to run the utility and guarantee the proper input in the proper order.

DBRC supports two types of database recovery for DBDSs and area data sets (ADSs):

- *Full recovery* restores a DBDS or ADS to its current state as recorded in the RECON data set.
- *Time-stamp recovery* restores a DBDS or ADS to the contents it had at the time you specify.

Full recovery of a DBDS or ADS is generally a two-step process. You must restore a copy of the DBDS or ADS, then apply changes made subsequent to that copy. These changes can be contained in change accumulation data sets and log data sets. If you make backup copies of your databases, running the Database Recovery utility can accomplish both steps.

Time-stamp recovery of a DBDS or ADS involves recovering a DBDS or ADS to some previous point in time, usually when the DBDS or ADS was not being updated. Generally, you perform time-stamp recoveries to recover from logic errors, such as bad input data, or an operational error, such as duplicate execution of a batch job. A time-stamp recovery has the effect of backing out one or more of the most recent sets of updates.

Recommendation: Perform time-stamp recoveries with great caution. When recovering one DBDS or ADS, you must perform similar recoveries for all related DBDSs or ADSs. Examples of such related data sets include not only those connected through logical relationships, but also indexes and databases containing multiple data set groups. Because DBRC does not know how DBDSs or ADSs are related, you must be sure to perform all related time-stamp recoveries.

The general strategy for recovering a DEDB is to use the Change Accumulation utility and then perform forward recovery. To assist in obtaining valid input, you can use the Log Recovery utility to create a copy of the OLDS that excludes the incomplete DEDB updates. You can also create a data set with only DEDB recovery records. Use the DEDB Area Data Set Compare utility to assist in the repair of a damaged area data set.

Recommendation: For an area that uses the shared VSO option, do not bring the area back online or restart failed IMS systems that owned the connections until after you ensure that there are no failed-persistent XES connections to the CF structure used by the area. You can use the z/OS `SETXCF` command to delete the connections.

Related concepts:

"Overview of recovery of databases" on page 613

◆ Overview of DBRC (System Administration)

Related tasks:

◆ Recovering databases (System Administration)

Related reference:

◆ Recovery utilities (Database Utilities)

Denial of authorization for recovery utility

If DBRC denies authorization for an IMS recovery utility to run, you might receive message DFS3710A, which usually indicates that a previous recovery utility abended while processing and did not release its database authorization.

To recover in this situation, delete the subsystem record in the RECON data set using either the following command sequence or API request sequence:

- Command sequence:
  1. `CHANGE.SUBSYS SSID(name) STARTRCV`
  2. `CHANGE.SUBSYS SSID(name) ENDRECOV`
  3. `DELETE.SUBSYS SSID(name)`

  For more information about the DBRC commands, see *IMS Version 15 Commands, Volume 3: IMS Component and z/OS Commands*.

- API request sequence:
  1. `DSPAPI FUNC=COMMAND COMMAND=CHANGE.SUBSYS SSID(name) STARTRCV`
  2. `DSPAPI FUNC=COMMAND COMMAND=CHANGE.SUBSYS SSID(name) ENDRECOV`
  3. `DSPAPI FUNC=COMMAND COMMAND=DELETE.SUBSYS SSID(name)`

  For more information about the DBRC API, see *IMS Version 15 System Programming APIs*.

Overview of recovery of databases

You perform a forward recovery of a database by restoring a backup copy of the database and then reapplying all of the logged database changes that occurred after the backup copy was made.

While these two basic steps are common to all forward recoveries of databases, the specific steps of a forward recovery can vary depending on the type of database you are recovering and whether you are operating in a data-sharing environment or a non-data-sharing environment.

A backup copy of an IMS database is usually an image copy of the database. Image copies are created by one of the IMS image copy utilities or a separate image copy tool. The IMS image copy utilities automatically register the image copies they create with the Database Recovery Control facility (DBRC). Image copies created by some other means, or with DBRC=N specified on the image copy utility EXEC parameter list, might need to be registered with DBRC before DBRC and the Database Recovery utility (DFSURDB0) can use them for recovery.

The logs that contain the database changes can be processed before recovery by the Change Accumulation utility (DFSUCUM0), to consolidate all of the database change records and optimize them for recovery. In a data-sharing environment, you are required to use the Change Accumulation utility. In a non-data-sharing environment, using the Change Accumulation utility is optional, although it can be beneficial to use for performance reasons.

DBRC manages the forward recovery process and can generate the required JCL for all of the backup and recovery related utilities, except in cases in which the database being recovered is not registered with DBRC. The JCL that is generated by the DBRC command GENJCL.RECOV for the Database Recovery utility automatically identifies and includes the correct image copies, logs, DBD names, ddnames and time stamps.

The process for recovering the database data sets, that is, the data sets that contain the database records, is the same across most database types, including full-function, HALDB partitioned full-function, and Fast Path DEDB database types. The same process is used for a primary index of a HIDAM database in the same way as you recover a database data set: using image copies and logs.

For HALDB databases, however, the PHIDAM primary indexes and indirect list data sets (ILDSs) are not backed up or recovered; instead, after the database data sets they support are recovered, HALDB primary indexes and ILDSs are rebuilt by using the HALDB Index/ILDS Rebuild utility (DFSPREC0).

Another difference about recovering HALDB databases is that the DBRC command GENJCL.RECOV can generate JCL to recover all of the partitions of the HALDB database at once or to recover only a single partition. The GENJCL.RECOV command also provides a similar option for recovering entire Fast Path DEDB databases or individual DEDB areas.

When rebuilding HALDB primary indexes and ILDSs, consider using the DBRC command GENJCL.USER to create skeletal JCL members to allocate the necessary data sets. A separate skeletal JCL member is required for primary index data sets and ILDSs. When rebuilding the ILDS, use the free space option of the HALDB Index/ILDS Rebuild utility by specifying either ILEF or BOTHF on the utility control statement. The free space option uses VSAM load mode, which includes the free space called for in the VSAM DEFINE CLUSTER command.

Forward recovery in a data-sharing environment requires additional steps to consolidate the logs of all of the IMS systems sharing the database to be recovered.

A forward recovery can be either a *full recovery* or a *time-stamp recovery*. A full recovery restores a DBDS or area data set to its current state as recorded in the RECON data set. A time-stamp recovery restores a DBDS or area data set to the contents it had at the time you specify.

The following topics include examples of the steps for performing full forward recoveries of HIDAM and PHIDAM databases. Because the recovery process for database data sets is the same across database types, the steps presented for the recovery of HIDAM and PHIDAM DBDSs also apply to HDAM, PHDAM, and DEDB database data sets. Similarly, the recovery steps for an ILDS in a PHIDAM also apply to an ILDS in a PHDAM.

Related concepts:

"Supervising recovery using DBRC" on page 611

Related tasks:

"Example: recovering a HIDAM database in a non-data-sharing environment" on page 615

"Recovering a PHIDAM database in a non-data-sharing environment" on page 617

"Example: recovering a single HALDB partition in a non-data-sharing environment" on page 621

"Example: recovering a HIDAM database in a data-sharing environment" on page 622

Example: recovering a HIDAM database in a non-data-sharing environment

The following steps are an example of a full forward recovery of a HIDAM database with a secondary index in a non-data-sharing environment.

The database uses OSAM for its database data sets; however, database data sets can be OSAM or VSAM. Primary and secondary indexes are always VSAM. The database is registered to DBRC, operates in an online environment, and does not participate in data sharing. Because the database does not participate in data sharing, an accumulation of the database changes recorded in the logs is not required.

For a HIDAM database, you must back up and recover the primary index data set separately from the database data sets (DBDSs); however, you can simplify the backup and recovery process by defining DBRC groups that include the DBD names of both the database and the primary index. You can then issue GENJCL commands against the DBRC group to generate the necessary JCL for both DBDs at the same time.

To recover a HIDAM database:

1. If any OLDS contains database change records that are required for recovery, issue the DBRC command GENJCL.ARCHIVE to generate the necessary JCL to run the Log Archive utility (DFSUARC0).
2. Run the Log Archive utility. The Log Archive utility archives the records in the OLDS to an SLDS.
3. Delete and define both the OSAM database data sets and the VSAM KSDS primary index data set.
4. Issue the GENJCL.RECOV command to generate all of the required JCL for recovery of the HIDAM database. The JCL identifies the correct image copies to use, all of the appropriate logs, all the correct ddnames, and the correct time stamps.
5. Run the Database Recovery utility (DFSURDB0) on the database by executing the JCL that is generated by the GENJCL.RECOV command. The Database Recovery utility recovers the database data sets from the image copies and the database changes that are recorded in the logs.
6. Issue the GENJCL.RECOV command to generate all of the required JCL for recovery of the primary index of the HIDAM database. The JCL identifies the correct image copies to use, all of the appropriate logs, all the correct ddnames, and the correct time stamps.
7. Run the Database Recovery utility on the primary index by executing the JCL that is generated by the GENJCL.RECOV command. The Database Recovery utility recovers the primary index data set from the image copy and updates the data set with the primary index changes that are recorded in the logs.
8. Delete and define secondary index VSAM KSDS data sets.
9. Recover the secondary indexes by either:
   - Using the same forward recovery steps that you use to recover HIDAM DBDSs.
   - Rebuilding the secondary indexes by using a separately sold index builder tool.

The following JCL is an example of the GENJCL.RECOV commands that generate the JCL that is required for recovering a HIDAM database that has a secondary index:

```
//GENRECVX JOB CLASS=A,REGION=6M,
//         MSGCLASS=U,NOTIFY=&SYSUID
//MYLIB JCLLIB ORDER=(IMS.PROCLIB,IMSVS.CMXXX.TEAM01.JCL)
//*
//DELDEF1 EXEC PGM=IDCAMS
//*************************************************************
//*    D E F I N E    V S A M   D A T A   D B S
//*************************************************************
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD DSN=IMSVS.CMXXX.TEAM01.UTIL(XSTAP),DISP=SHR
//         DD DSN=IMSVS.CMXXX.TEAM01.UTIL(XSTAX),DISP=SHR
//         DD DSN=IMSVS.CMXXX.TEAM01.UTIL(XSTAY),DISP=SHR
//*
//* RUN DB RECOVERY
//A   EXEC  DBRC,
//SYSIN DD *
 GENJCL.RECOV DBD(X1STAP) LIST DEFAULTS(T01DFLT) ONEJOB JOB(T01RJOB) -
   MEMBER(RECV1JCL)
 GENJCL.RECOV DBD(X1STAX) LIST DEFAULTS(T01DFLT) ONEJOB JOB(T01RJOB) —
 GENJCL.RECOV DBD(X1STAY) LIST DEFAULTS(T01DFLT) ONEJOB JOB(T01RJOB) —
```

The following example shows the JCL that is generated by the GENJCL.RECOV commands to recover a HIDAM database that has a secondary index.

```
//T01RECOV JOB TIME=1,MSGCLASS=H,REGION=4096K,
// CLASS=A
//RCV1 EXEC PGM=DFSRRC00,REGION=1300K,
//          PARM='UDR,DFSURDB0,X1STAP,,,,,,,,,,,,,,,,,,,'
//*
//*    THIS JCL ORIGINATES FROM THE USER'S 'JCLPDS' LIBRARY.
//*    KEYWORDS ARE REPLACED BY THE GENJCL FUNCTION OF
//*    THE IMS/VS DATA BASE RECOVERY CONTROL FEATURE.
//*
//*            JCL FOR RECOVERY.
//*
//STEPLIB   DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT  DD SYSOUT=A
//IMS       DD DSN=IMS.DBDLIB,DISP=SHR
//X1STAP    DD DSN=IMSVS.CMXXX.TEAM01.X1STAP,
//          DISP=OLD
//DFSUDUMP  DD DSN=IMSVS.IMS1.TEAM01.X1STAP.D2006157.T155726,
//          UNIT=3390,
//          VOL=(PRIVATE,,,,SER=(SM4104)),
//          LABEL=(1,SL),
//          DISP=(OLD,KEEP),DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP  DD *
//SYSIN     DD *
//*...
//T01RECOV JOB TIME=1,MSGCLASS=H,REGION=4096K,
// CLASS=A
//RCV1 EXEC PGM=DFSRRC00,REGION=1300K,
//          PARM='UDR,DFSURDB0,X1STAX,,,,,,,,,,,,,,,,,,,'
//STEPLIB   DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT  DD SYSOUT=A
//IMS       DD DSN=IMS.DBDLIB,DISP=SHR
//X1STAX    DD DSN=IMSVS.CMXXX.TEAM01.X1STAX,
//          DISP=OLD
//DFSUDUMP  DD DSN=IMSVS.IMS1.TEAM01.X1STAX.D2006157.T155727,
//          UNIT=3390,
//          VOL=(PRIVATE,,,,SER=(SM4106)),
```

```
//             LABEL=(1,SL),
//             DISP=(OLD,KEEP),DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP  DD *
//SYSIN     DD *
//*...
//T01RECOV JOB TIME=1,MSGCLASS=H,REGION=4096K,
//  CLASS=A
//RCV1 EXEC PGM=DFSRRC00,REGION=1300K,
//             PARM='UDR,DFSURDB0,X1STAY,,,,,,,,,,,,,,,,,,,,'
//STEPLIB   DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT  DD SYSOUT=A
//IMS       DD DSN=IMS.DBDLIB,DISP=SHR
//X1STAX    DD DSN=IMSVS.CMXXX.TEAM01.X1STAY,
//             DISP=OLD
//DFSUDUMP  DD DSN=IMSVS.IMS1.TEAM01.X1STAY.D2006157.T155727,
//             UNIT=3390,
//             VOL=(PRIVATE,,,,SER=(SM4106)),
//             LABEL=(1,SL),
//             DISP=(OLD,KEEP),DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP  DD *
//SYSIN     DD *
```

Related concepts:

"Overview of recovery of databases" on page 613

Related tasks:

"Example: recovering a HIDAM database in a data-sharing environment" on page 622

Recovering a PHIDAM database in a non-data-sharing environment

The following steps are an example of a full forward recovery of a PHIDAM database that has two partitions and a partitioned secondary index (PSINDEX).

The database uses OSAM for its database data sets; however, database data sets can be OSAM or VSAM. Primary and secondary indexes are always VSAM. The database is registered to DBRC and operates in an online environment. The database does not participate in data sharing. Because the database does not participate in data sharing, an accumulation of the database changes recorded in the logs is not required.

To recover a PHIDAM database:

1. If any OLDS contains database change records that are required for recovery, issue the DBRC command GENJCL.ARCHIVE to generate the necessary JCL to run the Log Archive utility (DFSUARC0).

   If the DBRC JCLOUT DD statement for the GENJCL output is directed to the internal reader, the archive jobs are automatically started.

2. If the archive jobs are not started automatically, run the Log Archive utility on each sharing IMS system that has unarchived records in the OLDS. The Log Archive utility archives the records in the OLDS to an SLDS.

3. Delete and define the OSAM database data sets.

4. Issue the GENJCL.RECOV command to generate all of the required JCL for recovery of the PHIDAM database. The JCL identifies the correct image copies to use, the correct logs, the correct ddnames, and the correct time stamps.

5. Run the Database Recovery utility (DFSURDB0) on the database by executing the JCL that is generated by the GENJCL.RECOV command. The Database Recovery utility recovers the database data sets from the image copies and the database changes that are recorded in the logs.

6. Delete and define both the primary index data set and the ILDS.

7. Run the HALDB Index/ILDS Rebuild utility (DFSPREC0) to rebuild both the primary index and the ILDS in each partition.

   Specify BOTHF to select the free space option of the HALDB Index/ILDS Rebuild utility. The free space option uses VSAM load mode to include the free space called for in the FREESPACE parameter of the DEFINE CLUSTER command.

   The HALDB Index/ILDS Rebuild utility rebuilds the primary index data sets and the ILDS of one partition at a time; however, you can run multiple instances of the utility on multiple partitions in parallel.

8. Delete and define secondary index VSAM KSDS data sets.

9. Recover the PSINDEXes by either:
   - Using the same forward recovery steps that you use to recover PHIDAM DBDSs.
   - Rebuilding the secondary indexes by using a separately sold index builder tool.

The following code shows an example of the GENJCL.RECOV command that generates the JCL to recover a PHIDAM database.

```
//DBRC      EXEC PGM=DSPURX00
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//         DD DISP=SHR,DSN=IMS.MDALIB
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//JCLPDS   DD DISP=SHR,DSN=IMS.PROCLIB
//JCLOUT   DD DISP=SHR,DSN=JOUKO4.HALDB.CNTL(RECOVOUT)
//JCLOUTS  DD SYSOUT=*
//SYSIN    DD *
GENJCL.RECOV NOJOB DBD(NORDDB) MEMBER(RECOVJCL)
/*
```

The following code shows the JCL required for recovering a PHIDAM database that has two partitions:

```
//RCV1 EXEC PGM=DFSRRC00,
//             PARM='UDR,DFSURDB0,NORDDB,,,,,,,,,,,,Y,,,,,,,,,'
//STEPLIB   DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSPRINT  DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*
//IMS       DD DISP=SHR,DSN=IMS.DBDLIB
//NORDDB1A    DD DSN=IMSPSA.IM0A.NORDDB.A00001,
//            DISP=OLD,
//            DCB=BUFNO=10
//DFSUDUMP  DD DSN=JOUKO3.IMCOPY2.NORDDB.C01,
//            DISP=OLD,DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP  DD DISP=SHR,
//            DSN=IMS.PROCLIB(DFSVSMDB)
//SYSIN     DD *
```

```
S NORDDB NORDDB1A
/*
//RCV2 EXEC PGM=DFSRRC00,
//           PARM='UDR,DFSURDB0,NORDDB,,,,,,,,,,,,Y,,,,,,,,,'
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//NORDDB2A   DD DSN=IMSPSA.IM0A.NORDDB.A00002,
//           DISP=OLD,
//           DCB=BUFNO=10
//DFSUDUMP DD DSN=IMSPSA.IMCOPY.NORDDB2,
//           DISP=OLD,DCB=BUFNO=10
//DFSVDUMP DD DUMMY
//DFSUCUM  DD DUMMY
//DFSULOG  DD DUMMY
//DFSVSAMP DD DISP=SHR,
//           DSN=IMS.PROCLIB(DFSVSMDB)
//SYSIN    DD *
S NORDDB NORDDB2A
/*
```

The following code shows an example of the GENJCL.RECOV command that generates the JCL to recover a PSINDEX.

```
//DBRC     EXEC PGM=DSPURX00
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//         DD DISP=SHR,DSN=IMS.MDALIB
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//JCLPDS   DD DISP=SHR,DSN=IMS.PROCLIB
//JCLOUT   DD DISP=SHR,DSN=JOUKO4.HALDB.CNTL(RECOVOU3)
//JCLOUTS  DD SYSOUT=*
//SYSIN    DD *
GENJCL.RECOV DBD(CUSTSI) ONEJOB LIST MEMBER(RECOVJCL)
```

The following code shows the JCL for recovering a PSINDEX that has two partitions:

```
//IVPGNJCL JOB (999,POK),
// 'JJ',
// CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
// REGION=64M
/* JOBPARM SYSAFF=SC42
//RCV1 EXEC PGM=DFSRRC00,
//           PARM='UDR,DFSURDB0,CUSTSI,,,,,,,,,,,,Y,,,,,,,,,'
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//CUSTSI1A   DD DSN=IMSPSA.IM0A.CUSTSI.A00001,
//           DISP=OLD
//DFSUDUMP DD DSN=IMSPSA.IC1.CUSTSI1.CUSTSI1A.D03073.T143654,
//           DISP=OLD,DCB=BUFNO=10
//DFSVDUMP DD DUMMY
//DFSUCUM  DD DUMMY
//DFSULOG  DD DUMMY
//DFSVSAMP DD DISP=SHR,
//           DSN=IMS.PROCLIB(DFSVSMDB)
//SYSIN    DD *
S CUSTSI CUSTSI1A
/*
//RCV2 EXEC PGM=DFSRRC00,
//           PARM='UDR,DFSURDB0,CUSTSI,,,,,,,,,,,,Y,,,,,,,,,'
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSPRINT DD SYSOUT=*
```

```
//SYSUDUMP  DD SYSOUT=*
//IMS       DD DISP=SHR,DSN=IMS.DBDLIB
//CUSTSI2A   DD DSN=IMSPSA.IM0A.CUSTSI.A00002,
//          DISP=OLD
//DFSUDUMP DD DSN=IMSPSA.IC1.CUSTSI2.CUSTSI2A.D03073.T143654,
//          DISP=OLD,DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP  DD DISP=SHR,
//            DSN=IMS.PROCLIB(DFSVSMDB)
//SYSIN     DD *
S CUSTSI CUSTSI2A
/*
```

Related concepts:

"Overview of recovery of databases" on page 613

Related reference:

➡ Generating JCL and user-defined output (Commands)

Recovering a PHIDAM database in a data sharing environment

The following steps are an example of a full forward recovery of a PHIDAM database that has two partitions and a partitioned secondary index (PSINDEX). The PHIDAM database is shared by multiple IMS systems.

Image copies of the database data sets must be available.

The PHIDAM database data sets must be registered as members of a DBRC change accumulation group.

The database uses OSAM for its database data sets; however, database data sets can be OSAM or VSAM. Primary and secondary indexes are always VSAM. The database is registered to DBRC and operates in an online environment. The database is shared by multiple IMS systems. Because the database is shared, an accumulation of the database changes recorded in the logs is required.

In a full recovery, recovering secondary indexes is not required, so in following example procedure, the steps for recovering the secondary index are optional.

IMS provides a skeletal JCL member, DSPUPJCL, for rebuilding the index or indirect list data sets (ILDS) for HALDB partitions.

To recover a PHIDAM database:

1. If any OLDS contains database change records that are required for recovery, issue the DBRC command GENJCL.ARCHIVE to generate the necessary JCL to run the Log Archive utility (DFSUARC0).

   If the DBRC JCLOUT DD statement for the GENJCL output is directed to the internal reader, the archive jobs are automatically started.

2. If the archive jobs are not automatically started, run the Log Archive utility on each sharing IMS system that has unarchived records in the OLDS. The Log Archive utility archives the records in the OLDS to an SLDS.

3. Run the GENJCL.CA command to merge the logs from the IMS systems in to the latest CA.

4. Delete and define the OSAM database data sets.

5. Issue the GENJCL.RECOV command to generate all of the required JCL for recovery of the PHIDAM database. The JCL identifies the correct image copies to use, the correct logs, the correct change accumulation data set, the correct ddnames, and the correct time stamps.

6. Run the Database Recovery utility (DFSURDB0) on the database by executing the JCL that is generated by the GENJCL.RECOV command. The Database Recovery utility recovers the database data sets from the image copies and the database changes that are recorded in the logs.

7. Delete and define both the primary index data set and the ILDS.

8. Run the HALDB Index/ILDS Rebuild utility (DFSPREC0) to rebuild both the primary index and the ILDS in each partition.

   Specify BOTHF to select the free space option of the HALDB Index/ILDS Rebuild utility. The free space option uses VSAM load mode to include the free space called for in the FREESPACE parameter of the DEFINE CLUSTER command.

   The HALDB Index/ILDS Rebuild utility rebuilds the primary index data sets and the ILDS of one partition at a time; however, you can run multiple instances of the utility on multiple partitions in parallel.

9. Optionally, delete and define secondary index VSAM KSDS data sets.

10. Optionally, recover the PSINDEXes by either:
    • Using the same forward recovery steps that you use to recover PHIDAM DBDSs.
    • Rebuilding the secondary indexes by using a separately sold index builder tool.

Related concepts:

"Overview of recovery of databases" on page 613

Example: recovering a single HALDB partition in a non-data-sharing environment

To perform a full forward recovery for a single HALDB partition, specify the partition name instead of the HALDB master database name in the DBD parameter of the DBRC command GENJCL.RECOV.

The GENJCL.RECOV command generates the JCL to recover only the specified partition. After the DBDSs are recovered, you must rebuild the primary index and the ILDS.

To recover a single partition in a PHIDAM database:

1. If any OLDS contains database change records that are required for recovery, issue the DBRC command GENJCL.ARCHIVE to generate the necessary JCL to run the Log Archive utility (DFSUARC0).

2. Run the Log Archive utility. The Log Archive utility archives the records in the OLDS to an SLDS.

3. Delete and define the OSAM partition database data sets.

4. Issue the GENJCL.RECOV command, specifying the partition name in the DBD parameter to generate the required JCL for recovery of the database data sets of the partition. The JCL identifies the correct image copies to use, the correct logs, the correct ddnames, and the correct time stamps.

5. Run the Database Recovery utility (DFSURDB0) on the partition by executing the JCL that is generated by the GENJCL.RECOV command. For the specified

partition, the Database Recovery utility recovers the database data sets from the image copies and the database changes that are recorded in the logs.

6. Delete and define both the primary index data set and the ILDS.

7. Run the HALDB Index/ILDS Rebuild utility (DFSPREC0) to rebuild both the primary index and the ILDS in the partition. Specify BOTHF to select the free space option of the HALDB Index/ILDS Rebuild utility

   The free space option uses VSAM load mode to include the free space called for in the FREESPACE parameter of the DEFINE CLUSTER command.

   The HALDB Index/ILDS Rebuild utility rebuilds the primary index data sets and the ILDS of one partition at a time; however, you can run multiple instances of the utility on multiple partitions in parallel.

The following code shows an example of the GENJCL.RECOV command that generates the JCL to recover a single HALDB partition.

```
//DBRC     EXEC PGM=DSPURX00
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//         DD DISP=SHR,DSN=IMS.MDALIB
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//JCLPDS   DD DISP=SHR,DSN=IMS.PROCLIB
//JCLOUT   DD DISP=SHR,DSN=JOUKO4.HALDB.CNTL(RECOVOUT)
//JCLOUTS  DD SYSOUT=*
//SYSIN    DD *
  GENJCL.RECOV NOJOB DBD(NORDDB1) MEMBER(RECOVJCL)
/*
```

The following code shows the JCL that is generated by the GENJCL.RECOV command to recover a single HALDB partition.

```
//RCV1 EXEC PGM=DFSRRC00,
//             PARM='UDR,DFSURDB0,NORDDB,,,,,,,,,,,,Y,,,,,,,,,'
//*
//STEPLIB  DD DISP=SHR,DSN=IMS.SDFSRESL
//         DD DISP=SHR,DSN=IMS.MDALIB
//IMS      DD DISP=SHR,DSN=IMS.DBDLIB
//SYSPRINT  DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*
//NORDDB1A    DD DSN=IMSPSA.IM0A.NORDDB.A00001,
//             DISP=OLD,
//             DCB=BUFNO=10
//DFSUDUMP  DD DSN=IMSPSA.NORDDB1.IMCOPY,
//             DISP=OLD,DCB=BUFNO=10
//DFSVDUMP  DD DUMMY
//DFSUCUM   DD DUMMY
//DFSULOG   DD DUMMY
//DFSVSAMP DD DSN=IMS.PROCLIB(DFSVSM0S),DISP=SHR
//SYSIN    DD *
S NORDDB NORDDB1A
/*
```

Related concepts:

"Overview of recovery of databases" on page 613

Example: recovering a HIDAM database in a data-sharing environment

The following steps perform a full forward recovery a HIDAM database in a data-sharing environment.

The database uses OSAM for its database data sets; however, database data sets can be OSAM or VSAM. Primary and secondary indexes are always VSAM. The database is registered to DBRC and operates in an online environment. Because the database participates in data sharing, prior to recovering the database, you must run the Database Change Accumulation utility (DFSUCUM0). The Database Change Accumulation utility consolidates and sorts the database changes recorded in the logs of each of the IMS systems that share the database.

Apart from requiring you to accumulate the database changes in the logs, the recovery steps in a data-sharing environment are the same as those in a non-data-sharing environment.

For a HIDAM database, the primary index data set must be backed up and recovered separately from the database data sets (DBDSs); however, you can simplify the backup and recovery processes by defining a DBRC group that includes the database names of both the database and the primary index as defined in their respective database definitions. You can then issue GENJCL commands against the DBRC group to generate the necessary JCL for both of the databases at the same time.

To recover a HIDAM database in a data-sharing environment:
1. Ensure that the OLDS in each of the IMS systems that share the database has been archived by issuing the DBRC command `GENJCL.ARCHIVE`. The `GENJCL.ARCHIVE` command generates the necessary JCL to run the Log Archive utility (DFSUARC0).
2. Run the Log Archive utility in each of the sharing IMS systems. The Log Archive utility archives the records in the OLDS to an SLDS.
3. Build the change accumulation JCL by issuing the DBRC command `GENJCL.CA`.
4. Run the Database Change Accumulation utility to consolidate, sort, and save the database change log records to a change accumulation data set. The Change Accumulation utility registers the change accumulation data set with DBRC.
5. Delete and define both the OSAM database data sets and the VSAM KSDS primary index data set.
6. Issue the `GENJCL.RECOV` command to generate all of the required JCL for recovery of the HIDAM database. The JCL identifies the correct image copies to use, the appropriate logs, the correct ddnames, and the correct time stamps.
7. Run the Database Recovery utility (DFSURDB0) on the database by executing the JCL that is generated by the `GENJCL.RECOV` command. The Database Recovery utility recovers the database data sets from the image copies and the database changes that are recorded in the logs.
8. Issue the `GENJCL.RECOV` command to generate all of the required JCL for recovery of the primary index of the HIDAM database. The JCL identifies the correct image copies to use, all of the appropriate logs, all the correct ddnames, and the correct time stamps.
9. Run the Database Recovery utility on the primary index by executing the JCL that is generated by the `GENJCL.RECOV` command. The Database Recovery utility recovers the primary index from the image copies and the primary index changes that are recorded in the logs.

The following JCL shows an example of the GENJCL.CA command that generates the JCL that is required to run the Database Change Accumulation utility for a HIDAM database in a data sharing environment.

```
//STEP1    EXEC PGM=DSPURX00,REGION=4096K
//STEPLIB  DD  DISP=SHR,DSN=IMS.SDFSRESL
//IMS      DD  DSN=IMS.DBDLIB,DISP=SHR
//JCLPDS   DD  DSN=IMS.JCLLIB,DISP=SHR
//JCLOUT   DD  SYSOU&JCLOUT
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
GENJCL.CA GRPNAME(NORDDBCA) JOB MEMBER(CAJCLF) LIST JOB(CAJOB)
//*
```

The following JCL is an example of the JCL that is generated by the GENJCL.CA
command.

```
//CA1  EXEC PGM=DFSUCUM0,PARM='CORE=100000,DBRC=Y'
//*           JCL FOR CHANGE ACCUMULATION.
//STEPLIB   DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSPRINT  DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*
//IMS       DD DISP=SHR,DSN=IMS.DBDLIB
//SYSOUT    DD SYSOUT=*
//SORTLIB   DD DSN=SYS1.SORTLIB,DISP=SHR
//SORTWK01  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//SORTWK02  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//SORTWK03  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//SORTWK04  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//SORTWK05  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//SORTWK06  DD UNIT=SYSALLDA,SPACE=(CYL,(5),,CONTIG)
//DFSUCUMO  DD DUMMY,DCB=BLKSIZE=100
//DFSUCUMN  DD DSN=IMS.CA.NORDDBCA.CA172538,
//             DISP=(NEW,CATLG),
//             UNIT=SYSALLDA,
//             VOL=SER=DJX112,
//             SPACE=(CYL,(1,1))
//DFSULOG   DD DSN=IMS.SLDS.G2170V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//          DD DSN=IMS.SLDS.G2171V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//          DD DSN=IMS.SLDS.G2172V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//          DD DSN=IMS.SLDS.G2173V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//          DD DSN=IMS.SLDS.G2174V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//          DD DSN=IMS.SLDS.G2175V00,
//             DCB=RECFM=VB,
//             DISP=OLD
//DFSUDD1   DD DUMMY
//SYSIN     DD *
DB0NORDDB  061392202286+0000          NORDDB1A
DB0NORDDB  061392202304+0000          NORDDB2A
DB0NORDDB  061392202321+0000          NORDDB3A
```

Related concepts:

"Overview of recovery of databases" on page 613

Related tasks:

"Example: recovering a HIDAM database in a non-data-sharing environment" on
page 615

Concurrent image copy recovery

A concurrent image copy is "fuzzy" because while IMS makes the image copy of the database, IMS might also be updating the database at the same time.

For example, IMS might could update the database:
- Before the start of the concurrent image copy, and IMS might not have written those updates to the DBDS yet
- While the concurrent image copy is being taken

After restoring the data set from the concurrent image copy, IMS applies changes which occurred while the image copy was being taken from the log. Therefore, to recover a DBDS that has fuzzy image copies, you might need to supply logs to the recovery job which have time stamps that precede the start of the concurrent image copy. The DBRC GENJCL.RECOV command generates JCL with the correct logs listed.

If you use the database recovery service to recover a DBDS that has fuzzy image copies, the database recovery service automatically processes all the necessary logs required for the recovery.

Related concepts:

"Concurrent image copies" on page 597

HSSP image copy recovery

An HSSP image copy is usable as a fuzzy image copy whenever the HSSP image copy completes successfully.

When you use an HSSP image copy for database recovery, you must tell the Database Recovery utility that you are supplying an HSSP image copy.

If the image copy process fails during the creation of an image copy, IMS returns the image copy data set to the pool of available image copy data sets.

Related concepts:

"HSSP image copies" on page 599

DL/I I/O errors and recovery

The way IMS handles DL/I I/O errors allows you to delay database recovery until it is convenient.

If the errors do not recur or can be corrected automatically, recovery might even be unnecessary. IMS maintains data integrity by reconstructing information about I/O errors on later warm or emergency restarts.

When an I/O error occurs, IMS issues message DFS0451I and creates an extended error queue element (EEQE). When IMS successfully forward recovers a DBDS with an EEQE, DBRC updates the appropriate entries in the RECON data set, and IMS removes the EEQE.

You can use the /DISPLAY DB command with the BKERR keyword to display information about the outstanding EEQEs associated with a database.

Related concepts:

"Database failures" on page 589

DEDB full condition

The database-full condition in a DEDB area indicates that one or more portions of the root-addressable area of the DEDB can no longer accept ISRT calls.

IMS detects the out-of-space condition for DEDB sequential dependent segments during sync-point processing. Although the application program may receive a warning status code, some ISRT calls, issued from a Fast Path message-driven program or from an MPP region, cause the region to abnormally terminate with a U0844 abend code.

Recommendation: Track space usage within a DEDB area using one of the following methods:

- The application program can use information given by the position (POS) call
- The master terminal operator can monitor it using the appropriate /DISPLAY DB or QUERY DB command to show the total allocated and the total used CIs in both the root-addressable and the sequential-dependent portions of the area.

If a DEDB area does reach the database-full condition, Fast Path provides two online utilities that may negate the need for using an offline application program to perform a database unload and reload operation to resolve the problem. The two utilities are:

- The Sequential Dependent Delete utility, which deletes the sequential-dependent segments in all or part of the sequential-dependent portion of an area. This makes the space available for further sequential dependent inserts.
- The Direct Reorganization utility, which removes space fragmentation in an area. The utility runs against one DEDB area at a time.

  Recommendation: Although you can limit the reorganization to the one unit of work that did not allow inserts, you should reorganize more or all of the area. Any independent overflow space that is freed from one unit of work becomes available for use by any other unit of work within that area.

Continued database authorization

IMS performs a number of actions when a write I/O error is encountered on a DBDS.

If IMS encounters a write I/O error on a DBDS, and the DBDS is registered with DBRC, IMS:

1. Builds an EEQE for the data that could not be written to the DBDS
2. Calls DBRC

If you register the database with DBRC, DBRC updates the RECON data set for the DBDS by:

- Flagging the DBDS record to indicate that recovery is needed
- Recording the address of the EEQE in the DBDS record
- Incrementing the DB record to count the EEQE

The recovery-needed flag prohibits new authorizations for some IMS utilities such as the Batch Image Copy utility. Also, if the recovery-needed flag is set and there are no EEQEs, DBRC denies all authorizations except for the recovery utilities.

If you do not register the database with DBRC, IMS assumes that the EEQEs that it builds from the log records during warm or emergency restart are correct. IMS

discards EEQE information for nonregistered databases when you stop the
database by using the /DBRECOVERY DB or UPDATE DB STOP(ACCESS)
command.

I/O error retry

When IMS closes a database after updating it, IMS automatically retries EEQEs for
read and write errors.

IMS authorizes attached subsystems to use a database with write errors, but IMS
does not allow them to access blocks or CIs that were not successfully written. If
any application program in one of these subsystems requests data requiring access
to one of these blocks or CIs, IMS gives the program an A0 status code, indicating a
read error.

When the I/O-error retry is successful, IMS deletes the EEQE and sends messages
to the MTO and z/OS console. IMS sends message DFS0614I for each error that it
successfully retries, and message DFS0615I when it resolves all outstanding errors.

Correcting bad pointers

Ordinarily, bad pointers should not occur in your database. When they do, the
likely causes are few.

The cause of bad pointers is typically:
- Failure to run database backout
- Failure to perform emergency restart
- Omitting a log during backout or recovery

The normal way to correct a bad pointer is to perform recovery. However, some
cases exist in which a bad pointer can be corrected through reorganization. A
description of the circumstances in which this can or cannot be done is as follows:
- PC/PT pointers. The HD Unload utility issues unqualified GN calls to read a
  database. If the bad pointer is a PC or PT pointer, DL/I will follow the bad
  pointer and the GN call will fail. Therefore, reorganization cannot be used to
  correct PC or PT pointers.
- LP/LT pointers. LP and LT pointers are rebuilt during reorganization. However,
  DL/I can follow the LP pointer during unload. If the logical child segment
  contains a direct LP pointer and the logical parent's concatenated key is not
  physically stored in the logical child segment, DL/I follows the bad LP pointer
  to construct the logical parent's concatenated key. This causes an ABEND.
- LP pointer. When DBR= is specified for pre-reorganization and the database has
  direct LP pointers, the HD Unload utility saves the old LP pointer. Bad LP
  pointers produce an error message (DFS879) saying a logical child that has no
  logical parent exists.
- LP pointer. When DBIL= is specified for pre-reorganization of a logical child or
  parent database, the utilities that resolve LP pointers use concatenated keys to
  match logical parent and logical child segments. New LP pointers are created.

Chapter 28. Database backout

Backward recovery or *backout* is one of the two major types of recovery. Using backout allows you to remove incorrect or unwanted changes from existing information or work.

Dynamic backout

IMS automatically backs out changes to a database in several different circumstances.

IMS automatically backs out changes to a database when any of the following events occur:

- An application program terminates abnormally.
- An application program issues a rollback call (ROLL or ROLB), or ROLS call without a token.
- An application program tries to access an unavailable database but has not issued the INIT call.
- A deadlock occurs.

In batch, you can specify (in the JCL) that IMS automatically dynamically back out changes if a batch application program abends, issues a ROLB call, or issues a ROLS call without a token. In this case, the log data set must be on DASD.

However, GSAM DB's are not backed out but are repositioned during the BMP's restart process through the XRST call. The XRST call repositions the dataset pointers to the checkpoint ID specified in the call. When the application starts-up, it will pick-up from that point and go forward. The checkpoint ID specified in the XRST call should be the same one that the non-GSAM DBs would have been backed out to, through either dynamic or batch backout.

Dynamic backouts and commit points

During dynamic backout, IMS backs out all database changes made by an application program since its last commit point.

A commit point (or sync point) occurs in a batch or BMP program when the program issues a CHKP call. Commit points for message-driven application programs depend on the transaction mode (as specified by the MODE parameter of the TRANSACT macro).

Frequent commit points decrease performance. However, they also:

- Allow IMS to send output messages (replies) earlier
- Reduce the time required for emergency restart and database recovery
- Help avoid storage shortages for locking
- Help avoid reading SLDSs for backout, which decreases performance

Application programs can be batch oriented (non-message-driven BMPs) and choose when to commit their database changes using the CHKP call.

The following kinds of programs can issue a rollback (ROLB) call:

- Message-driven application programs that establish a commit point every time they attempt to get an input message
- Non-message-driven batch-oriented application programs

If an application program uses Fast Path resources or an external subsystem, IMS can issue an internal rollback call to back out the data to the last commit point. The application program receives status code `FD` as a result of this call.

IMS performs the following processing in response to a `ROLB` call:
- Backs out and releases locks for all database changes that the program has made since its most recent sync point
- Cancels all output messages since the program's most recent sync point
- Returns the first segment of the first input message since the most recent commit point to the application program, if an input message is still in process at this point, and if the call provides an I/O area

When IMS completes `ROLB` processing, the application program resumes processing from its last commit point.

When IMS initiates dynamic backout because of an application program abend, in many cases IMS stops both the transaction and application program. The MTO can restart the transaction or the application program. If you restart the application program before determining the exact cause of the abend, the program can abend again. However, if you restart the transaction, queuing of transactions continues.

Consider the transaction mode in deciding whether the MTO should restart transactions:
- If you restart response-mode transactions, and IMS subsequently enqueues new messages, any terminal entering the transaction is locked because no response is received.
- If you do not restart response-mode transactions, the terminal operator receives a message noting that IMS has stopped the transaction, but the originating terminal is not locked.
- If the transaction is not a response-mode transaction, you can restart it to allow terminal operators to continue entry. However, establish procedures in this case to warn terminal operators that they might not receive a response for some time.

When an application program abends, IMS issues message `DFS554A` to the master terminal. IMS does not issue this message for MPP regions when a resource shortage that is not expected to last long (such as short-term locks) occurs. In this case, IMS backs out the application program and places the input message back on the queue for rescheduling. When a BMP abends, IMS always issues message `DFS554A` because the z/OS operator must restart BMPs.

Message `DFS554A` identifies:
- The application program (PSB) name
- The transaction name
- The system or user completion codes
- The input logical terminal name
- Whether the program or transaction is stopped

IMS also sends message DFS555I to the input terminal or master terminal when the application program abends while processing an input message. This error message means that IMS has discarded the last input message the application was processing.

The DFS555I message contains:
- The system or user completion codes
- Up to the first 78 characters of the input message
- The time and date

Related information:

➥ DFS554A (Messages and Codes)

➥ DFS555I (Messages and Codes)

Dynamic backout in batch

In a batch environment, when the SLDS is on DASD, you can request that IMS perform dynamic backout if IMS pseudoabends or if the application program issues a ROLB call by specifying BKO=Y in the JCL.

In this case, IMS performs backout to the last program sync point. Abend U0828 does not occur in batch if you specify BKO=Y.

Database batch backout

You can use the Batch Backout utility (DFSBBO00) to remove database changes made by IMS batch jobs and online programs.

Because it is possible (in an online IMS subsystem) to have more than one dependent region using the same PSB, the utility might back out changes from several dependent regions when you execute batch backout against an online log.

You can use the Batch Backout utility to back out changes to the last checkpoint a batch job did not complete normally. If the batch region does not use data sharing, and if it is not a BMP, you can use the CHKPT control statement on the JCL for the Batch Backout utility to back out changes for the batch region to any valid checkpoint, whether or not it completed normally. Do not specify the CHKPT control statement when backing out changes for BMPs; the utility either rejects or ignores it.

The Batch Backout utility reads the log in the forward direction. Regardless of whether the backout is to the last checkpoint or to a specified checkpoint, the utility tries to back out all changes to all databases occurring between that checkpoint and the end of the log. When backing out BMPs, the utility always backs them out to the last checkpoint on the log.

In an IMS DBCTL environment, the Batch Backout utility backs out all in-flight updates, but only backs out in-doubt updates if you specify the COLDSTART option.

If dynamic backout fails or if backout during emergency restart fails, IMS stops the databases for which backout did not complete, and retries the backouts when you restart the databases.

When to use the Batch Backout utility

Use the Batch Backout utility in any of the following circumstances.

- If a batch job fails and you did not request automatic dynamic backout.
- If a batch job fails and you requested dynamic backout, but the failure was not a pseudo-abend.
- If you perform an emergency restart of IMS specifying the NOBMP option, and BMPs were active at the time of failure, you must run the utility once for each BMP. If you run the utility before restart completes, you must use the ACTIVE control statement.
- If you perform an emergency restart of IMS specifying the COLDBASE option, you must run batch backout once for each PSB (including MPPs, BMPs, and mixed mode IFPs) used by an application program that updates a full-function DL/I database, if the PSB was active at the time of the failure.

    You should also run batch backout for each PSB for which there is an in-doubt unit of recovery (UOR). Doing so puts the full-function databases in the same state as the Fast Path databases that are not updated until the commit point. If you choose not to back out an unresolved in-doubt UOR, you must remove it from the RECON backout record.

    If you want to initiate these backouts before restart completes, you must specify the COLDSTART control statement for the Batch Backout utility. You do not need to specify the COLDSTART statement for backouts performed after restart.

    DBRC protects all registered databases affected by in-flight and in-doubt UORs from access by other application programs until their backouts complete, even if the backout does not complete until after a cold start. You must ensure that programs do not access any unregistered databases before their backouts are complete.

Before issuing a /ERESTART COLDSYS command, you must run batch backout at least once using the COLDSTART or ACTIVE control statement. These control statements provide DBRC with the information necessary to protect registered databases in need of backout from erroneous access until the backouts are done.

If a failure occurs in one of the online backouts in a DB/DC or DBCTL subsystem that causes IMS to defer the backout:

1. Resolve the original problem.
2. Issue the /START DB or UPDATE DB START(ACCESS) command.
3. Complete the backout with a partial (restartable) backout.

In some cases, the restartable backout can also fail because the information it requires has not been saved. In those cases, you must use the Batch Backout utility. You must also use the utility if IMS defers one of the backout failures past a cold start.

System failure during backout

If a system failure occurs during backout, execute the Batch Backout utility again. Save the input logs and the output log from the successful backout run as input to the Database Change Accumulation or Database Recovery utility.

DL/I I/O errors during backout

IMS handles I/O errors for DL/I databases in the same way for dynamic backout, batch backout, and emergency restart backout.

Errors during dynamic backout

If a database I/O error occurs during backout, IMS issues message, creates an EEQE to record the event, and calls DBRC to flag the appropriate database entry in the RECON data set as needing backout.

You should run the Database Backout utility to back out the appropriate database entries and to inform DBRC to reset the appropriate backout-needed flag and counter. DBRC does not authorize flagged databases for use until you run the Database Backout utility.

For write errors, IMS copies the buffer contents of the blocks or control intervals in error to virtual buffers pointed to by the EEQE. IMS also tries to write each buffer that has an outstanding error when the database is closed. IMS issues message DFS0614I for each error buffer that it successfully writes or reads (if the EEQE is for a read error), and issues message DFS0615I when all outstanding errors have been resolved.

If a database read error occurs during dynamic backout, IMS issues message DFS983I, stops only the database in error, and continues dynamic backout of other databases within the PSB. IMS writes a X'4C01' log record for each database that it successfully backs out, and a X'4C80' log record for each database that it stops because of a read error. IMS then allows you to recover any stopped database and back it out offline.

If dynamic backout fails, you must run batch backout to reconstruct the database.

Related concepts:

➜ IMS failure recovery (Operations and Automation)

Recovering from errors during dynamic backout

Different kinds of failure require different recovery procedures.

Specifically:
- If a database is unavailable during dynamic backout or emergency restart, but subsequently becomes available (for example, you bring it back online or repair a physical device), use a /START DB or UPDATE DB START(ACCESS) command to reschedule dynamic backout to back out the (now available) database.
- If the database remains unavailable (because, for example, the database has an error that you must recover), you must run the Database Recovery utility before you allow further processing against the database in error, and then run the Batch Backout utility. Batch backout recognizes the X'4C01' log records and does not try to back out changes for databases that were backed out successfully.
- In the online environment, dynamic backout can fail if the log records are not available. If the only log records available are from SLDS, the performance of dynamic backout is poor. Slow dynamic backout can happen in the following cases:
  - The OLDS containing the log record has been archived and reused.

    Recommendation: Make sure that you have defined enough OLDS space, and that programs reach sync points (using GU or CHKP calls) frequently enough to ensure that records needed for backout are available from OLDSs.
  - IMS stops a database during dynamic backout if a nonrecoverable read error occurs on the logs when single or dual logging is in effect.

In this case, you must run the Log Recovery utility to get a valid log. The output log must terminate normally; use it as input to the Database Change Accumulation utility or the Database Recovery utility. After you have corrected the input error, execute the Batch Backout utility again to correct the databases.

Errors during batch backout

If an I/O error occurs during batch backout, the Batch Backout utility completes backout of all databases except for those affected by read errors.

IMS handles these I/O errors in the same way that it handles them when they occur during dynamic backout.

Before the Batch Backout utility makes any updates, it builds database buffers for any outstanding I/O errors. It applies updates to these buffers without trying to write them until the database is closed.

If any I/O errors remain unresolved for a database that has been backed out, you must eventually do a forward recovery, but you do not need to rerun the Batch Backout utility if it completed successfully (IMS issues message DFS395I).

Errors on log during batch backout

If an I/O error occurs on the input log, you should execute the Log Recovery utility to correct the error.

The output log must terminate normally; keep this log as input to the Database Change Accumulation or Database Recovery utility. After you correct the input error, execute the Batch Backout utility again.

If the I/O error occurs on the output log, terminate the output log correctly. Then execute the Batch Backout utility again. Keep both output logs as input to the Database Change Accumulation or Database Recovery utility.

Errors during emergency restart backout

IMS handles both read errors and failures to open a database during emergency restart in the same way as it does during dynamic backout.

For read errors, however, other restart backouts for the same database can take place, even though a database is stopped.

If a nonrecoverable OLDS read error occurs during backout (on both OLDSs if dual logging is in effect), run the Log Recovery utility and attempt to restart IMS again.

During restart processing, but before initiating backout, IMS determines if it must close the OLDS using the WADS. If a nonrecoverable read error occurs on the WADS, committed log records might be lost. In this case, you must close and archive the OLDS, and then reconstruct the affected databases by performing forward recovery and batch backout.

Chapter 29. Monitoring databases

You can use a number of IMS tools to monitor the performance of your databases.

Several tools this topic does not discuss, but which you can also use for monitoring purposes, include:

- IMS Performance Analyzer
- IMS DB Control Suite (On-demand Space Monitor)
- IMS DB Tools Space Monitor Utilities
- DB Integrity Control Facility

Related Reading: For information about these and other IMS tools, go to www.ibm.com/ims and link to the IBM DB2 and IMS Tools website.

Related concepts:

"Monitoring VSAM buffers" on page 703

"Code inspection 2" on page 32

"Number of buffers" on page 442

"When you should reorganize a database" on page 644

➡ IMS Monitor (System Administration)

➡ Data sharing in IMS environments (System Administration)

Related tasks:

"Changing the amount of space allocated" on page 711

IMS Monitor

The IMS Monitor is a tool that records data about the performance of your DL/I databases in a batch environment.

The recorded data is produced in a variety of reports. The monitor's usefulness is twofold. First, when you run the monitor routinely, it gives you performance data over time. By comparing this data, you can determine whether the performance trend is acceptable. This helps you make decisions about tuning your database and determining when it needs to be reorganized.

The second use of the monitor is to assess how the changes you make effect performance. Once you have accumulated reports describing normal database processing, you can use them as a profile against which to compare the effect of your changes. Examples of changes you might make (then test for performance) include:

- Changes in the structure of your databases
- A change from one DL/I access method to another
- A change in database buffer pool number and size
- Changes in application program logic

In all these cases, your primary goal is probably to minimize the number of I/Os required to perform an operation. The monitor helps you determine whether you have met your objective.

The following example shows how to use the IMS Monitor: suppose you are performing a final test on a new or revised application. The monitor reports show that some DL/I calls in the program, which should have required a single I/O retrieval, actually required a large database scan involving many I/Os. You might be able to correct this problem by making changes in the application program logic.

The monitor itself is actually two programs, as shown in the following figure.
- The IMS Monitor (DFSMNTR0)
- The IMS Monitor Report Print utility (DFSUTR20)
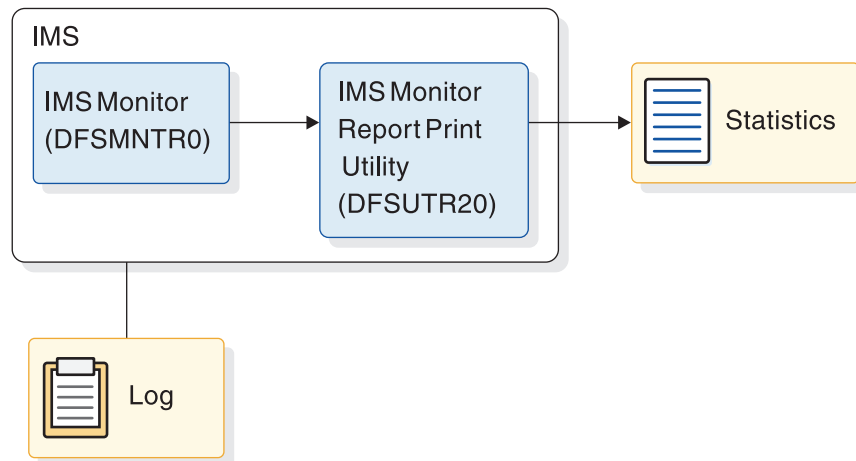
Figure 246. How the IMS Monitor works

The IMS Monitor collects data from IMS control blocks (when DL/I is operating) and records the data either on an independent data set or in the IMS log. It collects data with minimum interference to the system. The monitor runs in the same address space as the IMS job, and it can be turned on or off with the MON= parameter in the execution JCL.

The IMS Monitor Report Print utility is an offline program that produces reports summarizing information collected by the IMS Monitor. It produces the following reports:
- VSAM Buffer Pool report
- VSAM Statistics report
- Database Buffer Pool report
- Program I/O report
- DL/I Call Summary report
- Distribution Appendix report
- Monitor Overhead report

Many of these reports are also provided by the IMS Monitor.

When the IMS Monitor is on, it remains on until the batch execution ends, requiring some overhead. It cannot be turned on and off from the system console. To minimize the monitor's impact, use the IMS Monitor in a single-thread test environment rather than multi-thread application environments.

This ensures that the data gathered by the IMS Monitor can be related to a particular program.

Related concepts:

➪ DB Monitor reports (System Administration)

➪ IMS Monitor reports (System Administration)

IMS Monitor user exit

The IMS Monitor user exit provides access to the IMS Monitor data without the need to modify IMS code.

The exit provides an exit point that gives real-time information for the entire lifetime of an IMS, whereas the IMS Monitor Report utility is run after the fact on data that was collected at a certain time interval.

When EXITDEF=(TYPE=IMSMON,EXITS=(*exit_name(s)*)) is specified in the USER_EXITS section of the DFSDFxxx member, the IMSMON exit is called (which means the *exit_name(s)* that are defined as IMS Monitor user exit type are called) during IMS initialization, IMS termination, and anytime an IMS Monitor log record is written. The exit is called regardless whether the Monitor is on or off.

The IMS Monitor exit parameter list is mapped by DSECT MONEXPL in macro DFSLOGP.

Refer to the sample exit, DFSMONX0, in the IMS.SDFSSMPL data set for an example of how to register interest in SLOG codes for the INIT call.

Related reference:

➪ IMS Monitor user exit (IMSMON) (Exit Routines)

Monitoring Fast Path systems

The major emphasis for monitoring IMS online systems that include message-driven Fast Path applications is the balance between rapid response and high transaction rates.

With Fast Path, performance data is made part of the system log information. Because the bulk of the online traffic is expected to be handled by expedited message handling and not be present on the message queues, the Fast Path Log Analysis utility (DBFULTA0) is the prime tool for monitoring Fast Path applications. The IMS Monitor can also be used to monitor Fast Path systems.

Use the Fast Path Log Analysis utility (DBFULTA0) to prepare statistical reports for Fast Path based on data recorded on the IMS system log. This utility is offline and produces five reports useful for system installation, tuning, and troubleshooting:

- A detailed listing of exception transactions
- A summary of exception detail by transaction code for MPP (message-processing program) regions
- A summary by transaction code for MPP regions
- A summary of IFP, BMP, and CCTL transactions by PSB name or transaction code
- A summary of the log analysis

Do not confuse this utility with the IMS Monitor or the IMS Log Transaction Analysis utility.

As an administrator in the Fast Path environment, you should perform tasks, like establishing monitoring strategies, performance profiles, and analysis procedures. This topic highlights how to use the Analysis utility to do these tasks, and suggests some Areas where tuning activities might be valuable.

Related concepts:

➡ IMS Monitor (System Administration)

➡ IMS Monitor reports (System Administration)

Related reference:

➡ Fast Path Log Analysis utility (DBFULTA0) (System Utilities)

➡ CCTL exit routines (Exit Routines)

Fast Path log analysis utility

The Fast Path Log Analysis utility gathers statistics of Fast Path exclusive and potential transactions that are passed to Fast Path dependent regions.

It reports information for other PSBs (including Fast Path PCBs and the programs that enter the sync point processing) and produces three types of output:

• Formatted summary and detail reports
• A data set of fixed format records for the total traffic of Fast Path transactions extracted from the system logs that form the input to the utility
• A data set of records, in the same format, that are selected based on exception conditions (such as those transactions that exceed a certain fixed response time)

The latter data sets can be analyzed in more detail by your installation's programs. They can also be sorted to group critical transactions or events. The details of the record format and meaning of the fields are given in *IMS Version 15 System Utilities*.

Fast Path log reduction

To reduce log volume you can use the LGNR parameter, which is specified on the DBC, FDR, and IMS startup procedures.

LGNR indicates the maximum number of DEDB buffer alterations that are held before the entire CI is logged.

Another way to reduce log volume is to designate the DEDB as nonrecoverable. No changes to the database are logged and no record of database updates is kept in the DBRC RECON data set.

Related concepts:

"Non-recovery option" on page 203

➡ Fast Path EXEC parameters in DBCTL (System Definition)

➡ Fast Path EXEC parameters in DCCTL or DB/DC (System Definition)

Related reference:

➡ Parameter descriptions for IMS procedures (System Definition)

Fast Path transaction timings

For each Fast Path transaction, four time intervals are separately calculated.

The following figure shows the boundary events and intervals.
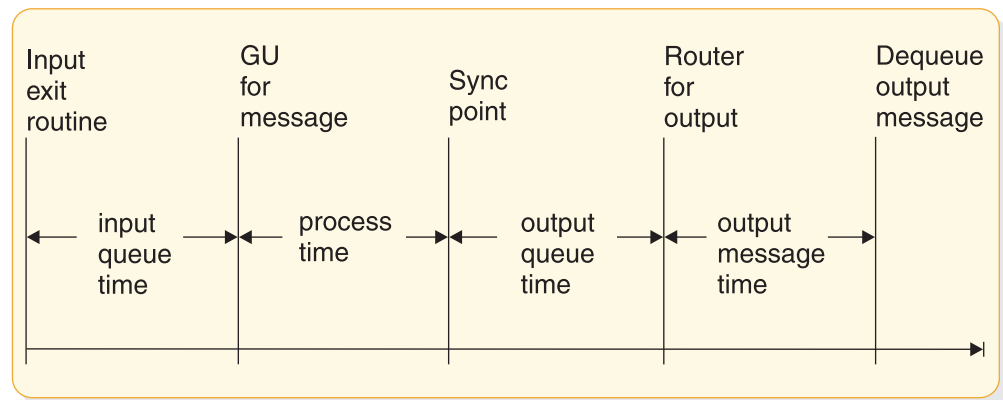
Figure 247. Fast Path transaction event timings

The following list describes the four intervals shown in the previous figure:

1. Input queue time: reflects the transaction input queuing within the balancing group to distribute the work.
2. Process time: records the actual elapsed processing time for the individual transaction.
3. Output queue time: shows the effect of sync point in delaying the output message release until after logging.
4. Output message time: shows the line and device availability for receiving the output message. If the transaction originated from a programmable controller, the output time could reflect a delay in dequeue caused by the output not being acknowledged until the next input.

The sum of the first three intervals is termed the *transit time*. This time is slightly different from a response time, because it excludes the line activity for the message, message formatting, and the input edit processing up to the time the message segment leaves the exit routine.

Monitored events for Fast Path

The control program automatically collects Fast Path event data during system operation.

The following table shows the information that is made part of the system log records for each Fast Path transaction.

Table 78. Monitor data for Fast Path transactions

Monitored data	Message-driven region	Other region
Transit and Output Message Times	x	
LTERM Name	x	
Routing Code	x	
Balancing Group Queue Count	x	
Number of DEDB Calls	x	x
Number of I/O to DEDB	x	x
Number of MSDB Calls	x	x
Number of CI Contentions	x	x

Table 78. Monitor data for Fast Path transactions (continued)

Monitored data	Message-driven region	Other region
Number of Buffers Allocated	x	x
Number of Waits for Buffer	x	x
Sync Point Failure Reason Code	x	x

Selecting transactions

The analysis utility lets you select transactions to be reported in detail.

You give the transaction code and a transit time that each transaction is to exceed, up to a maximum of 65.5 seconds. Several codes can be selected for each utility run. There is also a way to ask for all transactions that exceed the given transit time. In this case, the individual exception specification overrides the general one.

When you do not need to print all such occurrences of the exceptions, you can give a maximum number of detail records to be printed. The default is 1000 individual records, though you can specify up to 9999999 as the maximum number. When you cut off the number of *printed* records, the data set for the exception records contains all transactions that meet the selection criteria.

You can also specify a start time and end time for the transaction reporting interval. The start time corresponds to the earliest transaction that satisfies the clock time (format HH:MM:SS) specified by a utility input control statement. End time is set by the latest transaction that enters the sync point processing before the ending clock time that is specified on an input control statement.

Another selection technique that is available is to select only non-message-driven transactions for reporting. Use this to look at the activity (occurring against MSDBs or DEDBs) caused by calls from IMS programs or BMPs.

Interpreting Fast Path analysis reports

The analysis reports show the origin, database activity, and processing events for each transaction code, although most reported items show average and maximum values.

The reports produced are:
- Overall summary by transaction

  Summarized by transaction code, the transit times and input/output message lengths are given. The database calls and buffer usage are also included.
- Exception detail

  For those transactions selected, the terminal origin and routing code are given for each individual occurrence of the transaction. The detail also includes the data appearing in the overall summary.
- Summary of exception detail by transaction code

  This report is based on the transactions in the exception report. The items reported are the same as for the overall summary.
- Summary of transactions by PSB

  All programs that are in non-message-driven regions, MPP regions, and BMP regions that enter the sync point processing are reported. The items reported are the same as the summary of exception detail.
- Recapitulation of the analysis

This is a documentation aid that gives the grand totals of transactions input to the analysis, and the I/O for online utilities.

The combination of the interval covered by the system log input to the utility and the exception criteria you define in the input control statements determines the content of these reports.

Examples of the reports format and the definition of the items reported can be found in *IMS Version 15 System Utilities*, within the description of the Fast Path Log Analysis utility.

Following are some suggestions for interpreting the reported events:
- Examine the summary reports and investigate the reasons for sync point failure.
- Examine the summary report to see if buffer usage was consistently under the NBA values. Check all negative differences that indicate the need for overflow buffers to see that they were unusual occurrences.
- Compare the database call counts to those of the expected profile. Select those transactions that show unusual patterns for a run to produce a detailed exception report.
- Examine the balancing group queue counts to see if they are conforming with the scheduling algorithm expectations.

Chapter 30. Tuning databases

Tune your database either to improve performance or to better use database space.

You can tune your database in a variety of ways by using the reorganization utilities.

Keep in mind that when you tune your database, you are often making more than a simple change to it. For example, you might need to reorganize your database and at the same time change operating system access methods. This topic has procedures to guide you through making each type of change. If you are making more than one change at a time, you should look at the flowcharts in "Changing the number of data set groups" on page 749. When used in conjunction with the individual procedures in this topic, the flowcharts guide you in making some types of multiple changes to the database.

Also, some of the tuning changes you make can affect the logic in application programs. You can often use the dictionary to analyze the affect before making changes. In addition, some changes require that you code new DBDs and PSBs. If you initialize your changes in the dictionary, you can then use the dictionary to help create new DBDs and PSBs.

Related concepts:

"Number of buffers" on page 442

▶ Data sharing in IMS environments (System Administration)

Reorganizing the database

Reorganizing a database means changing how the data in the database is organized to improve performance.

In some cases, reorganizing a database might also refer to modifying the database's structure or the structure of the records and segments in the database. Although this topic focuses on changing how data is organized, you can use many of the reorganization utilities discussed here to make structural changes as well.

Two database types, DEDB and HALDB, support online reorganization in addition to the offline methods of reorganization discussed here.

For full-function databases, CI reclaim can also improve performance of DL/I calls to VSAM key sequenced data sets. CI reclaim reduces the number of empty CIs that are read on DL/I calls.

Related concepts:

"HALDB online reorganization" on page 669

"VSAM KSDS CI reclaim for full-function databases" on page 418

Chapter 31, "Modifying databases," on page 721

Related tasks:

"Ensuring a well-organized database" on page 699

Related reference:

➡ High-Speed DEDB Direct Reorganization utility (DBFUHDR0) (Database Utilities)

When you should reorganize a database

You should reorganize your database in the following circumstances.

- Database performance has deteriorated. This can happen either because segments in a database record are stored across too many CIs or blocks, or because you are running out of free space in your database.
- There is too much physical I/O to DASD.
- The database structure has changed. For example, you should reorganize a HALDB partition after changing its boundaries or high key.
- The HDAM or PHDAM randomizing routine has changed.
- The HALDB Partitions Selection exit routine has changed.

The DB Monitor can aid in monitoring a database to help you determine when it is time to reorganize your database.

Related concepts:

Chapter 29, "Monitoring databases," on page 635

Reorganizing databases offline

You perform three basic steps when reorganizing a database offline when you are not making structural changes to the database.

The steps for reorganizing a database offline are:

1. Unload the existing database.
2. Delete the old database space and defining new database space. (This practice is always good, but it is only necessary if you have multiple extents or volumes, or are using VSAM.) For VSAM, database space refers to the clusters defined to VSAM for database data sets.
3. Reload the database.

Related concepts:

Chapter 31, "Modifying databases," on page 721

Protecting your database during an offline reorganization

When you reorganize your database offline, you delete it. Therefore, you should protect it from system or reorganization failure.

You can protect your existing database by renaming the space it occupies and then defining new database space. You should take an image copy of your database as soon as it is reloaded and before any application programs are run against it. Taking an image copy provides you with a backup copy of the database and establishes a point of recovery with DBRC in case of system failure. You can create image copies of your database using the Database Image Copy utility or the Database Image Copy 2 utility, which are described in detail in *IMS Version 15 Database Utilities*.

Reorganization utilities

You can reorganize your database by using IMS utilities. This topic introduces you to these utilities and explains how they work together.

You can use the following reorganization utilities with HALDB:

- HD Reorganization Unload utility (DFSURGU0)
- HD Reorganization Reload utility (DFSURGL0)
- Database Prereorganization utility (DFSURPR0)
- HALDB Partition Data Set Initialization utility (DFSUPNT0)

For HALDB, both the Database Prereorganization utility and the HALDB Partition Data Set Initialization utility (DFSUPNT0) initialize partitions.

If you are migrating an HDAM or HIDAM database to HALDB, the Prereorganization utility allows some reuse of your existing JCL by disabling full-function database utilities, such as Scan, Prefix Resolution and Prefix Update, in the DFSURCDS data set. After the database is migrated, you can use the HALDB Partition Data Set Initialization utility, which has additional functions such as unconditional specific partition initialization.

The utilities cannot be used to reorganize HSAM, SHSAM, or GSAM databases. To reorganize these databases, you must write a program to read the old database and then create a new database.

You are not required to use these reorganization utilities to reorganize your database. You can write your own programs to unload and reload data. You need to write your own programs only if you are making structural changes to your database that cannot be done using these utilities.

Several of the reorganization utilities can be used when initially loading a database. They are not used to load the database but to collect and sort the pointer information needed in a segment's prefix. Therefore, as you read through the utilities you will find some described as "used for initial load or reorganization".

The reorganization utilities can be classified into three groups, based on the type of reorganization you plan to do:
- Partial reorganization
- Reorganization using UCF
- Reorganization without UCF

Related concepts:

"Loading a database with logical relationships or secondary indexes" on page 584

Chapter 31, "Modifying databases," on page 721

Related reference:

➡ Reorganization and conversion utilities (Database Utilities)

Partial offline reorganization

If you are reorganizing an HD database, you can reorganize parts of it, rather than the whole database.

You would need to reorganize parts, rather than all of it, for two reasons:
- Only parts of it need to be reorganized.
- By reorganizing only parts of it, you can break the amount of time it takes to do a total reorganization into smaller pieces.

The utilities you use to do a partial reorganization are:
- The Database Surveyor utility, which helps you determine which parts of your database to reorganize

- The Partial Database Reorganization utility, which does the actual reorganization

HALDB partitions do not support partial offline reorganization.

Offline reorganization using UCF

Reorganization can be done using a single program, called the Utility Control Facility (UCF), or by using various combinations of utilities.

When UCF is used, it acts as a controller, determining which of the various reorganization utilities need to be executed and then getting them executed. Using UCF reduces the number of JCL statements you must create and eliminates the need to sequence the various utilities for execution. It also reduces the number of decisions operations people must make.

Offline reorganization by using the reorganization utilities

If you do not use the Utility Control Facility (UCF), reorganization of a database is done using a combination of utilities. Which utilities you need to use, and how many, depends on the type of database and whether it uses logical relationships or secondary indexes.

If your database does not use logical relationships or secondary indexes, you simply run the appropriate unload and reload utilities, which are as follows:
- For HISAM databases, the HISAM Reorganization Unload utility and the HISAM Reorganization Reload utility
- For HIDAM index databases (if reorganized separately from the HIDAM database), the HISAM Reorganization Unload utility and the HISAM Reorganization Reload utility
- For SHISAM, HDAM, and HIDAM databases, the HD Reorganization Unload utility and the HD Reorganization Reload utility

If your database does use logical relationships or secondary indexes, you need to run the HD Reorganization Unload and Reload utilities (even if it is a HISAM database). In addition, you must run a variety of other utilities to collect, sort, and restore pointer information from a segment's prefix. Remember, when a database is reorganized, the location of segments changes. If logical relationships or secondary indexes are used, update prefixes to reflect new segment locations. The various utilities involved in updating segment prefixes are:
- Database Prereorganization utility
- Database Scan utility
- Database Prefix Resolution utility
- Database Prefix Update utility

These utilities can also be used to resolve prefix information during initial load of the database.

In the discussion of the utilities in this section, the four unload and reload utilities are discussed first. The four utilities used to resolve prefix information are then discussed. When reading through the utilities for the first time, you need to understand that, if logical relationships or secondary indexes exist (requiring use of the latter four utilities), the sequence in which operations is as follows:
1. Unload
2. Collect more prefix information
3. Reload

4. Collect more prefix information
5. Updated prefixes

You will find, for instance, that the HD Reorganization Reload utility does not just reload the database if a secondary index or logical relationship exists. It reloads the database using one input as a data set containing some of the prefix information that has been collected. It then produces a data set containing more prefix information as output from the reload. When the various utilities do their processing, they use data sets produced by previously executed utilities and produce data sets for use by subsequently executed utilities. When reading through the utilities, watch the input and output data set names, to understand what is happening.

The following figure shows you the sequence in which utilities are executed if logical relationships or secondary indexes exist.
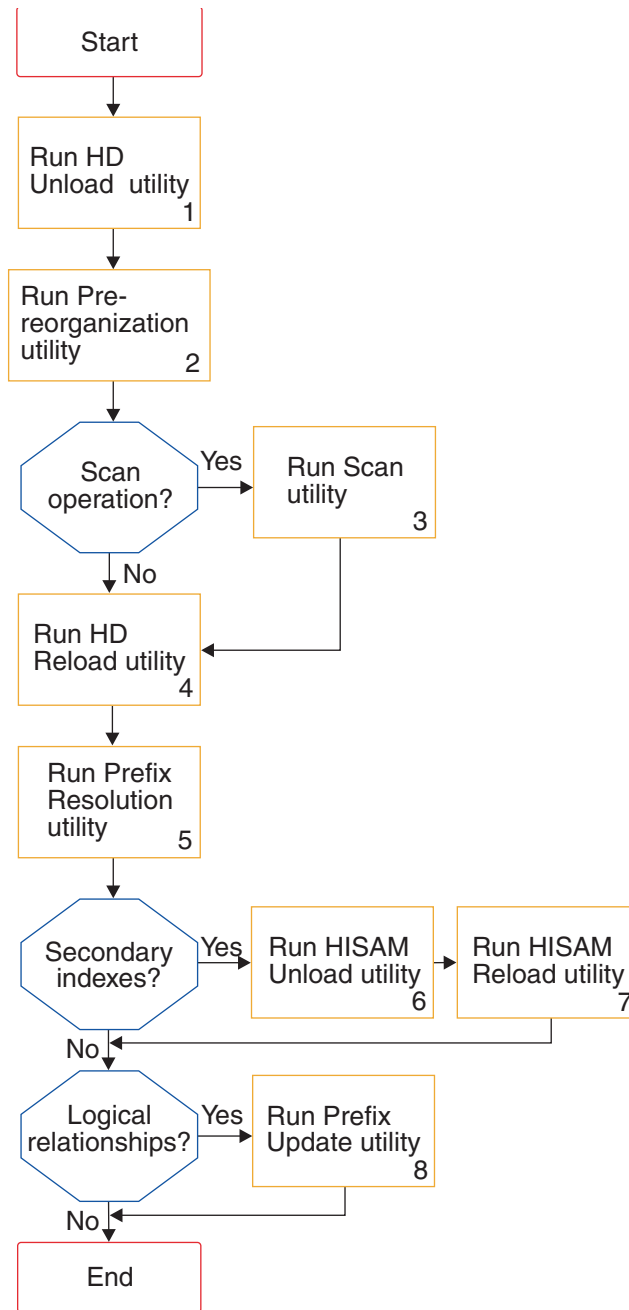
Figure 248. Steps in reorganizing when logical relationships or secondary indexes exist

The following figure shows the sequence for these utilities when using HALDB partitions.
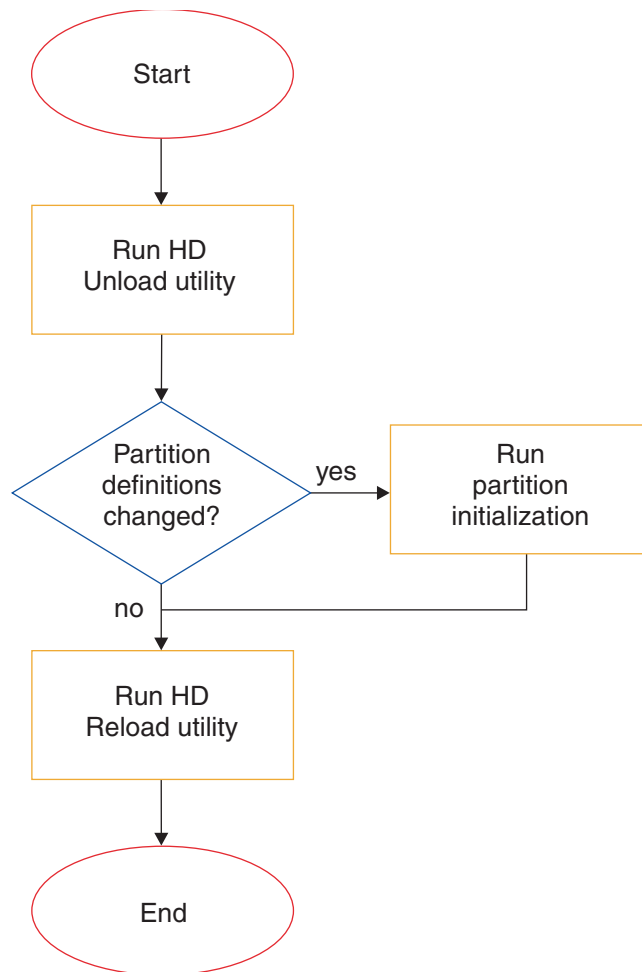
Figure 249. Steps for reorganizing HALDB partitions when logical relationships or secondary indexes exist

As an alternative, where above figure calls for the Partition Initialization utility, you can run the Prereorganization utility.

Related tasks:

"Overview of HALDB offline reorganization" on page 664

"Unloading and reloading using the reorganization utilities" on page 722

HISAM Reorganization Unload utility (DFSURUL0)

You use the HISAM Unload utility to unload a HISAM database or HIDAM index database.

SHISAM databases are unloaded using the HD Reorganization Unload utility.

If your database uses secondary indexes, you also use the HISAM Unload utility (later in the reorganization process) to perform a variety of other operations associated with secondary indexes.

The following figure shows the input to and output from the HISAM Reorganization Unload utility.
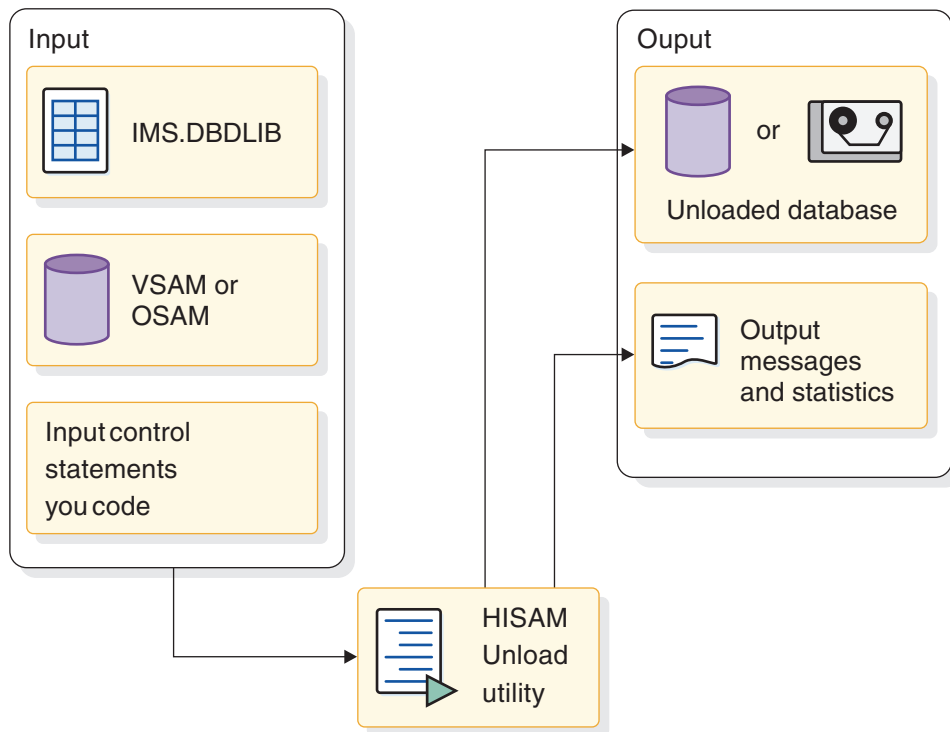
Figure 250. HISAM Reorganization Unload utility (DFSURUL0)

HISAM Reorganization Reload utility (DFSURRL0)

You use the HISAM reload utility to reload a HISAM database. You also use the HISAM reload utility to reload the primary index of a HIDAM database.

SHISAM databases are reloaded using the HD Reorganization Reload utility.

If your databases use secondary indexes, you use the HISAM reload utility (later in the reorganization process) to perform a variety of other operations associated with secondary indexes.

The following figure shows the input to and output from the HISAM Reorganization Reload utility.
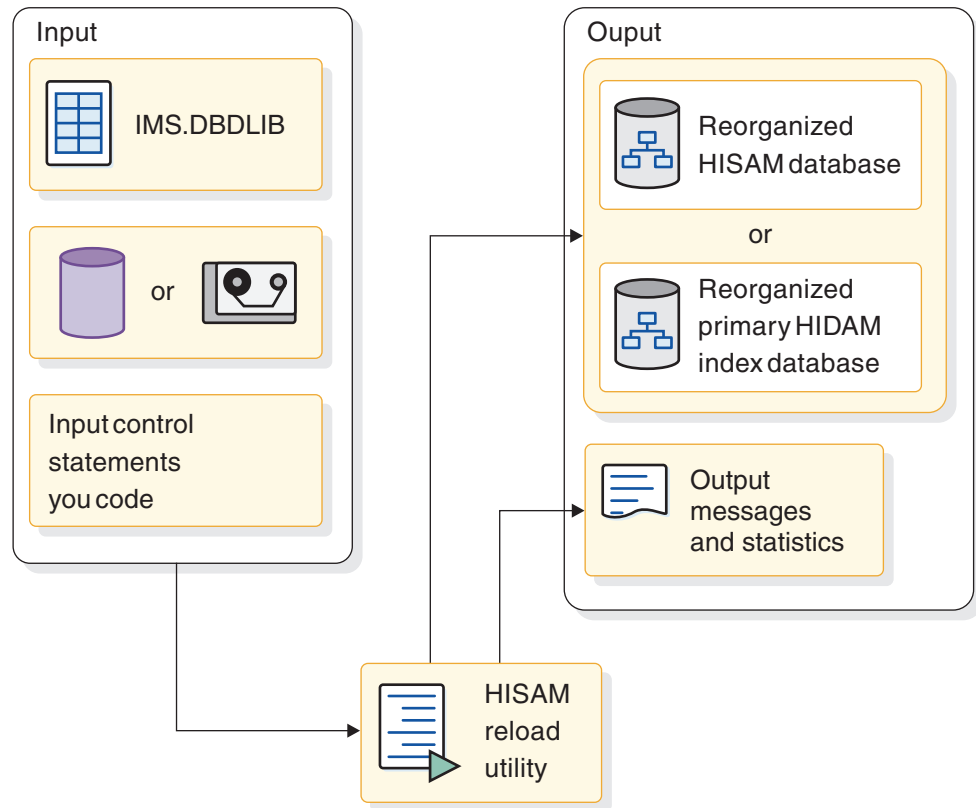
Figure 251. HISAM Reorganization Reload utility (DFSURRL0)

HD Reorganization Unload utility (DFSURGU0)

You use the HD Reorganization Unload utility to unload hierarchic direct (HD) databases.

You can use the HD Reorganization Unload utility to unload the following types of databases:

- HDAM, HIDAM, or SHISAM databases
- HISAM databases that use secondary indexes
- HISAM databases that use symbolic pointers in a logical relationship
- HISAM databases without segment/edit compression that are being converted to HISAM databases with segment/edit compression.
- PHDAM databases or partitions
- PHIDAM databases or partitions
- PSINDEX databases or partitions

The following figure shows the input to and output from the HD Reorganization Unload utility.
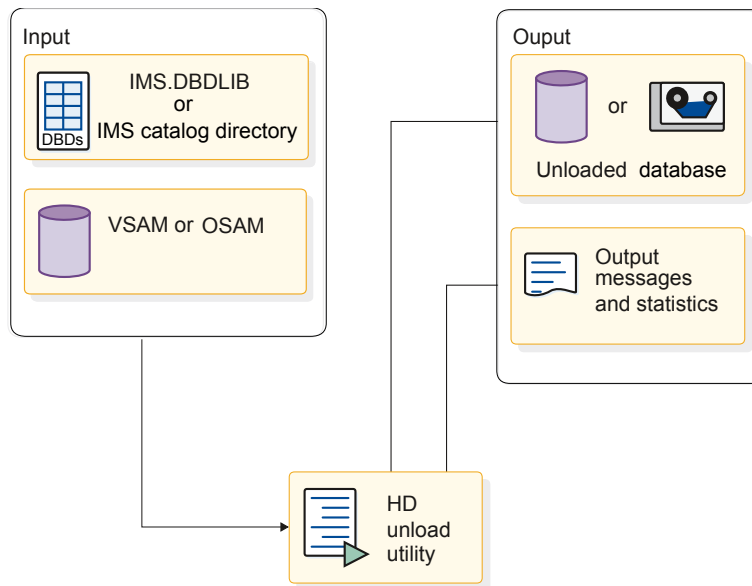
Figure 252. HD Reorganization Unload utility (DFSURGU0)

If you use the HD Reorganization Unload utility to unload a HALDB database (a PHDAM, PHIDAM, or PSINDEX database), you do not need to include DD statements for the database data sets. The HD Reorganization Unload utility uses dynamic allocation for HALDB data sets.

HD Reorganization Reload utility (DFSURGL0)

You use the HD Reorganization Reload utility to reload hierarchic direct (HD) databases.

You can use the HD Reorganization Reload utility to reload the following types of databases:

- HDAM, HIDAM, PHDAM, PHIDAM, PSINDEX, or SHISAM databases
- HISAM databases that use logical relationships or secondary indexes
- HISAM databases without segment/edit compression that are being converted to HISAM databases with segment/edit compression

The following figure shows the input to and output from the HD Reorganization Reload utility.
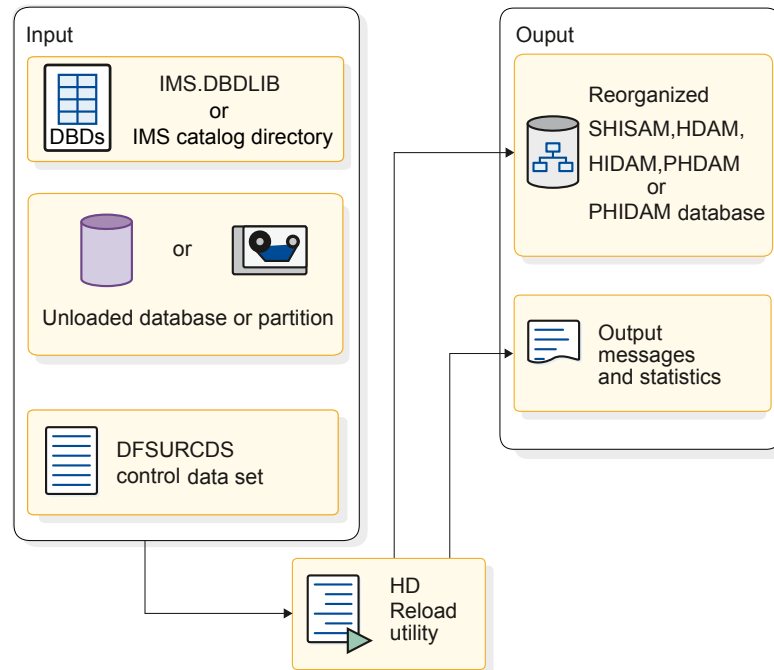
Figure 253. HD Reorganization Reload utility (DFSURGL0)

If logical relationships or secondary indexes exist in the database being reloaded, the DFSURCDS control data set created by the Prereorganization utility is used as one input to the HD Reorganization Reload utility. The DFSURCDS control data set contains information needed to resolve secondary index or logical relationship pointers.

When logical relationships or secondary indexes exist, the HD Reorganization Reload utility produces as output the DFSURWF1 work data set. DFSURCDS identifies the information that is collected on DFSURWF1.

The DFSURWF1 work data set becomes input to the Database Prefix Resolution utility. In the preceding figure note that, if the database being reloaded has a primary index, it is reloaded automatically when the main database is reloaded. A HIDAM index database can also be reorganized as a separate operation using the HISAM unload and reload utilities.

Exception: DFSURWF1 is not used for HALDB databases.

Database Prereorganization utility (DFSURPR0)

The Database Prereorganization utility (DFSURPR0) runs before you load or reorganize databases that have secondary indexes or logical relationships.

You use the Database Prereorganization utility when:
- A database to be initially loaded or reorganized has secondary indexes or logical relationships
- A database *not* being initially loaded or reorganized contains segments involved in logical relationships with databases that are being loaded or reorganized

The following figure shows the input to and output from the Database Prereorganization utility.
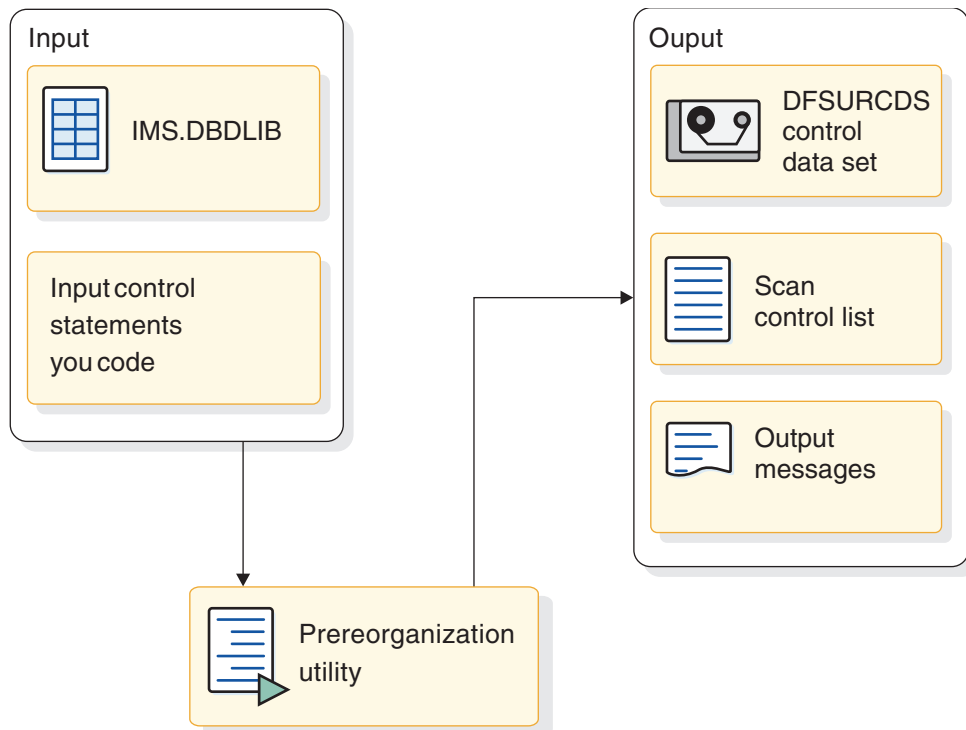
Figure 254. Database Prereorganization utility (DFSURPR0)

The DFSURPR0 utility produces the DFSURCDS control data set, which contains information about what pointers need to be resolved later if secondary indexing or logical relationships exist.

The DFSURCDS control data set produced by the Prereorganization utility is used as input to the following:

- The Database Scan utility, if that utility needs to be run
- The HD Reorganization Reload utility, if secondary indexing or logical relationships exist
- The Database Prefix Resolution utility, after the database is loaded or reloaded

The Prereorganization utility also produces a list of which databases not being initially loaded or reorganized contain segments involved in logical relationships with the database that is being initially loaded or reorganized.

This utility is always run before the database is loaded (for initial load) or reloaded (for reorganization).

Database Scan utility (DFSURGS0)

You use the Database Scan utility to scan databases that are not being initially loaded or reorganized but contain segments involved in logical relationships with databases that are being initially loaded or reorganized.

The following figure shows the input to and output from the Database Scan utility.
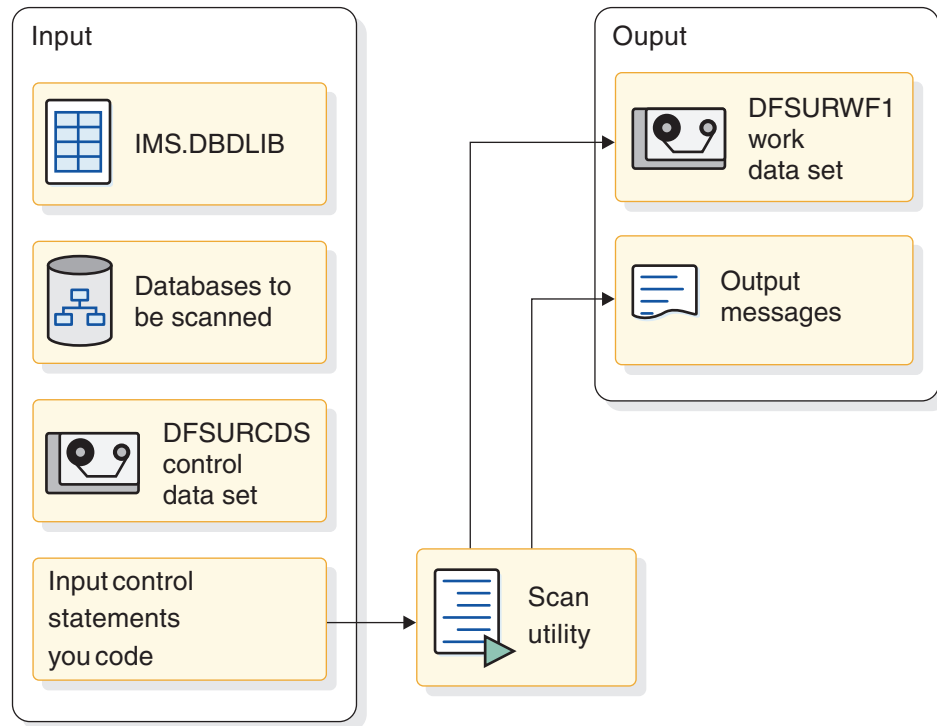
Figure 255. Database Scan utility (DFSURGS0)

For input, the utility uses the DFSURCDS control data set created by the Prereorganization utility. For output, the utility produces the DFSURWF1 work data set, which contains prefix information needed to resolve logical relationships. The DFSURWF1 work data set is used as input to the Database Prefix Resolution utility.

This utility is always run before the database is loaded (for initial load) or reloaded (for reorganization).

Database Prefix Resolution utility (DFSURG10)

You use the Prefix Resolution utility to accumulate and sort the information that has been put on DFSURWF1 work data sets up to this point in the load or reload process.

The following figure shows the input to and output from the Database Prefix Resolution utility.
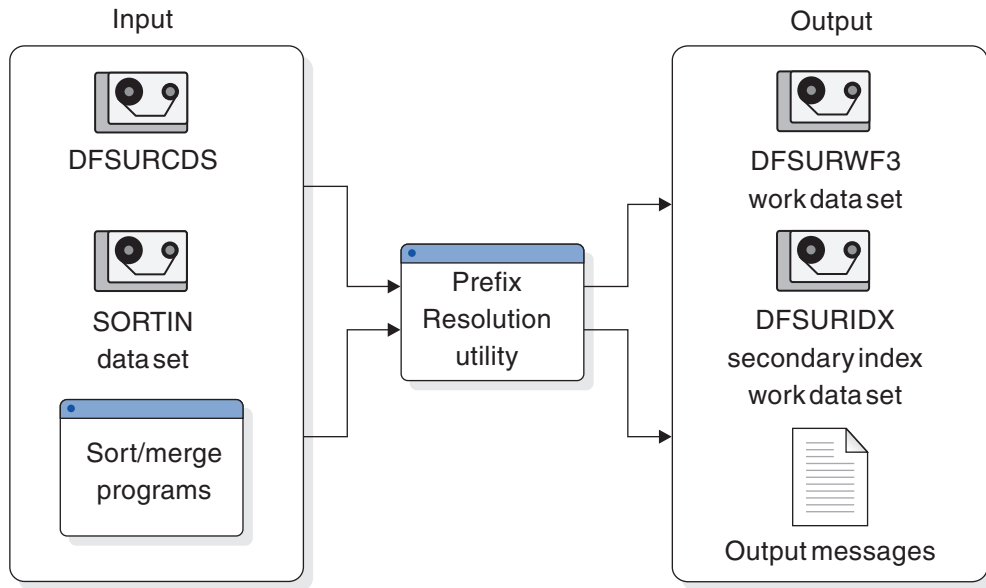
Figure 256. Database Prefix Resolution utility (DFSURG10)

The various work data sets that could be input to the DFSURG10 utility are:

- The DFSURCDS control data set produced by the Prereorganization utility
- The DFSURWF1 work data set produced by the scan utility
- The DFSURWF1 work data set produced by the HD Reorganization Reload utility

The DFSURWF1 work data sets must be concatenated to form an input data set for the Prefix Resolution utility. The name of the input data set is SORTIN.

The Prefix Resolution utility uses the z/OS sort/merge programs to sort the information that has been accumulated. For output, the utility produces the DFSURWF3 work data set, which contains the sorted prefix information needed to resolve logical relationships. The DFSURWF3 data set will become input to the Database Prefix Update utility.

If secondary indexes exist, the utility produces the DFSURIDX work data set, which contains the information needed to create a new secondary index or update a shared secondary index database. The DFSURIDX work data set is used as input to the HISAM unload utility. The HISAM unload utility formats the secondary index information before the HISAM reload utility creates a secondary index or updates a shared secondary index database.

This utility is always run after the database is loaded (for initial load) or reloaded (for reorganization).

Database Prefix Update utility (DFSURGP0)

You use the Prefix Update utility to update the prefix of each segment whose prefix was affected by the initial loading or reorganization of the database.

The following figure shows the input to and output from the Database Prefix Update utility.
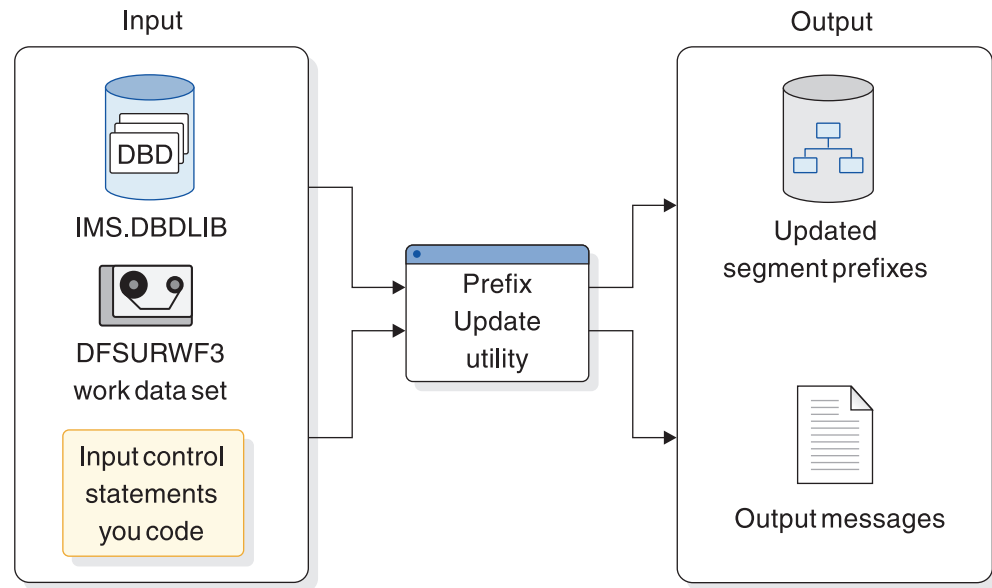
Figure 257. Database Prefix Update utility (DFSURGP0)

The prefix fields that are updated include the logical parent, logical twin, and logical child pointer fields, and the counter fields for logical parents. The Prefix Update utility uses as input the DFSURWF3 data set created by the Prefix Resolution utility.

This utility is always run after the database is loaded (for initial load) or reloaded (for reorganization) and after the Prefix Resolution utility has been run.

Using HISAM unload and reload utilities for secondary indexing operations

In addition to unloading and reloading a database, you can also use the HISAM unload and reload utilities to build secondary indexes and work with secondary indexes that are a part of a shared secondary index.

Specifically, you can use the HISAM unload and reload utilities to:
- Build a secondary index database
- Merge a secondary index into a shared secondary index database
- Replace a secondary index in a shared secondary index database
- Extract a secondary index from a shared secondary index database

Each of these operations is done separately. That is, none of them can be done in conjunction with running the HISAM unload and reload utilities to unload or reload a regular database.

The following figure shows the input to and output from the HISAM unload and reload utilities when performing the first three operations. The DFSURIDX work data set used as input to the HISAM unload utility was created by the Prefix Resolution utility. It contains the information needed to create or update a shared secondary index database. The HISAM unload utility formats the secondary index information for use by the HISAM reload utility. Note that the input control statement to the HISAM unload utility has an X in position 1 when the utility is used for secondary indexing operations rather than for unloading a regular database. Position 3 contains one of the following characters:

- M: means the operation is either to build a new secondary index database or merge a secondary index into a shared secondary index database
- R: means the operation is to replace a secondary index into a shared secondary index database

The HISAM reload utility uses the output from the HISAM unload utility to create the new secondary index or merge or replace the secondary index in a shared secondary index database.
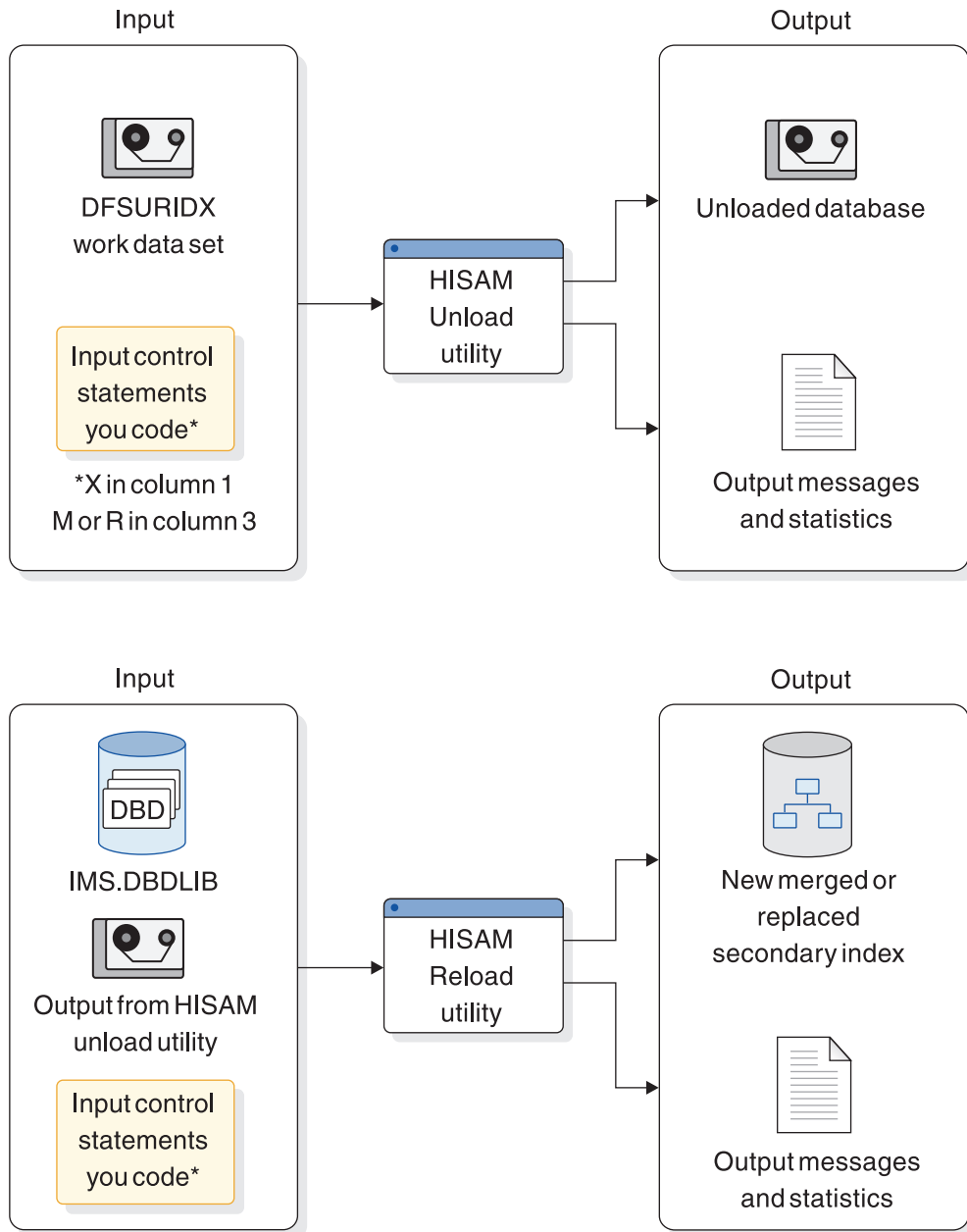
Figure 258. HISAM reorganization unload and reload utilities used for create, merge, or replace secondary indexing operations

The following figure shows the input to and output from the HISAM unload utility when an index is being extracted from a set of shared indexes. Note that the input can be one of the following:

* The DFSURIDX work data set created by the Prefix Resolution utility
* The shared secondary index database

Again, position 1 in the input control statement contains an X. Position 3 contains an E, which means the operation is to extract a secondary index.
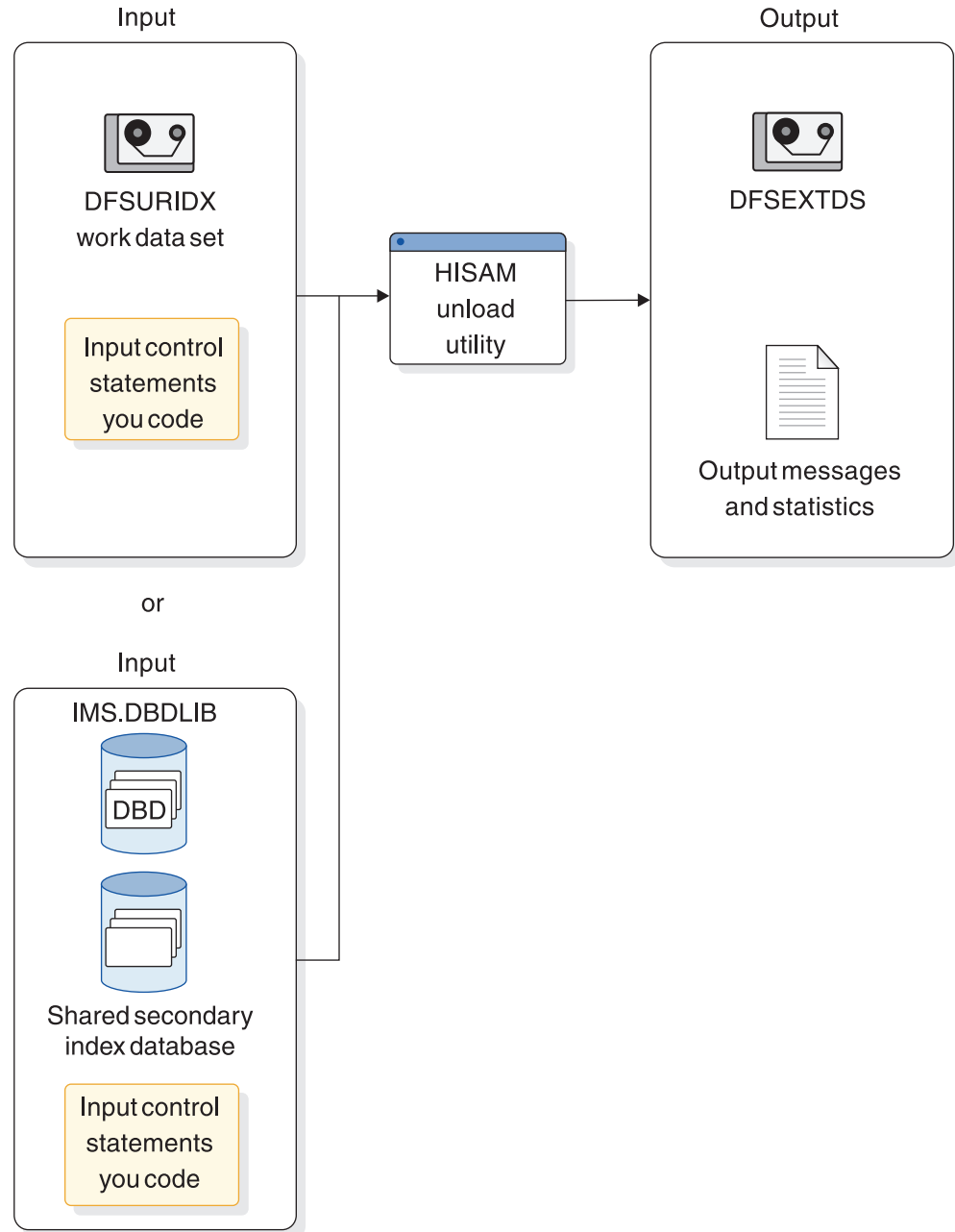
Figure 259. HISAM Reorganization Unload utility used for extract secondary indexing operations

Utility Control Facility (DFSUCF00)

The Utility Control Facility is a program that controls the execution of reorganization and recovery utilities.

Control here means that it generates many of the JCL statements you must create and eliminates the need to sequence the various utilities for execution. The only reorganization utilities that cannot be run under the control of UCF are the Database Surveyor utility and the Partial Database Reorganization utility. In addition to controlling the execution of other utilities, UCF allows you to stop and then later restart a job.

Database Surveyor utility (DFSPRSUR)

Use the Surveyor utility to scan all or part of an HDAM or a HIDAM database to determine whether a reorganization is needed.

The following figure shows the input to and output from the Database Surveyor utility.
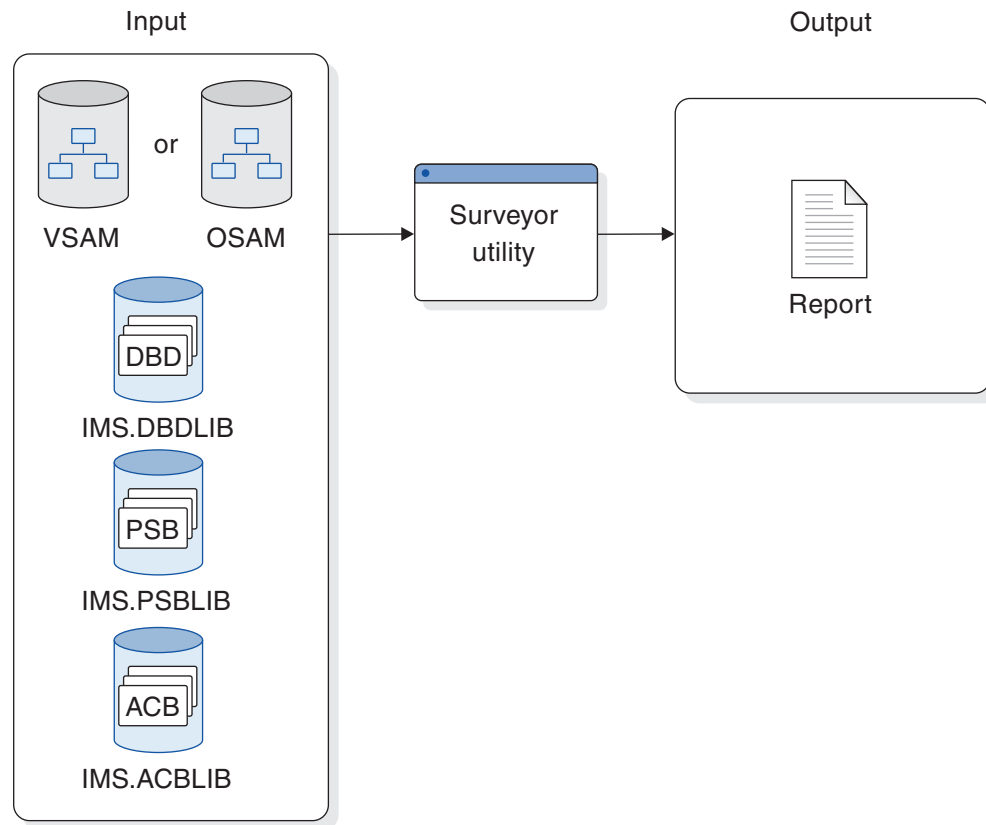
Figure 260. Database Surveyor utility (DFSPRSUR)

The Surveyor utility produces a report describing the physical organization of the database. The report includes the size and location of areas of free space. When you do a partial reorganization, you will know where free space exists into which you can put your reorganized database records.

Related concepts:

"Partial Database Reorganization utility (DFSPRCT1)"

Partial Database Reorganization utility (DFSPRCT1)

You can use the Partial Database Reorganization utility to reorganize parts of your HD database. It can be used when HD databases use secondary indexes or logical relationships.

You tell the utility what range of records you need reorganized.

- In an HDAM database, a range is a group of database records with continuous relative block numbers.
- In a HIDAM database, a range is a group of database records with continuous key values.

Generally, before using the Partial Database Reorganization utility, you would run the Database Surveyor utility. The Surveyor utility helps you determine whether a reorganization is needed and find the location and size of areas of free space. You need to know the location and size of areas of free space so you will know where to put reorganized database records.

The Partial Database Reorganization utility reorganizes the database in two steps:

1. In the first step, the utility produces control tables for use in Step 2, which is when the actual reorganization is done. As an option, the utility can produce PSB source statements for creating a PSB for use in Step 2. The utility also generates reports that show which logically related segments in logically related databases must be scanned in Step 2, and which can be optionally scanned in Step 2. (Some GSAM databases are involved in Step 2 for which a PSB is needed.)

2. In the second step, the utility does the actual reorganization. The database records you have specified are unloaded to a data set. The space they occupied in the database is freed. Then database records are reloaded into the database in the range of free space you specified. Finally, all pointers to database records with new locations are changed to point to the new location. A report is produced at the end of Step 2 to tell you what was done.

The following figure shows the input to and output from the Partial Database Reorganization utility.
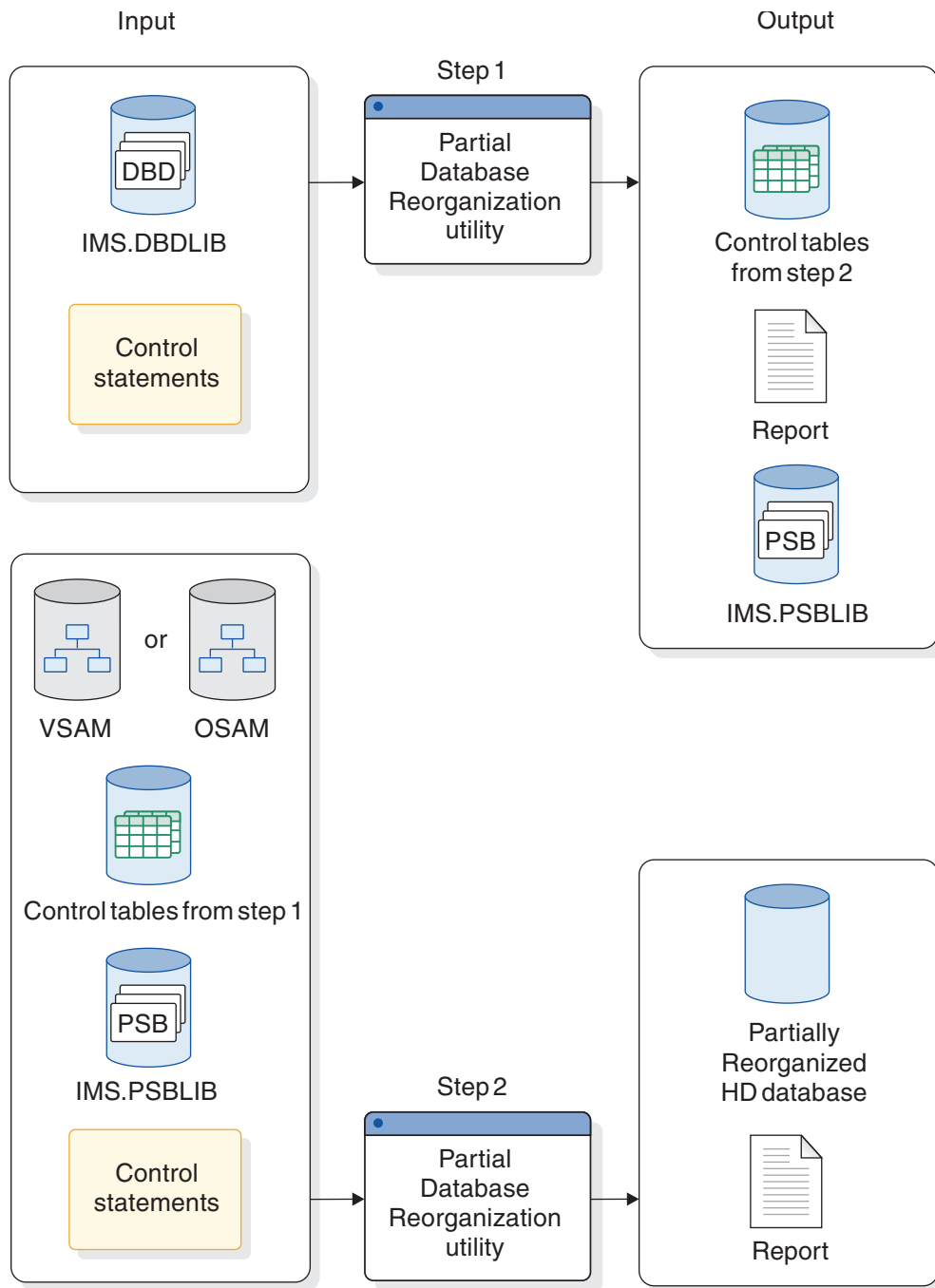
Figure 261. Partial Database Reorganization utility (DFSPRCT1)

Related concepts:

"Database Surveyor utility (DFSPRSUR)" on page 660

"Planning for maintenance" on page 457

Reorganizing HISAM, HD, and index databases offline

This topic describes how to reorganize offline HISAM, HDAM, HIDAM, primary index, and secondary index database types.

Reorganizing a HISAM database (no secondary indexes)

To reorganize a HISAM database when it does not use logical relationships or secondary indexes perform the following steps.

1. Unload the database using the HISAM Reorganization Unload utility.
2. Any time you unload a data set, you should delete and reallocate the data set before reloading.
3. Reload the database using the HISAM Reorganization Reload utility.
4. Make an image copy of your database once it is reloaded.

Reorganizing an HDAM or HIDAM database (no logical relationships or secondary indexes)

To reorganize an HDAM or HIDAM database that does not use logical relationships or secondary indexes perform the following steps.

1. Unload the database using the HD Reorganization Unload utility.
2. Any time you unload a data set, you should delete and reallocate the data set before reloading.
3. Reload the database using the HD Reorganization Reload utility.
4. Make an image copy of your database after it is reloaded.

Reorganizing a primary or secondary index

A HIDAM primary index and a secondary index of a HISAM, HDAM, or HIDAM database are reorganized in the same way.

To reorganize a HIDAM primary index or a HISAM, HDAM, or HIDAM secondary index perform the following steps.

1. Unload the index database using the HISAM Reorganization Unload utility.
2. Any time you unload a data set, you should delete and reallocate the data set before reloading.
3. Reload the index database using the HISAM Reorganization Reload utility. Make an image copy of your database as soon as it is reloaded.

Reorganizing HALDB databases

One of the primary advantages of HALDB is its simplified and shortened reorganization process and the ability to reorganize HALDB databases online using the integrated HALDB Online Reorganization function.

Both PHDAM and PHIDAM HALDB databases can be reorganized online or offline. A PSINDEX HALDB can be reorganized only offline. Whether you are reorganizing your HALDB online or offline, the reorganization process is different from the reorganization processes used for other full-function databases.

Reorganizations of HALDB databases with logical relationships and secondary indexes do not require the execution of utilities to update these pointers. Instead, HALDB uses a self-healing pointer process to correct these pointers when they are used.

Related concepts:

"Implementing HALDB design" on page 541

HALDB offline reorganization

The offline reorganization processes for a HALDB database and other full-function IMS databases are similar: they both consist of an unload and reload of the database.

The HALDB offline reorganization process has advantages over the reorganization process of other full-function databases, such as:

- You can reorganize one HALDB partition at a time or reorganize multiple partitions in parallel.
- The self-healing pointer process of HALDB databases eliminates the need to manually update logical relationships and secondary indexes after reorganizing a HALDB database.
- You do not need to include DD statements for HALDB data sets when you reorganize a HALDB database. HALDB data sets are dynamically allocated.

Related tasks:

"Offline Reorganizations after HALDB Online Reorganizations" on page 690

Overview of HALDB offline reorganization

An offline reorganization of a HALDB database can be done with one or more parallel processes. These processes unload one or more partitions and reload them.

If the database has secondary indexes or logical relationships, additional steps are not required. The HALDB self-healing process makes updates of pointers during the reorganization unnecessary. The amount of time required for a reorganization depends on the sizes of the partitions. Smaller partitions reduce the time. You can reduce your reorganization time by creating more partitions and reorganizing them in parallel.

The basic steps involved in reorganizing a HALDB database offline are:

1. Run the HD Reorganization Unload utility (DFSURGU0) to unload the entire database, a range of partitions, or a single partition.
2. Optionally, initialize the partitions by running either of the following utilities:
   - HALDB Partition Data Set Initialization utility (DFSUPNT0)
   - Database Prereorganization utility (DFSURPR0)
3. Run the HD Reorganization Reload utility to reload the database or partitions.
4. Make image copies of all reloaded partition data sets.

The following figure shows the offline processes used to reorganize a HALDB database with logical relationships and secondary indexes. In this case, the partitions are reorganized by parallel processes. Each partition can be unloaded and reloaded in less time than unloading and reloading the entire database. This is much faster than the process for a non-HALDB full-function database. Additionally, no time is required for updating pointers in the logically related database or rebuilding secondary indexes. This further shortens the process.
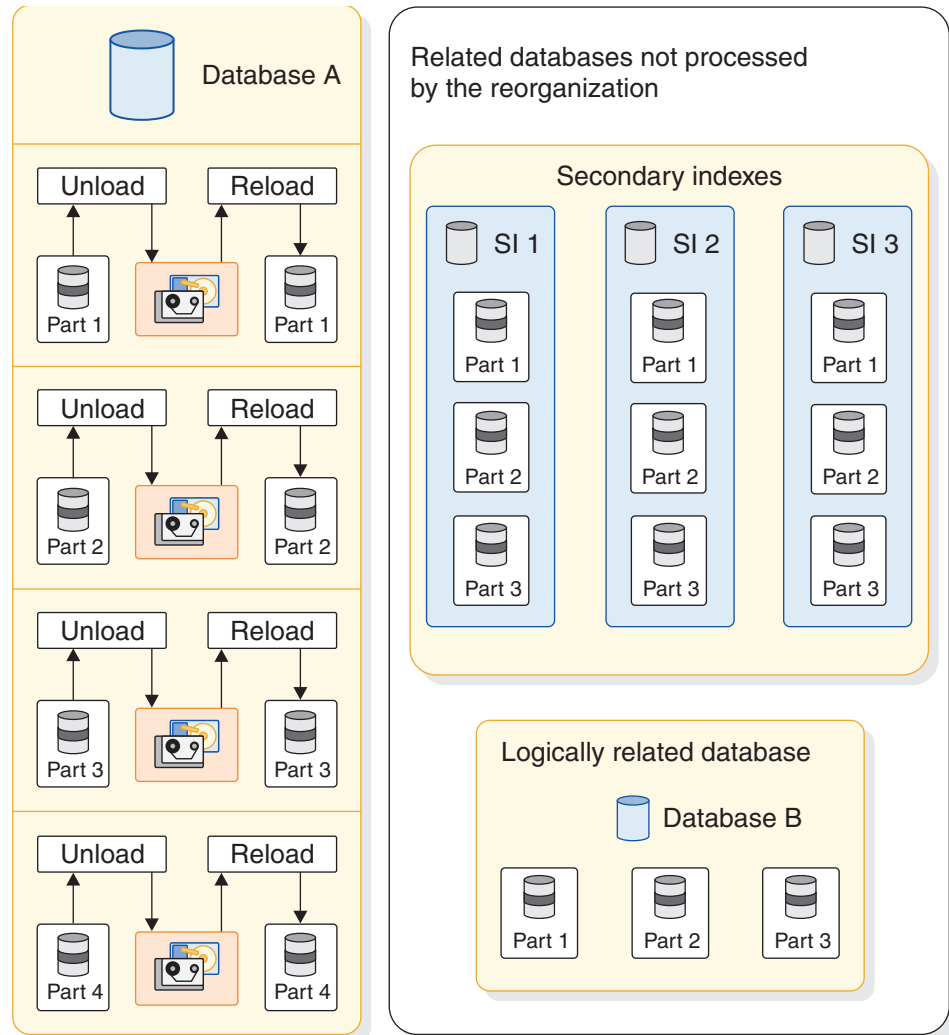
Figure 262. Offline reorganization of a HALDB database

Related concepts:

"Reorganizing HALDB partitioned secondary index databases" on page 669

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Options for offline reorganization of HALDB databases

You have several options when reorganizing a HALDB database.

Your options for reorganizing a HALDB database include:

- You can reorganize any number of partitions. If you need to reorganize only one partition, you can unload and reload it without processing other partitions.
- You can reorganize partitions in parallel or you can reorganize the database with one process. The degree of parallelism is determined by the number of reorganization jobs you run. Each job can process one or multiple partitions. To increase the parallelism, you can increase the number of reorganization jobs and decrease the number of partitions each job processes.
- You can reuse existing database data sets or you can delete them after they are unloaded and allocate new data sets for the reload.
- You can add partitions, delete partitions, or change partition boundaries.

Related tasks:

Unloading HALDB partitions and databases for offline reorganization

HALDB partitions or databases can be unloaded with the HD Reorganization Unload utility (DFSURGU0).

To unload an entire HALDB database, do not include a SYSIN DD statement. To unload any other number of partitions, you must include a control statement in your SYSIN data set. The control statement identifies the name of the first partition to unload and, if you are unloading more than one partition, the number of partitions to unload.

Multiple partitions are unloaded consecutively. For key range partitioning, consecutive partitions are determined by the high keys. If you are using a partition selection exit routine, consecutive partitions are determined by the order assigned by the exit routine.

Do not include DD statements for the HALDB database data sets. The HD Reorganization Unload utility uses dynamic allocation for HALDB data sets. This is not true for non-HALDB databases.

Requirement: You must supply buffer pools for all data sets in the partitions that are unloaded. This includes the ILDSs.

Recommendation: Enable OSAM sequential buffering for databases that use OSAM.

The following code is an example of the HD Reorganization Unload utility control statement to unload one partition.

```
SYSIN DD *
PARTITION=PE002
```

The following code shows the HD Reorganization Unload utility control statement to unload three partitions.

```
SYSIN DD *
PARTITION=PE004,NUMBER=3
```

The following JCL shows a sample job that unloads a HALDB partition.

```
//JOUKO3C JOB  (999,POK),JOUKO3,CLASS=A,NOTIFY=&SYSUID,
//        MSGLEVEL=(1,1),MSGCLASS=X,REGION=0M
//JOBLIB    DD DSN=IMSPSA.IMS0.SDFSRESL,DISP=SHR
//          DD DSN=IMSPSA.IM0A.MDALIB,DISP=SHR
//*******************************************************************
//*   HD UNLOAD FOR THE PARTITION PE002 OF PEOPLE DATABASE
//*******************************************************************
//UNLOAD   EXEC PGM=DFSRRC00,REGION=1024K,
//         PARM='ULU,DFSURGU0,PEOPLE,,,,,,,,,,,,Y,N'
//DFSRESLB  DD DSN=IMSPSA.IMS0.SDFSRESL,DISP=SHR
//IMS       DD DISP=SHR,DSN=JOUKO3.HALDB.DBDLIB
//DFSURGU1 DD DSN=JOUKO3.UNLOAD.PE002,UNIT=3390,VOL=SER=TOTIMN,
//         SPACE=(CYL,(10,5),RLSE),DISP=(NEW,CATLG)
//DFSVSAMP DD *
IOBF=(4096,50)
VSRBF=8192,50
```

```
/*
//SYSPRINT DD SYSOUT=*
//DFSCTL    DD *
SBPARM ACTIV=COND
/*
//SYSIN   DD *
PARTITION=PEO02
/*
```

The IMS High Performance Unload tool is an alternative to the HD Reorganization Unload utility. You can use the High Performance Unload tool to unload any number of partitions or the entire database.

Related Reading: For more information on the High Performance Unload tool, see *IMS High Performance Unload for z/OS User's Guide*.

Reallocating HALDB database data sets for offline reorganization

You do not have to delete and redefine HALDB database data sets before you reload them. This applies to both OSAM and VSAM data sets.

VSAM data sets, other than ILDSs, must be specified with the REUSE option. HALDB supports this option.

If you delete and redefine partition data sets, but do not reload data into them, you must initialize the partition data sets. If you reload data into the partition data sets after deleting and redefining them, you do not need to initialize the partition data sets.

If you delete and redefine VSAM data sets, you receive a z/OS IEC161I system message when reloading a partition. This is not an error message. It indicates that a VSAM data set was empty when it was opened. The following example shows the message for an ILDS.

```
IEC161I 152-061,JOUKO3D,RELOAD,PEO01L,,,
IEC161I JOUKO3.HALDB.DB.PEOPLE.L00001,
IEC161I JOUKO3.HALDB.DB.PEOPLE.L00001.DATA,CATALOG.TOTICF2.VTOTCAT
```

Related Reading: For more information on IEC system messages, see *z/OS MVS System Messages, Vol 7 (IEB-IEE)*.

Reloading HALDB partitions and databases for offline reorganization

HALDB partitions and databases can be reloaded with the HD Reorganization Reload utility (DFSURGL0).

The HD Reorganization Reload utility reads the output file from the HD Reorganization Unload utility. You do not specify the partitions to be reloaded. They are determined by the records in the input file to the HD Reorganization Reload utility.

Do not include DD statements for the HALDB database data sets. The HD Reorganization Reload utility uses dynamic allocation for HALDB data sets. This is not true for non-HALDB databases.

The data sets allocated for the HD Reorganization Reload utility are always the A–J data sets, even when the active data sets prior to the offline reorganization were the M–V data sets as a result of using the HALDB Online Reorganization function previously.

You must supply buffer pools for all data sets in the partitions that are reloaded. This includes the ILDSs.

The HD Reorganization Reload utility sets the image copy needed flag for data sets in partitions that it loads. You should image copy them as you would any database data sets after they have been reloaded.

The following code shows a sample job that reloads HALDB partitions. The partitions it reloads depend on the records in the input file.

```
//JOUKO3D JOB  (999,POK),JOUKO3,CLASS=A,NOTIFY=&SYSUID,
//         MSGLEVEL=(1,1),MSGCLASS=X,REGION=0M
//JOBLIB    DD DSN=IMSPSA.IMS0.SDFSRESL,DISP=SHR
//          DD DSN=IMSPSA.IM0A.MDALIB,DISP=SHR
//********************************************************************
//*   HD RELOAD FOR THE PEOPLE DATABASE
//********************************************************************
//RELOAD   EXEC PGM=DFSRRC00,REGION=1024K,
//         PARM='ULU,DFSURGL0,PEOPLE,,,,,,,,,,,Y,N'
//DFSRESLB  DD DSN=IMSPSA.IMS0.SDFSRESL,DISP=SHR
//IMS       DD DISP=SHR,DSN=JOUKO3.HALDB.DBDLIB
//DFSUINPT DD DSN=JOUKO3.UNLOAD.PEOPLE,DISP=OLD
//DFSVSAMP DD *
VSRBF=8192,50
IOBF=(4096,50)
/*
//SYSPRINT DD SYSOUT=*
//DFSSTAT  DD SYSOUT=*
```

ILDS reorganization updates:

The HD Reorganization Reload utility updates the ILDS for partitions that contain targets of logical relationships or secondary indexes.

The utility has three options for updating an ILDS after a reorganization:
- No control statement
- ILDSINGLE control statement
- NOILDS control statement

If you do not specify a control statement in the SYSIN data for the HD Reorganization Reload utility, an ILDS entry is updated or created when a target of a secondary index or logical relationship is inserted in the partition. An entry exists if a previous reorganization loaded the target segment in the partition. The updates to the ILDS are done in VSAM update mode. When a CI or CA is filled, it must be split by VSAM. Free space in the ILDS can help avoid these splits. Updates can be random or sequential. This depends on the order in which these segments are inserted and their ILKs. The ILDS keys are based on the ILK that is based on the location of the target segment when it was created.

You can create free space in the ILDS by specifying the ILDSINGLE control statement. The ILDSINGLE option provides a number of performance benefits, including eliminating CI and CA splits, eliminating unused ILEs, and better performance for subsequent reorganizations and recoveries. The ILDS must be deleted and redefined prior to running the utility with the ILDSINGLE option.

If you specify the NOILDS control statement in the SYSIN data, the HD Reorganization Reload utility does not update or create entries in the ILDSs. You must rebuild the ILDS by using some other means, such as the HALDB Index/ILDS Rebuild utility (DFSPREC0).

Using the NOILDS control statement provides the fastest reload, and the HALDB Index/ILDS Rebuild utility can update the ILDS for each partition in parallel; however, the HALDB Index/ILDS Rebuild utility reads each partition to update its ILDS. Optionally, the HALDB Index/ILDS Rebuild utility can rebuild the ILDS in VSAM load mode, which can improve performance and includes free space in the ILDS.

The HD Reorganization Reload utility also has an ILDSMULTI control statement; however, ILDSMULTI applies only to migration reloads. For more information about ILDSMULTI, see the HD Reorganization Reload utility section of *IMS Version 15 Database Utilities*.

Reorganizing HALDB partitioned secondary index databases

You might need to reorganize your partitioned secondary index (PSINDEX) database. Because the reorganization of HALDB databases does not require the recreation of their secondary indexes, a PSINDEX database can become disorganized as entries are added to it over time.

The HD Reorganization Unload utility and the HD Reorganization Reload utility can be used to reorganize PSINDEX databases. The restrictions and recommendations for reorganizing other HALDB databases also apply to PSINDEX databases with one exception: HALDB secondary indexes have no ILDSs. The HD Reorganization Reload utility control statements should not be used with secondary indexes.

The steps for reorganizing a PSINDEX database are the same as those for reorganizing other types of HALDB databases offline.

Related tasks:

"Overview of HALDB offline reorganization" on page 664

HALDB online reorganization

The integrated HALDB Online Reorganization function of IMS allows HALDB partitions to remain online and available for IMS application programs during a database reorganization.

HALDB Online Reorganization provides reclustering and space distribution advantages that improve performance.

An online reorganization of a PHDAM or PHIDAM HALDB partition runs non-disruptively, allowing concurrent IMS updates, including updates by data-sharing IMS systems. The online reorganization is non-disruptive because IMS copies small amounts of data from the partition's original data sets (the input data sets) to separate output data sets. IMS tracks which data has been copied so that IMS applications can automatically retrieve or update data from the correct set of data sets.

HALDB Online Reorganization extends the established data definition and data set naming convention for HALDB databases. The data set groups in a HALDB database use the characters A-through-J in the DDNAMEs and data set names of the ten supported data set groups. The primary index for a PHIDAM database uses the character X in these names. This data definition and data set naming convention is extended so that IMS uses the characters M-through-V (and Y) for an alternate set of data sets.

The initial load or offline reorganization reload of a HALDB partition always uses the A-through-J (and X) data sets. Until the first time that you reorganize a HALDB partition online, only the A-through-J (and X) data sets are used.

There are three phases of online reorganization for a HALDB partition:

1. The initialization phase, during which IMS prepares the output data sets and updates the RECON data sets.
2. The copying phase, during which IMS performs the actual reorganization by copying the data from the input data sets to the output data sets.
3. The termination phase, during which IMS closes the input data sets and updates the RECON data sets.

The HALDB online reorganization function can also be used to make certain changes to the structure of an online HALDB database. Additional restrictions and requirements apply when using the alter option of the HALDB online reorganization function. For more information about the alter option, see "Altering the definition of an online HALDB database" on page 757.

Related concepts:

"Planning for maintenance" on page 457

"Reorganizing the database" on page 643

"DL/I calls that can be issued against HD databases" on page 146

"Data set naming conventions for HALDB Online Reorganization" on page 677

"Online reorganization processing when the ALTER option is specified" on page 765

Related tasks:

"Estimating the minimum size of the database" on page 559

"Altering the definition of an online HALDB database" on page 757

The initialization phase for HALDB Online Reorganization

You start the online reorganization of a HALDB partition using the `INITIATE OLREORG` command.

During the initialization phase, IMS updates the RECON data sets to establish the ownership of the online reorganization by the IMS system that is performing the online reorganization. This ownership means that no other IMS system can perform a reorganization of the HALDB partition until the current online reorganization is complete or until ownership is transferred to another IMS system. IMS adds the M-V (and Y) DBDSs to the RECON data sets if those DBDS records do not already exist. IMS also adds the M-V (and Y) DBDSs to any existing change accumulation groups and DBDS groups that include the corresponding A-J (and X) DBDSs.

Before online reorganization begins for a HALDB partition, there is a single set of *active* data sets for the HALDB partition. These active data sets are the input data sets for the copying phase. There might also be a set of *inactive* data sets from a prior online reorganization that are not used by IMS application programs.

During the initialization phase, IMS evaluates each of the inactive data sets to ensure that it meets the requirements for output data sets. If any of the output data sets does not exist, IMS creates it automatically during this phase.

At the end of the initialization phase, IMS treats the original active set of data sets as the input set and the inactive data sets as the output set. This use of the input

and output sets of data sets is represented by the *cursor-active* status for the partition, which is recorded in online reorganization records in the RECON data sets. A listing of the partition's database record in the RECON data sets shows `OLREORG CURSOR ACTIVE=YES`. A listing of the partition also shows that both sets of DBDSs are active: the first set of DBDSs listed is for the input data set and the second set of DBDSs is for the output data set, for example, `DBDS ACTIVE=A-J` and `M-V`. While the partition is in the cursor-active status, both sets of data sets must be available for the partition to be processed by any application.

The following example shows part of a listing of the RECON data sets for a HALDB partition that has the cursor-active status.

```
--------------------------------------------------------------------------------
04.174 12:30:54.1                      LISTING OF RECON              PAGE 0003
--------------------------------------------------------------------------------
 DB
  DBD=POHIDKA   MASTER DB=DBOHIDK5  IRLMID=*NULL  CHANGE#=6     TYPE=PART
  USID=0000000004  AUTHORIZED USID=0000000004  HARD USID=0000000004
  RECEIVE USID=0000000004  RECEIVE NEEDED USID=0000000000
  DBRCVGRP=**NULL**
  DSN PREFIX=IMSTESTS.DBOHIDK5                            PARTITION ID=00001
  PREVIOUS PARTITION=**NULL**    NEXT PARTITION=POHIDKB
  OLRIMSID=IMS1        ACTIVE DBDS=A-J and M-V
  REORG#=00004

  FREE SPACE:
    FREE BLOCK FREQ FACTOR=2    FREE SPACE PERCENTAGE=20

  PARTITION HIGH KEY/STRING (CHAR):              (LENGTH=5  )
    K2000
  PARTITION HIGH KEY/STRING (HEX):
    D2F2F0F0F04040404040404040404040404040404040404040404040404040404040

  OSAM BLOCK SIZE:
    A = 4096
    B = 4096

  FLAGS:                               COUNTERS:
    BACKOUT NEEDED       =OFF            RECOVERY NEEDED COUNT   =0
    READ ONLY            =OFF            IMAGE COPY NEEDED COUNT =0
    PROHIBIT AUTHORIZATION=OFF           AUTHORIZED SUBSYSTEMS   =1
                                        HELD AUTHORIZATION STATE=6
                                        EEQE COUNT              =0
    TRACKING SUSPENDED   =NO             RECEIVE REQUIRED COUNT  =0
    OFR REQUIRED         =NO             OLR ACTIVE HARD COUNT   =0
    PARTITION INIT NEEDED =NO            OLR INACTIVE HARD COUNT =0
    OLREORG CURSOR ACTIVE =YES
    PARTITION DISABLED   =NO
    ONLINE REORG CAPABLE =YES
    REORG INTENT         =NO
    QUIESCE IN PROGRESS  =NO
    QUIESCE HELD         =NO
    ALTER IN PROGRESS    =NO
    ALTER COMPLETE       =NO

  ASSOCIATED SUBSYSTEM INFORMATION:
                            ENCODED  B/O NEEDED
    -SSID-   -ACCESS INTENT- -STATE-  -COUNT- -SS ROLE-
    IMS1        UPDATE         6         0      ACTIVE
--------------------------------------------------------------------------------
```

During the initialization phase, various error conditions, such as an unacceptable preexisting data set or an insufficient amount of disk space for an automatically created data set, can cause the initialization to fail. However, if an error occurs during or after the data set creation and validation process, but before IMS records

the cursor-active status in the RECON data sets, any automatically created output data sets are retained along with any preexisting ones.

Also during the initialization phase of an online reorganization, IMS dynamically creates a program specification block (PSB) for the reorganization. The PSB name is the 7-character HALDB partition name prefixed with the single character zero ('0'). For example, a HALDB partition with the name SSN5603 has a dynamic PSB with the name 0SSN5603 for the reorganization work. This PSB does not exist in the PSBLIB or the ACBLIB, but the name can appear in a listing of RECON or in output from utilities.

Related concepts:

"Requirements for existing output data sets" on page 678

Related reference:

↪ INITIATE OLREORG command (Commands)

The copying phase for HALDB Online Reorganization

During the copying phase of a HALDB online reorganization, the HALDB partition comprises the A-through-J (and X) data sets and the M-through-V (and Y) data sets and both sets of data sets must be available in order for IMS applications to access the partition.

While IMS reorganizes a HALDB partition online, IMS applications can make database updates to the partition. Some of the database updates are made to the input data sets, while others are made to the output data sets, depending on which data is updated by the application. Which data sets are updated is transparent to the application program. The following figure illustrates the relationship between the input and output data sets at a point during the online reorganization.
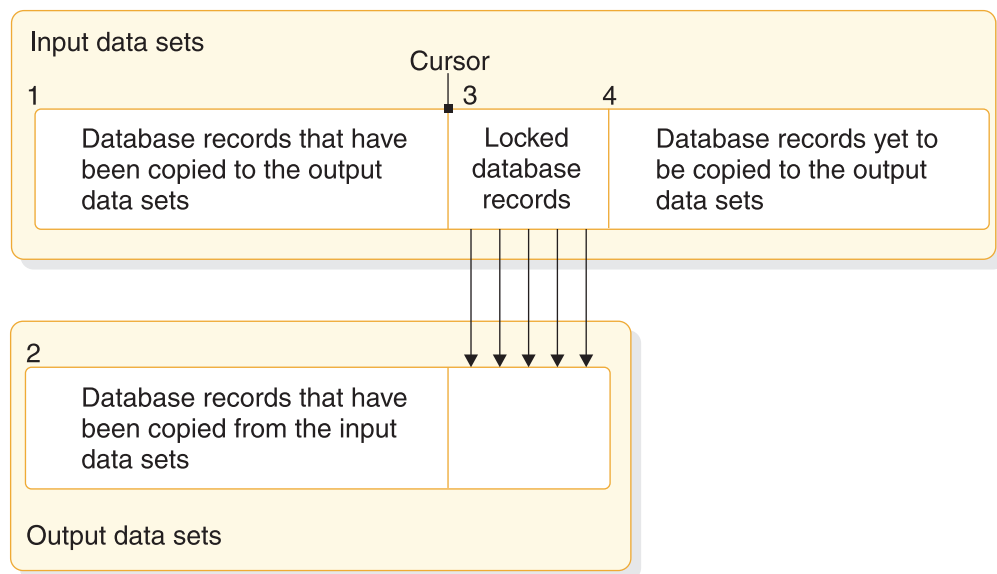
Figure 263. The relationship between input data sets and output data sets during the online reorganization of a HALDB partition

The previous figure shows two sets of database data sets for a HALDB partition, the input data sets that have not been reorganized and the output data sets that have been (at least partially) reorganized. The figure shows the reorganization as progressing from left to right, from the input data sets above to the output data sets below. The data sets in the figure are divided into four areas:

1. Data within the input data sets that has been copied to the output data sets. This area reflects the old data organization (prior to the reorganization), and is not used again by IMS applications until the data sets are reused as the output data sets for a later online reorganization.
2. Data within the output data sets that has been copied from the input data sets. This data in this area has been reorganized, and can be used by IMS applications during the reorganization.
3. Data within both the input and output data sets that is locked and in the process of being copied and reorganized from the input data sets to the output data sets. This area of locked records is called a *unit of reorganization*. From a recovery point of view, this unit of reorganization is equivalent to a unit of recovery.

   While IMS processes the current unit of reorganization, IMS applications that access any of the locked data records must wait until IMS completes the reorganization for those records. After the copying and reorganization completes for the unit of reorganization, IMS commits the changes and unlocks the records, thus making them available again for IMS applications.
4. Data within the input data sets that has not yet been copied to the output data sets. This data has also not yet been reorganized, and can be used by IMS applications during the reorganization.

As the online reorganization progresses, IMS uses a kind of pointer called a *cursor* to mark the end point of those database records that have already been copied from the input data sets to the output data sets. As the reorganization and copying proceeds, this cursor moves through the partition (from left to right in the preceding figure).

When an IMS application program accesses data from a HALDB partition that is being reorganized online, IMS retrieves the data record:
- From the output data sets if the database record is located "at or before" the cursor.
- From the input data sets if the database record is located "after" the cursor.

If the data record happens to fall within the unit of reorganization, IMS retries the data access after the records are unlocked. An application program does not receive an error status code for data within a unit of reorganization.

To allow recovery of either an input data set or an output data set, all database changes are logged during the online reorganization, including the database records that are copied from the input data set to the output data sets.

The termination phase for HALDB Online Reorganization
The online reorganization of a HALDB partition terminates after the end of the copying phase, or when IMS encounters an error condition during the reorganization.

You can also stop the online reorganization of a HALDB partition using the TERMINATE OLREORG command. If multiple partitions are being reorganized, you can terminate the reorganization of all of the partitions by specifying an asterisk in the NAME keyword.

After the copying phase is complete for a HALDB partition, the output data sets become the *active* data sets, and the input data sets become the *inactive* data sets. The active data sets are used for all data access by IMS application programs. The inactive data sets are not used by application programs, but can be reused for a

subsequent online reorganization. Unless you perform an initial load or a batch reorganization reload for the partition, successive online reorganizations for the partition alternate between these two sets of data sets.

IMS updates the partition's database record in the RECON data sets to reset the cursor-active status for the partition to reflect that there is now just one set of data sets. A listing of this record from the RECON data sets shows `OLREORG CURSOR ACTIVE=NO` and the ACTIVE DBDS field shows the active (newly reorganized) data sets. IMS also updates the online reorganization records in the RECON data sets with the time stamp of when the reorganization completed.

If you specified the `DEL` keyword for the `INITIATE OLREORG` command (or the `UPDATE OLREORG` command), IMS deletes the inactive data sets after resetting the cursor-active status for the partition. Before deleting the inactive data sets, IMS notifies all sharing IMS systems, including batch jobs, that the online reorganization is complete and is recorded in the RECON data sets. The IMS system that is performing the online reorganization waits until it receives an acknowledgment from each of these sharing IMS systems that they have closed and deallocated the now-inactive data sets, and then it deletes these data sets. However, if the acknowledgments are not received within 4.5 minutes, the owning IMS system will attempt to delete the inactive data sets anyway. Without the acknowledgments, the deletion attempt is likely to fail.

Finally, at the end of the termination phase, IMS updates the RECON data sets to reset the ownership of the online reorganization so that no IMS system has ownership. This resetting of ownership means that any IMS system can perform a subsequent reorganization of the HALDB database.

If online reorganization of a HALDB partition terminates prior to completion, either because of an error or because you issued the `TERMINATE OLREORG` command, you must restart the online reorganization or perform an offline reorganization for the partition.

The following figure shows the normal processing steps of a successful online reorganization of a HALDB partition. The columns represent the flow of control through the phases of the online reorganization, from the user to IMS, and the status of the data sets as the processing events occur.

User	IMS	Status of data sets

Initialization phase

Invokes INITIATE OLREORG command	Gets ownership from DBRC	
	Performs initial steps (create data sets, etc.)	
	Tells DBRC about cursor-active status	Now two active sets of data sets exists OLREORG CURSOR ACTIVE=YES in RECON listing

Copying phase

| | Copies data from input to output data sets | Database records are locked, copied and then unlocked by UOR |

Termination phase

	When copying is complete, tells DBRC to end cursor-active status	Now only one active set of data sets exists OLREORG CURSOR ACTIVE=NO in RECON listing
	Possibly deletes inactive set of data sets	
	Relinquishes ownership through DBRC	

Figure 264. Normal processing steps of HALDB Online Reorganization

Related tasks:

"Stopping alter processing before completion" on page 771

Related reference:

⇨ INITIATE OLREORG command (Commands)

⇨ TERMINATE commands (Commands)

⇨ UPDATE commands (Commands)

Restrictions for HALDB Online Reorganization

The following restrictions apply to HALDB Online Reorganization

The restrictions include:

- You can perform an online reorganization only for a HALDB that is defined in the RECON data sets as capable of being reorganized online. The OLRCAP parameter of the `INIT.DB` command or the `CHANGE.DB` command defines a HALDB as being OLRCAP.
- A HALDB that uses 8-GB OSAM data sets cannot be reorganized online. To be reorganized online, the HALDB must use 4-GB OSAM data sets. The OSAM8G and NOOSAM8G parameters of the `INIT.DB` command or the `CHANGE.DB`

command defines the maximum data set capacity. OLRCAP and OSAM8G are mutually exclusive parameters and flags in the RECON data sets.

- You cannot start an online reorganization for a partition if another IMS system already owns an online reorganization for that partition. This is the default behavior. To change the default behavior, specify the RELOLROWNER parameter in the DATABASE section of the DFSDFxxx member of the IMS.PROCLIB data set to instruct IMS systems that terminate during an online reorganization to release ownership of the reorganization. Another IMS system can then resume the reorganization. You can also use the OPTION(REL) parameter of the INITIATE OLREORG command to override the default behavior for a single, specific online reorganization.
- You cannot start an online reorganization of any partition that has a status of ALTER IN PROGRESS.
- Image copy for a partition is not allowed if the partition is in the cursor-active status. This restriction applies even if the online reorganization terminated before the cursor-active status has been reset and the online reorganization for the partition is not owned by any IMS.
- Image copy for a partition is not allowed if the partition has an ALTER IN PROGRESS status.
- To backout in-flight work from an online reorganization, you must run a batch backout using a DL/I region type.
- To use a type-2 command to start an online reorganization for a HALDB partition, you must have an IMS Common Service Layer that includes the Operations Manager and the Structured Call Interface.
- HALDB Online Reorganization runs only in a local storage option-subordinate (LSO=S) environment, in which DL/I runs in a separate address space. IMS rejects attempts to initiate an online reorganization for a HALDB partition in a local storage option-yes (LSO=Y) environments.
- You cannot perform an online reorganization for a HALDB partition from an alternate IMS system in an XRF complex. However, after an XRF takeover, the new active IMS system will continue a reorganization that was active when the takeover process began.
- You cannot issue the following commands for a HALDB partition while it is being reorganized online:
  - `/START DATABASE` or `UPDATE DATABASE NAME(`*name*`) START(ACCESS)`
  - `/DBRECOVERY DATABASE` or `UPDATE DATABASE NAME(`*name*`) STOP(ACCESS)`
  - `/DBDUMP DATABASE` or `UPDATE DATABASE NAME(`*name*`) STOP(UPDATES)`
  - `/STOP DATABASE` or `UPDATE DATABASE NAME(`*name*`) STOP(SCHD)`

  If you issue any of these commands for a HALDB partition that is actively being reorganized online, IMS displays error message DFS0488I with return code 58 and does not process the command for the named partition.
- You cannot issue the following commands for a HALDB master while any of its partitions is being reorganized online:
  - `/START DATABASE ACCESS UP` or `UPDATE DATABASE NAME(`*name*`) START(ACCESS)`
  - `/DBRECOVERY DATABASE` or `UPDATE DATABASE NAME(`*name*`) STOP(ACCESS)`
  - `/DBDUMP DATABASE` or `UPDATE DATABASE NAME(`*name*`) STOP(UPDATES)`

Related concepts:

➡ Type-2 command environment (System Administration)

➡ Using a DL/I separate address space (System Definition)

Related reference:

Data set naming conventions for HALDB Online Reorganization

The data sets for HALDB partitions use a specified naming convention. HALDB Online Reorganization extends this naming convention to include a second set of data sets.

Data sets for partitions that are involved in HALDB Online Reorganization use the following naming convention: *bbbbbbb.dppppp*

bbbbbbb
Represents a data set name prefix of up to 37 characters that you defined using the HALDB Partition Definition utility or DBRC batch command (INIT.DB, INIT.PART, CHANGE.DB, or CHANGE.PART). The same data set base name is used for every data set within a HALDB partition.

d
Represents an IMS-assigned data set name type character that uniquely identifies a specific data set for a HALDB partition. The possible single-character values are:

A-through-J
"A" corresponds to the first, or possibly only, data set group that is defined in the DBD, "B" corresponds to the second data set group, and so on. The use of the characters A-through-J applies generally to HALDB partitions, regardless of whether they are capable of being reorganized online.

M-through-V
"M" corresponds to the first, or possibly only, data set group that is defined in the DBD, "N" corresponds to the second data set group, and so on. The use of the characters M-through-V applies only to HALDB partitions that are capable of being reorganized online.

L
The indirect list data set (ILDS). The online reorganization process does not make a copy of this data set.

X
The primary index of a PHIDAM database. This data set is the index for the A-through-J data sets and is replaced by the Y data set when the online reorganization process copies the database records from the A-through-J and X data sets into the M-through-V and Y data sets. The use of the X character applies generally to HALDB partitions, regardless of whether they are capable of being reorganized online.

Y
The primary index of a PHIDAM database. This data set is the index for the M-through-V data sets and it is replaced by the X data set when the online reorganization process copies the database records from the M-through-V and Y data sets into the A-through-J and X data sets. The use of the Y character applies only to HALDB partitions that are capable of being reorganized online.

ppppp
Specifies the five-digit partition ID that is assigned by IMS.

The data set names for the output data sets are identical to the names of the corresponding input data sets, except for the IMS-assigned data set name type character (A-through-J, M-through-V, X, or Y). The following table shows example data set names.

Table 79. Data set name examples for HALDB Online Reorganization

Active data set before online reorganization	Data set group or index	Partition id	Input data set name	Output data set name
A-through-J (and X)	1	00003	DH41.A00003	DH41.M00003
A-through-J (and X)	Index	00065	ACCT.X00065	ACCT.Y00065
M-through-V (and Y)	2	00005	PAY.MST.N00005	PAY.MST.B00005
M-through-V (and Y)	8	00001	PAY.EMP.T00001	PAY.EMP.H00001

Related concepts:

"Naming conventions for HALDB partitions, ddnames, and data sets" on page 26

"HALDB online reorganization" on page 669

Output data set requirements for HALDB Online Reorganization

During the initialization phase for an online reorganization of a HALDB partition, IMS creates any output data sets that do not already exist.

For example, when the input data sets are the A-through-J (and X) set, if the M and P output data sets already exist, but the N and O output data sets do not, IMS creates the N and O data sets and uses the existing M and P data sets.

Any existing output data sets must conform to the HALDB requirements for existing output data sets. Any data in the existing output data sets is overwritten during the copying phase of an online reorganization.

Requirements for existing output data sets:

If an existing output data set does not meet the requirements described in this section, IMS displays an error message and the online reorganization for the HALDB partition does not begin.

An OSAM output data set has the following requirements:
- Must be cataloged
- Must be a DASD data set
- Must not be a VSAM data set, except for the primary index data set of a PHIDAM database
- Must not be a PDS, PDSE, or a member of a PDS or PDSE

A VSAM output data set has the following requirements:
- Must be a VSAM entry-sequenced data set (ESDS), except for the primary index data set of a PHIDAM database
- Must have the REUSE attribute
- Must have a fixed-length record length that is identical to that of the corresponding input data set
- Must have a control interval size that is identical to that of the corresponding input data set
- Must have a SHAREOPTIONS attribute value that is at least as high as that of the corresponding input data set if the database is defined to DBRC with a SHARELVL attribute value of 2 or 3

For both OSAM and VSAM data sets, if you are increasing the block or CI size when you are altering the structure of the database by using the INIT OLREORG OPTION(ALTER) command, you cannot use existing data sets from a previous

online reorganization. If you manually allocate the output data sets when the ALTER option is specified, you must ensure that the new data sets can accommodate any increase to the block or CI size that you have specified in the RECON data set for the alter operation.

A primary index data set has the following requirements:
- Must be a VSAM key-sequenced data set (KSDS)
- Must have the same key offset and length as the corresponding input KSDS
- Must have the other required characteristics listed for VSAM output data sets

Related concepts:

"The initialization phase for HALDB Online Reorganization" on page 670

Attributes of automatically created output data sets:

For those output data sets that do not already exist at the beginning of the initialization phase of an online reorganization, IMS creates the data sets.

IMS creates the data sets with the following attributes:

Number of Volumes
> If a particular input data set is SMS-managed, IMS creates the corresponding output data set with the same number of volumes.
>
> If the input data set is not SMS-managed, IMS automatically creates the corresponding output data set only when the input data set resides on a single volume. For a non-SMS-managed input data set that resides on multiple volumes, you must create the corresponding output data set before starting the online reorganization.

Location of SMS-managed output data sets
> If a particular input data set is SMS-managed, the corresponding output data set is also SMS-managed, and uses the same storage class as the input data set.
>
> Your site's storage administrator must ensure that this storage class refers to a storage group with sufficient space to hold the output data set, or that the automatic class selection (ACS) routine selects an appropriate storage class for the data set.

Location of non-SMS-managed, non-VSAM output data sets
> Regardless of the type of DASD on which the input data set resides, IMS creates the corresponding non-VSAM output data set using the equivalent of a DD statement `UNIT=SYSALLDA` parameter.
>
> When it creates the output data set, IMS does not request any specific volume serial number, thus allowing the data set to be created on a storage volume or, if no storage volume is available, on a public volume.

Location of non-SMS-managed, VSAM output data sets
> IMS creates a VSAM output data set on the same volume as the corresponding input data set. This restriction can limit the usefulness of automatically creating a VSAM data set that is not SMS-managed.

Format of the output data sets (DSNTYPE)
> IMS uses the format type of the input data sets for the output data sets. If the input data sets are defined as OSAM sequential large format data sets (DSNTYPE=LARGE), IMS automatically defines the output data sets as large format data sets.

Size of output data sets on a single volume

When the input data set has extents on only one DASD volume, IMS creates the output data set on a single volume using the equivalent of a DD statement `VOLUME=(,,,1)` parameter.

The amount of primary space for the output data set is derived from the space allocation of the input data set:

- For a non-VSAM data set, the primary space is the total amount of space in the first five extents on the volume.
- For a VSAM data set, the primary space is the primary space allocation used when the input data set was created.

If you specified secondary space amount for the input data set, IMS uses this same secondary amount for the output data set.

To reserve approximately the same amount of space for the output data set as was reserved for the input data set, regardless of the DASD types involved, IMS requests the space for the output data set as a number of OSAM blocks or VSAM records. For input data sets that did not specify a number of OSAM blocks or VSAM records, IMS converts the cylinder or track allocation to an equivalent number of blocks or records.

An automatically created output data set could have a considerably different amount of available DASD space than was used for the input data set. For example, for an input data set that used secondary allocation, the automatic creation process reserves the primary space for the output data set, but there might not be enough space on the volume for secondary allocation either during the online reorganization or during later database processing.

Size of output data sets on multiple volumes (SMS-managed only)

IMS automatically creates multiple-volume output data sets only when the input data set (and, therefore, the output data set) is SMS-managed. You can determine the storage class by examining the input data set or the site's ACS routine.

Although it is not strictly a requirement for SMS-managed multiple-volume output data sets, you should ensure that the storage class specifies the guaranteed-space attribute. By specifying the guaranteed-space attribute, you allow VSAM to use the primary-space allocation for *each* of the volumes when it creates the output data sets. Secondary space is used as needed. However, even with the guaranteed-space attribute, the output data sets might not have the same amount of space as the input data sets, especially if secondary-space allocation was used for the input data sets.

The requested primary and secondary space is based on the input data set's space allocation on the first DASD volume.

Block or control interval (CI) sizes of output data sets

If you are not altering the structure of a database, each output data set that IMS creates for an online reorganization has the same block or control interval size as its corresponding input data set.

If you are altering the structure of a database by using the ALTER option of the INIT OLREORG command and you specified ALTERSZE values for one or more data set groups in the RECON data set, IMS creates the output data sets for which you specified ALTERSZE values with block or CI sizes that match the ALTERSZE values.

When the ALTER option is used, the online reorganization applies to the entire HALDB database. For each partition that contains block or CI sizes that must change, the ALTERSZE values must be set separately before you issue the INIT OLREORG command.

To set ALTERSZE values prior to a reorganization that alters the structure of a database, you can use either the DBRC command CHANGE.PART.

For VSAM data sets, if you specify an ALTERSZE value and the output data set already exists with a CI size that does not match the ALTERSZE value, the reorganization fails for the partition. If this happens, you must correct the discrepancy between the ALTERSZE value and the CI size and, after alter processing is complete for all other partitions in the database, reissue the INIT OLREORG command with OPTION(ALTER) specified. One way you can correct a discrepancy between an ALTERSZE value and a CI size is to delete the existing VSAM data set so that the alter function can automatically create the data set with the new CI size.

For OSAM data sets, the alter function can change the block sizes even if the output data sets already exist.

Starting HALDB Online Reorganization

You start a HALDB Online Reorganization by using either version of the INITIATE OLREORG command.

Before HALDB partitions can be reorganized online, you must enable the HALDB master database for online reorganizations by issuing either of the DBRC commands INIT.DB OLRCAP or CHANGE.DB OLRCAP.

The following table describes the tasks and commands for starting or resuming an online reorganization for a HALDB partition.

Table 80. Mapping startup tasks to commands for HALDB Online Reorganization

Task	Type-1 command format	Type-2 command format
Begin HALDB Online Reorganization for one or more partitions.	/INITIATE OLREORG	INITIATE OLREORG
Begin HALDB Online Reorganization to alter the structure of a HALDB database across all partitions.	Not supported	INITIATE OLREORG OPTION(ALTER)
Resume HALDB Online Reorganization for one or more partitions.	/INITIATE OLREORG	INITIATE OLREORG
Set the RATE for a HALDB Online Reorganization.	/INITIATE OLREORG SET(RATE(*rate*)) or /UPDATE OLREORG SET(RATE(*rate*))	INITIATE OLREORG SET(RATE(*rate*)) or UPDATE OLREORG SET(RATE(*rate*))

Related reference:

➡ INITIATE OLREORG command (Commands)

Monitoring HALDB Online Reorganization

Several commands allow you to monitor a HALDB Online Reorganization.

The following table describes the tasks and commands for monitoring an online reorganization for a HALDB partition.

Table 81. Mapping monitoring tasks to commands for HALDB Online Reorganization

Task	Command
Display status and rate information about HALDB Online Reorganizations that are in progress.	`QUERY OLREORG` type-2 command
Display status and rate information about HALDB Online Reorganizations when the ALTER option is specified.	QUERY OLREORG STATUS(ALTER) type-2 command
Monitor and display the status of the specified databases or partitions (including those HALDB Online Reorganizations that are in progress).	`/DISPLAY DB OLR` type-1 command
Display HALDB Online Reorganization status.	`QUERY DB` type-2 command
List all of the databases for which HALDB Online Reorganizations are in progress.	`QUERY DB STATUS(OLR)` type-2 command

Modifying and tuning HALDB Online Reorganization

Modifying and tuning a HALDB Online Reorganization is performed primarily by modifying the rate at which the reorganization is performed.

The following table describes the tasks and commands for modifying and tuning an online reorganization for a HALDB partition.

Table 82. Mapping modifying and tuning tasks to commands for HALDB Online Reorganization

Task	Type-1 command	Type-2 command
Change the impact of HALDB Online Reorganization on overall system performance, for one or more partitions.	/UPDATE OLREORG SET(RATE(*rate*))	UPDATE OLREORG SET(RATE(*rate*))
Specify whether to delete the inactive data sets after the copying phase completes.	/UPDATE OLREORG OPTION(DEL | NODEL)	UPDATE OLREORG OPTION(DEL | NODEL)

If the database is shared, only the type-2 command can be routed to all IMS systems that share the database. The type-1 command applies only to the system in which it is issued.

If the reorganization was started by an INITIATE OLREORG command with OPTION(ALTER) specified, all of the partitions in the database are reorganized. In this case, if you need to modify or tune the online reorganization of all partitions in the database, specify an asterisk in the NAME keyword, for example UPDATE OLREORG NAME(*) SET(RATE(*rate*)).

Related reference:

➡️ UPDATE commands (Commands)

Stopping HALDB Online Reorganization

You can stop a HALDB online reorganization by issuing the TERMINATE OLREORG command against one or more partitions.

The command can be issued as either an IMS type-1 command, /TERMINATE OLREORG, or a type-2 command, TERMINATE OLREORG.

If the database is shared, only the type-2 command can be routed to all IMS systems that share the database. The type-1 command applies only to the system in which it is issued.

If the reorganization was started by an INITIATE OLREORG command with OPTION(ALTER) specified, all of the partitions in the database are reorganized. In this case, if you need to stop the reorganization across the entire database, specify an asterisk in the NAME keyword, for example TERMINATE OLREORG NAME(*).

Example: The following figure shows the processing steps for an online reorganization of a HALDB partition and how it is affected by a TERMINATE OLREORG command that temporarily stops the reorganization:

- When you issue the TERMINATE OLREORG command, IMS terminates the reorganization by entering the termination phase.
- Later, when you issue the INITIATE OLREORG command, IMS restarts the reorganization from the initialization phase, then proceeds to the copying phase. In the figure, the reorganization then completes successfully through the termination phase.

Note that there are two sets of data sets for the second initialization phase because the reorganization is not complete.

In the figure, the columns represent the flow of control through the phases of the online reorganization, from the user to IMS, and the status of the data sets as the processing events occur.

User	IMS	Status of data sets

Initialization phase

Invokes INITIATE OLREORG command → Gets ownership from DBRC

Performs initial steps (create data sets, etc.)

Tells DBRC about cursor-active status → Now two active sets of data sets exists OLREORG CURSOR ACTIVE=YES in RECON listing

Copying phase

Copies data from input to output data sets → Database records are locked, copied and then unlocked by UOR

Termination phase

Invokes TERMINATE OLREORG command → Tells DBRC to relinquish OLREORG ownership → Still two active sets of data sets exist but no copying is taking place

Elapsed time

Initialization phase

Invokes INITIATE OLREORG command → Gets ownership from DBRC → Still two active sets of data sets

Copying phase

Resumes copying data from input to output data sets (from where it stopped) → Database records are locked, copied and then unlocked by UOR

Termination phase

When copying is complete, tells DBRC to end cursor-active status → Now only one active set of data sets exists OLREORG CURSOR ACTIVE=NO in RECON listing

Possibly deletes inactive set of data sets

Relinquishes ownership through DBRC

Figure 265. Processing steps for an interrupted online reorganization of a HALDB partition

Related tasks:

"Stopping alter processing before completion" on page 771

Related reference:

➥ TERMINATE commands (Commands)

How a HALDB Online Reorganization impacts IMS logging

The online reorganization of a HALDB partition generates X'50' database change log records for all of the data in the partition.

IMS also logs other HALDB Online Reorganization information in a small number of X'29' log records. Thus, the total amount of log data generated by a HALDB Online Reorganization is considerably greater than the amount of data in the partition.

Reorganizing partitions online, especially reorganizing multiple partitions in parallel on the same IMS system, can generate sufficient log data to impact normal transaction processing. The large number of log records that are generated can also affect the rate of OLDS switches and log archiving.

Controlling the overall system impact of a HALDB Online Reorganization

An online reorganization of a HALDB partition can impact the overall system performance of an IMS system, and likewise, other IMS work can affect the performance of an online reorganization.

Your options for controlling the impact of an online reorganization on your IMS system include:
- Specifying a separate buffer subpool for the data set being reorganized by using the DBD statement in the DFSVSMxx member in the IMS.PROCLIB data set.
- Adjusting the rate at which the online reorganization runs by using the RATE parameter of the INITIATE OLREORG and UPDATE OLREORG commands.

Specifying a separate buffer subpool for the data set being reorganized can reduce buffer contention between the online reorganization function and other processes that also require buffer resources.

Adjusting the value of the RATE parameter can also help minimize the impact of an online reorganization on the IMS system by introducing an intentional, periodic delay in the online reorganization process, which allows other IMS work to proceed.

You specify the RATE value as a percentage, with values less than 100 representing the addition of the intentionally introduced delay.

The default value for the RATE parameter is 100, which allows the online reorganization to run as fast as possible, depending on system resources, system contention, and log contention, with no intentionally introduced delay. However, if you set the RATE value to 25, for example, IMS adds a delay to the reorganization processing so that 25% of the total processing time for a unit of reorganization is spent copying the data, and the remaining 75% is spent in an intentionally introduced delay. Thus, RATE(25) would cause the online reorganization to take approximately four times as long to run as it would have run with RATE(100).

You can change the RATE value at any time by issuing the UPDATE OLREORG command.

IMS restart and XRF processing for HALDB Online Reorganization

If you shut down IMS while any online reorganizations of HALDB partitions are running, IMS suspends the reorganizations before completing the shutdown checkpoint.

After IMS restarts, IMS automatically resumes the online reorganizations. IMS resumes the online reorganization even if you specified the NOPDBO option in the DFSVSMxx member.

If IMS terminates abnormally while any online reorganizations are running, IMS dynamically backs out all uncommitted changes for these reorganizations to the most recent sync point. After IMS restarts, IMS automatically resumes the online reorganizations.

When an XRF takeover occurs, IMS automatically resumes the online reorganizations on the new active IMS system.

You can also use the OPTION(REL) parameter of either the INITIATE OLREORG command or the UPDATE OLREORG command, or code RELOLROWNER=Y in the DFSDFxxx member of the IMS.PROCLIB data set, to instruct the IMS system to release ownership if the system terminates (either normally or abnormally) before completing the reorganization. Using one of these options allows another IMS system to resume the suspended reorganization.

Related reference:

➡ INITIATE OLREORG command (Commands)

IMS restart and Fast Database Recovery processing for HALDB Online Reorganization

When an FDBR takeover occurs, IMS backs out all uncommitted changes for these reorganizations to the most recent sync point, closes the partition, and unauthorizes the partition.

After the FDBR terminates, the failed IMS system no longer has ownership of the online reorganization. The online reorganization can be resumed by issuing the command INITIATE OLREORG on either another IMS system or on the failed IMS system after it is restarted.

Locking impacts of HALDB Online Reorganization

To maintain partition integrity and recoverability during an online reorganization of a HALDB partition, HALDB Online Reorganization requests a very large number of locks for each UOR from the active lock manager, either the program isolation lock manager or the Internal Resource Lock Manager (IRLM) component.

When IRLM is the lock manager, an online reorganization can incur a significant number of IRLM lock structure accesses per second, especially if the reorganization is running with the RATE(100) specification. Also, if the database is shared by multiple IMS systems when IRLM is used, the online reorganization function requests global locks, just as application programs do.

Recommendations:

- Use the IBM System z Coupling Facility Structure Sizer Tool (CFSizer) to model the additional coupling facility activities to ensure that your coupling facility configuration is capable of handling the extra load introduced by the online reorganizations:
  - For IRLM 2.1 with `PC=NO` specified, each additional 1000 concurrently held locks requires 256 KB of ECSA storage.
  - For IRLM 2.2, each additional 1000 concurrently held locks requires 540 KB obtained from IRLM private storage. No increase in ECSA storage is necessary.
- Review your LOGL latch contention rate, OLDS logging rate, IRLM lock structure access, and DBBP (for OSAM) latch contention.

CFSizer is available for you to use at the following website: www.ibm.com/
servers/eserver/zseries/cfsizer/, or search for "CFSizer" at the IBM website:
www.ibm.com.

Using IMS utilities with HALDB Online Reorganization

The IMS utilities Batch Backout (DFSBBO00), Database Change Accumulation
(DFSUCUM0), Database Image Copy (DFSUDMP0), Database Recovery
(DFSURDB0), and Primary Index and ILDS Rebuild (DFSPREC0) have special
considerations when used with HALDB Online Reorganization.

Batch Backout (DFSBBO00)
> If dynamic backout fails for a unit or reorganization, IMS creates a backout
> record in the RECON data sets that contains the dynamic PSB name. Run
> the Batch Backout utility (DFSBBO00) for the listed PSB name.
>
> If dynamic backout was not attempted, use the Log Recovery utility
> (DFSULTR0) with the PSB option to list those PSBs that require backout. If
> there is in-flight online reorganization work for a HALDB partition that
> requires backout, the Log Recovery utility lists the dynamic PSB name.
> Run the Batch Backout utility (DFSBBO00) for each of the listed PSB
> names.

Database Change Accumulation (DFSUCUM0)
> You can use the Database Change Accumulation utility (DFSUCUM0) to
> accumulate changes for HALDB partition A-through-J data sets and for the
> M-through-V data sets. You need to specify the DB0 control statement so
> that the Database Change Accumulation utility accumulates changes or
> purges changes from before the online reorganization started.
>
> The Database Change Accumulation utility might create Database Change
> Accumulation header records (type X'25' records) for a corresponding
> A-through-J and M-through-V data set if the online reorganization
> checkpoint is not complete at the start of the database change
> accumulation.

Database Image Copy (DFSUDMP0)
> You can use the Database Image Copy utility (DFSUDMP0) to copy the
> currently active data set that is recorded in the RECON data sets. The
> Database Image Copy utility also determines if it should copy the
> M-through-V data sets or the A-through-J data sets. However, if a partition
> is in the cursor-active status, you cannot run the Database Image Copy
> utility for that partition.
>
> It is not necessary to code a DD statement in the JCL when copying
> HALDB partition data sets, because they are dynamically allocated.

Database Recovery (DFSURDB0)
> The Database Recovery utility (DFSURDB0) expects the utility output data
> sets to exist, and makes no attempt to create them.

Primary Index and ILDS Rebuild (DFSPREC0)
> You can use the HALDB Index/ILDS Rebuild utility (DFSPREC0) to
> recover the primary index data set for both the input and output data sets.
> You can also use the HALDB Index/ILDS Rebuild utility to recover both
> the X and Y primary index data sets and the ILDS in a single run. You
> cannot specify a particular input or output data set, but if the partition is
> in the cursor-active status, the utility allocates and rebuilds all of the
> necessary data sets.

Related tasks:

"Recovery for HALDB Online Reorganization" on page 688

Related reference:

⮕ Database Image Copy 2 utility (DFSUDMT0) (Database Utilities)

⮕ Recovery utilities (Database Utilities)

Recovery for HALDB Online Reorganization

After DBRC sets the cursor-active status for the partition in the RECON data sets, and until the copying phase completes and DBRC resets the cursor-active status, you can recover any of the input or output data sets using the Database Recovery utility (DFSURDB0).

To restore the output data sets, the Database Recovery utility uses the database change records (type X'50' log records) and applies them to empty output data sets.

Recommendation: Make an image copy of the output data sets as soon as possible after the online reorganization completes. Recovering from this image copy is faster than recovering from the database change records that are logged during the online reorganization. However, you cannot make an image copy while the partition is in cursor-active status.

To recover an output data set before the online reorganization completes, perform the following tasks:

1. Stop the online reorganization by using the `TERMINATE OLREORG` command. If the online reorganization encountered an abend, it is stopped automatically.
2. Issue the `/DBR` or the `UPDATE DB` command for the HALDB partition.
3. Run database change accumulation, as necessary. You can create the JCL by issuing the `GENJCL.CA` command, or you can run the Database Change Accumulation utility (DFSUCUM0) from your own JCL. The purge time for the change accumulation must be equal to the time of the beginning of the online reorganization to represent restoring from the initial empty state of the data set.
4. Create the output data set to be recovered, either by using a JCL DD statement or by using Access Method Services, as appropriate.
5. Recover the database changes. You can create the JCL by issuing the `GENJCL.RECOV` command. Alternatively, you can run the Database Recovery utility (DFSURDB0) from your own JCL with the DD statement for DFSUDUMP specified as DUMMY to indicate that there is no image copy from which to restore.
6. Run the Batch Backout utility (DFSBBO00), because you might need to back out uncommitted data.
7. After you have recovered, and possibly backed-out, all of the required data sets of the HALDB partition, issue the `/STA DB` or the `UPDATE DB` command for the HALDB partition.
8. Issue the `INITIATE OLREORG` command to resume the online reorganization.

You can also recover an output data set after the online reorganization completes but before an image copy has been made. Follow the same steps as for recovering an output data set before the online reorganization completes, except the steps for stopping and restarting the online reorganization.

In addition, you can recover an output data set from a point other than the beginning of the online reorganization, such as from a full dump of a DASD volume, using existing procedures if the online reorganization is either completed or terminated.

Specifying a purge time for the database change accumulation utility

When you run the Database Change Accumulation utility (DFSUCUM0) for one of the output data sets, specify a purge time that is equal to the online reorganization start time.

Specifying this purge time is necessary if change accumulation records (or an input log) that involve the output data set span the time that a online reorganization was started. Specifying the purge time eliminates database change records from before this point in time and is analogous to eliminating database change records from prior to the start time of an image copy.

Specifying a starting point for the GENJCL.CA and GENJCL.RECOV commands

Even if no image copy exists for the output data sets, the RECON data sets reflect the beginning of the online reorganization as a starting point from which you can perform forward recovery of one of these data sets, even after the online reorganization is complete. Until you make an image copy of an output data set, the `GENJCL.CA` command treats this starting point as though it were the most recent image copy and causes changes to the output data set to be accumulated from that point. Similarly, the `GENJCL.RECOV` command prepares recovery of an output data set from this point, even if no physical image copy exists.

Specifying the active data sets for the database image copy utilities

The database image copy utilities always copy from the currently active data sets that are recorded in the RECON data sets. Regardless of whether the A-through-J or the M-through-V data sets are active, you do not need to change the JCL or control statements for these utilities to specify which set of data sets to use.

On the utility control statement for the Database Image Copy utility (DFSUDMP0), the DDNAME does not need to refer to the currently active data set. Regardless of whether the A-through-J or the M-through-V data sets are active, the utility automatically uses currently active data sets.

Example: Assume that the data set for a second data set group defined in the DBD is to be copied, and that the partition name is PARTNO3. Regardless of which set of data sets is active, you can code a DDNAME of either PARTNO3B or PARTNO3N on the control statement. If the A-through-J data sets are active, whether you specify PARTNO3B or PARTNO3N, the utility copies from PARTNO3B. Likewise, if the M-through-V data sets are active, the utility copies from PARTNO3N.

In the JCL statements for the Database Image Copy utility, you should omit the DD statement that refers to the input data set. Based on whether the A-through-J or the M-through-V data sets are active, the utility dynamically allocates the appropriate data set. A DD statement that refers to a specific data set name can cause the utility job to fail because of a "`Data Set Not Found`" condition during job-step initiation. This condition occurs if an inactive data set name is coded in the JCL and the data set does not exist.

Rebuilding the primary indexes and the ILDS while in the cursor-active state

If the integrated HALDB Online Reorganization function is stopped before completion and the partition remains in the cursor-active state, you can still use the HALDB Index/ILDS Rebuild utility (DFSPREC0) to rebuild both the X and the Y primary index data sets and the ILDS.

When the primary index must be rebuilt, the DFSPREC0 utility rebuilds both the X and the Y data sets because the records in the partition are split between the input and output sets of data sets.

Related concepts:

"Using IMS utilities with HALDB Online Reorganization" on page 687

Offline Reorganizations after HALDB Online Reorganizations

HALDB Online Reorganization does not prevent you from performing offline reorganizations.

If an online reorganization is currently in progress, you can perform an offline reorganization after stopping the online reorganization by issuing the TERMINATE OLREORG command and then stopping access to the partition by issuing either the /DBRECOVERY command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command. The HD Reorganization Unload utility (DFSURGU0) automatically unloads the records from the active portions of both the A–J and M–V data sets. The HD Reorganization Reload utility (DFSURGL0) loads all records into the A–J data sets and turns off the cursor-active status of the online reorganization in the RECON data set.

If an online reorganization is not currently in progress, you can perform an offline reorganization of a HALDB database without taking any additional special steps.

Related concepts:

"HALDB alter and offline reorganization" on page 766

Related tasks:

"HALDB offline reorganization" on page 664

Activating sequential buffering to improve the performance of HALDB Online Reorganization

You can use sequential buffering to improve the performance of online reorganization of OSAM databases by including the SBONLINE statement in the IMS.PROCLIB data set member DFSVSMxx.

Using the SBONLINE statement causes IMS to load the sequential buffering modules during initialization so that, whenever you start an online reorganization for an OSAM partition, IMS activates sequential buffering immediately. If you do not include the SBONLINE statement, IMS analyzes the DL/I calls to determine whether sequential buffering is suited for processing the reorganization.

The two forms of the SBONLINE control statement are:

```
SBONLINE
SBONLINE,MAXSB=nnnnn
```

where *nnnnn* is the maximum amount of storage (in kilobytes) that can be allocated to sequential buffers.

When the maximum amount of storage is reached, IMS stops allocating sequential buffers to online applications (including HALDB Online Reorganization) until these

applications release sequential buffer space. If you do not specify the MAXSB= keyword, the maximum amount of storage for sequential buffers is unlimited.

Related concepts:

➡ Specifications for OSAM sequential buffering (System Definition)

Related reference:

➡ Specifying sequential buffering for an online system (System Definition)

The HALDB self-healing pointer process

Reorganizations of HALDB databases with logical relationships and secondary indexes do not require the execution of utilities to update pointers. Instead, HALDB uses a self-healing pointer process to correct logical relationship and secondary index pointers.

This process is implemented by placing a target key and an extended pointer set (EPS) in the secondary index or logically related database and by using an indirect list data set (ILDS) in each partition of PHDAM and PHIDAM databases.

Related concepts:

"HALDB partition reorganization numbers" on page 183

"Considerations for HALDB partitioned secondary indexes" on page 374

"Logical relationships and HALDB databases" on page 329

"Logical relationship pointer types" on page 252

"Automatic update of HALDB secondary index and logical relationship pointers" on page 818

Related tasks:

"Allocating an ILDS" on page 546

How the self-healing pointer process works

The elements of the self-healing pointer process can be seen in the following figure, which shows the interrelationships between the elements prior to a database reorganization.
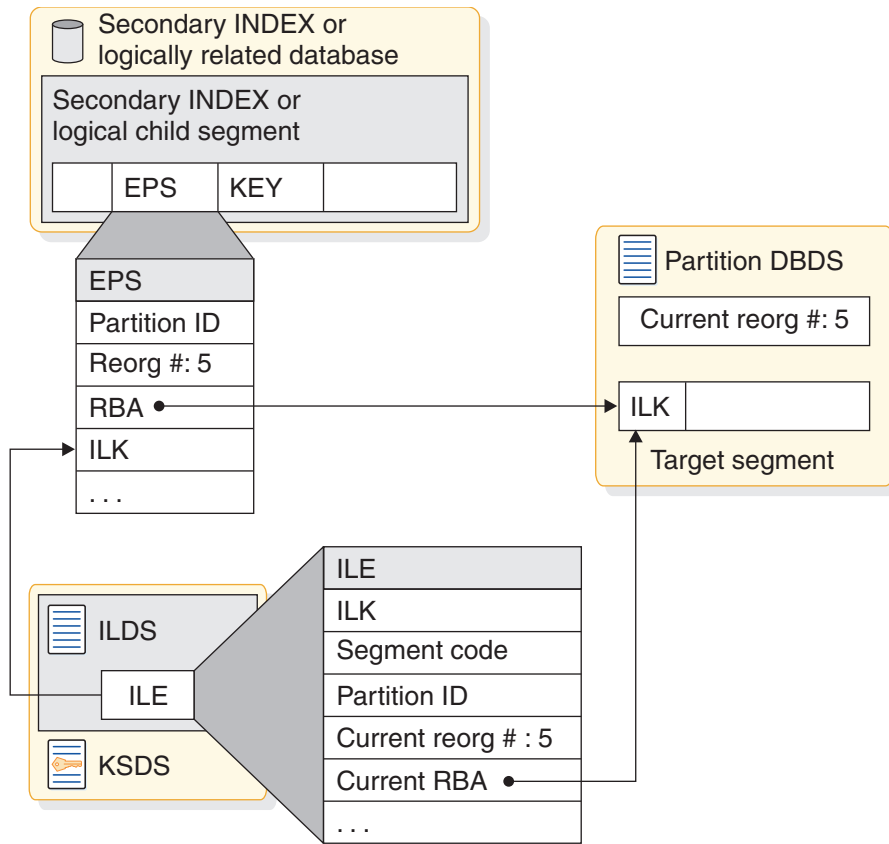
Figure 266. HALDB pointer before a reorganization

Each secondary index entry and each logical child segment contains the key of its target record. For secondary indexes, the key of the target's root segment is included in the prefix. For logical child segments, the concatenated key of the logical parent is included in the segment data.

Each segment in a PHDAM or PHIDAM database has an indirect list key (ILK). The ILK is unique for the segment type across the entire database. It is composed of the relative byte address (RBA), partition ID, and partition reorganization number of the segment when it was first created, as shown in the following figure. The ILK for a segment never changes. It is maintained across reorganizations.

ILK Prefix		
Initial RBA	Partition ID	Reorg. Number
Bytes 4	2	2

Figure 267. Format of an ILK

Each secondary index entry or logical child segment has an extended pointer set (EPS). The EPS includes the ILK of its target segment. It also contains the RBA, partition ID, and partition reorganization number for the target segment. These parts of the EPS might not be accurate. That is, they might not reflect the current

location of the target segment or the current reorganization number of the target segment's partition. In the preceding figure they are accurate.

The target segment has an indirect list entry (ILE) in the ILDS for a partition. The ILE contains accurate information about the target segment. This includes its current RBA, the correct partition ID, and the current reorganization number for the partition. The key of the ILE is composed of the ILK and the segment code of the target segment.

The reorganization number for a partition is physically stored in the partition's first database data set. This number is initialized by partition initialization or load, and incremented with each reorganization that reloads segments in the partition.

Finding target segments

When IMS accesses the target segment from the secondary index entry or logical child segment, it must first determine the partition in which the target resides.

IMS uses the key in the secondary index or logical child to determine the partition. Next it must determine the location in the target partition database data set. It compares the partition ID and reorganization number of the target partition with the partition ID and reorganization number stored in the EPS. If they match, IMS uses the RBA in the EPS to locate the target segment. If they do not match, the RBA in the EPS cannot be used.

When the RBA in the EPS cannot be used, IMS uses the information in the ILE to locate the target segment. The ILE key is found by using the ILK from the EPS and the target's segment code. The ILE is read from the ILDS of the partition determined from the target's key.

The following figure illustrates a situation in which the RBA in the EPS cannot be used. In the figure, the target partition has been reorganized three times since the EPS was accurate. This has moved the target segment and updated the reorganization number in the partition data set. The EPS still contains a reorganization number of 5, but the reorganization number in the partition data set is now 8. The information in the ILE has been updated by the HD Reorganization Reload utility. IMS uses the ILK from the EPS to find the ILE and uses the RBA in the ILE to find the target segment.
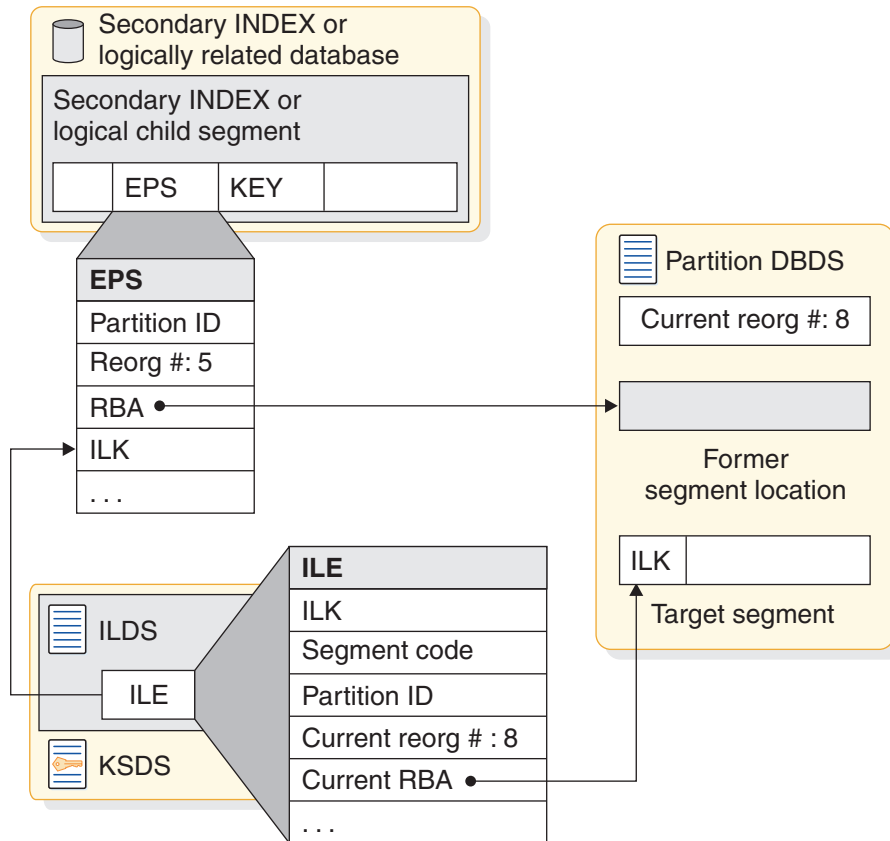
Figure 268. HALDB pointer after a reorganization

Even though the retrieval is indirect, often the CI containing the ILE will already be in an IMS buffer pool.

Recommendation: If possible, avoid the indirect process of locating target segments. Instead, get the target segment location from the EPS without reading the ILE. The self-healing process allows IMS to limit the use of ILEs.

Healing pointers

The self-healing process updates or corrects the information in extended pointer sets.

When the ILE is used, the information about the current location of the segment in the ILE is moved to the EPS. This allows IMS to avoid the indirect process if the EPS is used for a later retrieval. This correction to the EPS in the database buffer pool is always done.

Because of locking considerations, the update might not be written to the database on DASD. The buffer containing the entry or segment with the updated EPS is marked as altered if the application program is allowed to update the database. The call must be done with a PCB allowing updates, and the IMS system must have an access intent for the partition that allows updates. If updates are not allowed, the buffer is not marked as altered.

When the application reaches a sync point, it does not write buffers to DASD if they are not marked as altered. If the updated EPS is not written to DASD, the next time it is retrieved from DASD and used to find its target, IMS must use the indirect process. That is, IMS must read the ILE again.

The following figure shows the EPS after it has been healed. The RBA points to the current location. The partition ID is correct. The partition reorganization number matches the number stored in the partition database data set.
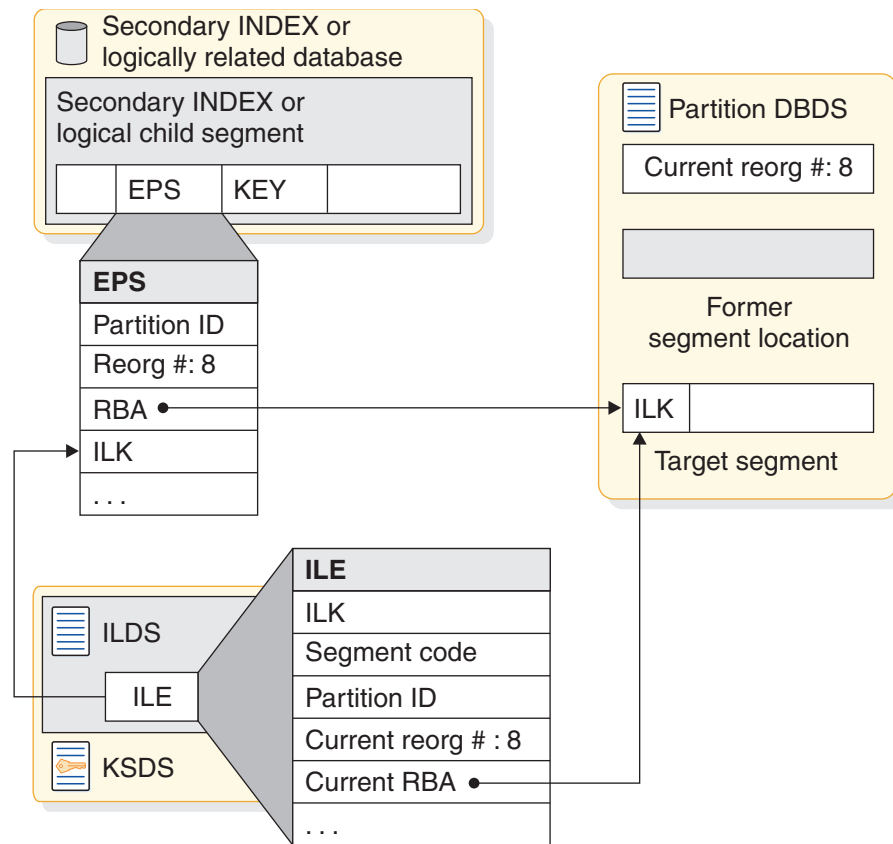
Figure 269. HALDB pointer after the self-healing process

Performance of the self-healing process
The performance of the self-healing process can be much more efficient than you might anticipate.

Many pointers can be healed with a small number of ILDS reads. This is due to the use of IMS database buffering. ILDSs are database data sets. They use database buffer pools in the same way that other database data sets use them. If a CI is already in its buffer pool, it does not have to be read from DASD.

Each ILE is 50 bytes. You specify the CI sizes for your ILDSs. An 8 KB ILDS CI holds up to 163 ILEs and a 16 KB CI holds up to 327 ILEs, so a single CI can hold many ILEs. After a reorganization, IMS might need to heal many pointers to the reorganized partitions.

When there are frequent uses of the CIs in an ILDS, they tend to remain in their buffer pool. One read of an ILDS CI might be sufficient to heal hundreds of pointers. As with most IMS database tuning, having a large number of buffers for frequently used data sets can be highly beneficial.

Another benefit of the self-healing process is that it does not waste resources healing pointers that are not used. In many secondary indexes, only a small number of entries are actually used. With a non-HALDB database, the entire index

is rebuilt every time the indexed database is reorganized. With HALDB, the index is not rebuilt and only a small number of referenced index entries are updated. HALDB does not use resources to update pointers that are never used.

When an EPS is updated, an application marks buffers as altered only if the application is allowed to make updates. If updates are allowed and a block-level data sharing environment is being used, a block lock is requested for the altered block. Block level data sharing environments exist when the IRLM is used and the share level for the database is either 2 or 3. The block locks are held until the application program commits its unit of work, which could cause a performance problem.

Optimizing self-healing performance:

Usually application programs with update authority commit frequently. This is good programming practice.

Occasionally, an application program that is allowed to do updates does not actually do them. For example, a program with a PCB specifying PROCOPT=A might only read. In this case, it might not commit frequently. Because it only reads, it never holds many locks. This could change with the implementation of HALDB. If the program runs in a block level data sharing environment and invokes the healing process, it will hold block locks until they are committed. This could cause two problems. First, it might hold the locks for a long time and cause other programs to wait before they can update the blocks. Second, it could hold many locks. This could cause a storage shortage in the IRLM or a lock structure.

If you have a program that holds locks for a long time or that holds many locks when performing the self-healing pointer process, you have four options:
- If the application program does not make updates, use PROCOPT=G.
- Have your program commit frequently.
- Invoke the pointer healing process before you run application programs that use PROCOPT=A, but do not do any updates. Run another program or utility before this type of application program. The HALDB Conversion and Maintenance Aid tool supplies a pointer healing utility.
- Rebuild secondary indexes with an index builder, such as the IMS Index Builder for z/OS. The IMS Index Builder for z/OS creates EPSs with accurate RBAs.

This scenario is not common. Most users can let the pointer healing process occur without taking any special precautions.

Recommendation: Do not rebuild your secondary indexes after a reorganization. Let the self-healing process of HALDB correct the pointers. This shortens the outage for reorganizations and tends to minimize the use of resources.

Related Reading:
- For more information about the IMS High Availability Large Database Conversion and Maintenance Aid, see the *IMS High Availability Large Database Conversion and Maintenance Aid for z/OS User's Guide*.
- For more information about the IMS Index Builder, see the *IMS Index Builder for z/OS User's Guide*.

Changing the hierarchical structure of database records

There are two types of tuning changes you might need to make that involve changes to the structure of your database record.

The first is changing the hierarchical sequence of segment types in your database record to improve performance. The second is combining segments to maximize the use of space.

Related concepts:

"Modifying record segments" on page 721

Changing the sequence of segment types

In general, performance is best if frequently used dependent segments are close to the root segment and infrequently used dependent segments are toward the end of the database record.

This arrangement maximizes performance because all types of databases (except HSAM) have direct (therefore, fast) access to root segments. But, after the root is located, dependent segments are found by one of the following:

- Searching sequentially through the database record (HSAM and HISAM)
- Following pointers from the root segments to a dependent path and then searching through twin chains until the correct segment is reached (HDAM, HIDAM, PHDAM, and PHIDAM).

One way to determine whether the order of dependent segment types in your hierarchy is an efficient one is to examine the IWAITS/CALL field on the DL/I Call Summary report.

Related Reading: For detailed information about interpreting the DL/I Call Summary report, see *IMS Version 15 System Administration*.

The IWAITS/CALL field tells you, by DL/I call against a specific segment, the average number of times a segment had to wait for I/O operations to finish before the segment could be processed. A high number (and high, of course, is relative to the application) indicates that multiple I/O operations were required to process the segment.

If the database does not need to be reorganized, the high number can mean this is a frequently used segment type placed too far from the beginning of the database record. If you determine this is the situation, you can change placement of the segment type. The change can increase the value in the IWAITS/CALL field for other segments.

To change the placement of a segment type, you must write a program to unload segments from the database in the new hierarchical sequence. (The reorganization utilities cannot be used to make such a change.) Then you need to load the segments into a new database. Again, you must write a program to reload.

Combining segments

Combining segment types to maximize use of space is the second type of change you might need to make in the structure of your database record.

For example, having two segment types, a dependent segment for college classes with a dependent segment for instructors who teach the classes, is an inefficient

use of space if typically only one or two instructors teach a class. Rather than having a separate instructor segment, you can combine the two segment types, thereby saving space.

Combining segments also requires that you write an unload and reload program. (The reorganization utilities cannot be used to make such a change.)

Changing the hierarchical structure of a HALDB database

You can change the hierarchical structure of a HALDB database.

To change the hierarchical structure, you need to:
1. Determine whether the change you are making will affect the code in any application programs. If so, make sure the code gets changed.
2. Unload your database using your unload program and the existing DBD.
3. Code a new DBD.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
6. Reload your database using your load program and the new DBD. Remember to make an image copy of your database as soon as it is reloaded.
7. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Changing direct-access storage devices

Several situations might warrant tuning your database by changing DASDs (direct-access storage devices).

First, when application requirements change, you might require a faster or slower device. Second, you might want to take advantage of new devices offering better performance. Finally, you might need to change devices to get database data sets on two different devices, so as to minimize contention for device use.

You can change your database (or part of it) from one device to another using the reorganization utilities. To change direct-access storage devices:
1. Unload your database using the existing DBD and the appropriate unload utility.
2. Recalculate CI or block size to maximize use of track space on the new device.
3. Code a new DBD.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload your database, using the new DBD and the appropriate reload utility. Remember to make an image copy of your database as soon as it is reloaded.

6. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading the database to resolve prefix information.

Related concepts:

"Determining the size of CIs and blocks" on page 439

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Tuning OSAM sequential buffering

If you are using OSAM Sequential Buffering, you can do two things to help ensure that it processes your databases efficiently.

- Keep your databases well organized; that is, the logical (database record) sequence is nearly the same as the physical (DASD block) sequence.
- Select the right number of SB buffer sets.

Related concepts:

"Usage data for OSAM sequential buffering" on page 708

Example of a well-organized database

Keeping your databases well-organized is more important to the processing of OSAM sequential buffering (SB) than selecting the right number of SB buffer sets.

When the databases SB processes are well organized, you note elapsed time improvements. This is because your programs process IMS database segments and records, and they do not process DASD blocks directly. Processing a well-organized database in logical-record sequence results in an I/O reference pattern that accesses most DASD blocks in physical sequence. SB can take advantage of these sequential I/O patterns by issuing many sequential reads. Extensive use of sequential reads considerably reduces the elapsed time for your job.

Example of a badly organized database

Processing a badly-organized database in logical-record sequence typically results in an I/O reference pattern that accesses many DASD blocks in a random sequence.

This happens because many segments were stored in randomly scattered blocks after the database was loaded or reorganized. When your database is accessed in a predominantly random pattern, most I/O operations issued by the SB buffer handler are random reads. SB is not able to issue many sequential reads, and the elapsed time for your job is not considerably reduced.

You can use the SB buffering statistics in the optional //DFSSTAT reports to see if your database is well-organized. Your database is likely to be badly organized if a large percentage of the blocks were read with random reads during sequential processing. You can monitor this percentage over a period of time to see if it increases as the database ages.

Related concepts:

➡ //DFSSTAT reports (System Administration)

Ensuring a well-organized database

You can ensure your databases are reasonably well-organized by following a few common practices.

To ensure a well-organized database, take the following actions when designing your database:

- Provide enough embedded free space at database load or reorganization time. IMS can then use this free space to insert new segments near their related segments (segments in the same database record).

  Tip: Choose the amount of free space based on the growth and performance characteristics of your database. For new databases, use a value of 25% and increase or decrease this value as needed. It is a good idea to schedule a reorganization for the database when the reusable free space is less than 5%.

- Select an appropriate database reorganization frequency.
- Use efficient HDAM and PHDAM randomizing modules and randomizing parameters.

Related concepts:

"Reorganizing the database" on page 643

"Determining which randomizing module to use (HDAM and PHDAM only)" on page 433

Related tasks:

"Specifying free space (HDAM, PHDAM, HIDAM, and PHIDAM only)" on page 431

Adjusting HDAM and PHDAM options

You can choose from a number of different HDAM and PHDAM options, each of which have performance implications.

You can adjust HDAM and PHDAM options using the reorganization utilities:

1. Determine whether the change you are making will affect the code in any application programs. It should only do so if you are changing to a sequential randomizing module.
2. Unload your database, using the existing DBD and the appropriate unload utility.
3. Code a new DBD (for non-PHDAM) using the TSO Partition Definition Utility. If you changed your CI or block size, you need to allocate buffers for the new size.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Determine whether you need to recalculate database space.
6. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
7. Reload your database or partition using the new DBD (if any) and the appropriate reload utility. Make an image copy of your database as soon as it is reloaded.

Related concepts:

"Choosing HDAM or PHDAM options" on page 434

Chapter 22, "Designing full-function databases," on page 431

"Determining which randomizing module to use (HDAM and PHDAM only)" on page 433

Adjusting buffers

Adjusting buffers can affect performance. Adjust buffers when you see buffer performance begin to degrade, or if you want to add options to boost performance in anticipation of increased buffer activity.

This topic also discusses the performance implications of choosing a buffer size and number.

The size and number of buffers you can choose are described in "Multiple buffers in virtual storage" on page 440. To improve performance, read that topic and assess your current buffer settings before you adjust your buffers.

Overview of dynamic database buffer pools

You can dynamically reconfigure OSAM subpools and VSAM shared resource pools without taking your IMS system offline.

The dynamic database buffer pools function provides an alternative to specifying the OSAM and VSAM subpool definitions by changing the DFSVSMxx member of the IMS PROCLIB data set. DFSVSMxx is only processed during IMS initialization and the subpool definitions cannot be changed dynamically while IMS is online.

The dynamic database buffer pools function provides the ability to quiesce all activities against a subpool to make subpool definition changes dynamically by using the UPDATE POOL command. This command initiates the buffer pool reconfiguration that is defined in the DFSDFxxx member of the IMS PROCLIB data set while IMS resources are still actively in use.

In a single DFSDFxxx member, you can specify multiple unique OSAMxxx or VSAMxxx sections that contain different sets of definitions. When you invoke the UPDATE POOL command, specify the particular OSAMxxx or VSAMxxx section from which to take the updated definitions.

The dynamic changes are retained across an emergency restart because they are stored in the restart data set (RDS). However, the changes are lost with a subsequent cold or warm start. To make the changes permanent, it is necessary to make the changes in the DFSVSMxx member.

After you run the UPDATE POOL command, you can invoke the QUERY POOL command to return statistics for the current buffer pools and to confirm that the updated definitions are implemented as intended.

Types of supported buffer pool definition updates

You can make the following types of changes to your buffer pool definitions:
* Change the number of buffers for an existing OSAM subpool
* Create an OSAM subpool with a new subpool ID
* Add a new set of buffers to an existing OSAM subpool

- Delete an OSAM subpool
- Change the number of buffers for an existing VSAM subpool
- Create a subpool for an existing VSAM shared resource pool
- Delete a subpool from an existing VSAM shared resource pool
- Change the size of the buffers in a VSAM subpool
- Create a VSAM shared resource pool

Related concepts:

"Adjusting OSAM and VSAM database buffers" on page 705

"Monitoring VSAM buffers" on page 703

"VSAM buffers"

"Options for improving VSAM buffer performance" on page 703

"Defining a VSO DEDB area" on page 226

Related tasks:

"OSAM buffers" on page 704

"Adjusting OSAM database buffers dynamically" on page 706

"Adjusting VSAM database buffers dynamically" on page 707

Related reference:

➡ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

➡ UPDATE POOL command (Commands)

➡ QUERY POOL command (Commands)

VSAM buffers

IMS builds the VSAM buffer pool based on the number of buffers and the subpool sizes specified in the DFSVSMxx member of the IMS PROCLIB data set or, in batch or utility environments, in the DFSVSAMP data set.

You also can the use dynamic database buffer pools function to make changes to the number of buffers and the subpool sizes without bringing your IMS system offline.

Defining during system definition

When a DL/I call, such as an ISRT or GU, requires access to a VSAM data set, IMS makes VSAM PUT and GET calls as needed. The VSAM buffer pool is managed and manipulated by DFSMS, not by IMS.

Modifying dynamically

You can create new buffer pools and modify existing pools to be changed (or deleted) without having to bring your IMS system offline. While IMS resources are still actively in use, specify new VSAM shared resource pool definitions in the DFSDFxxx member of the IMS PROCLIB data set, and then issue the appropriate UPDATE POOL TYPE(DBAS) command.

Related concepts:

"Adjusting OSAM and VSAM database buffers" on page 705

Related reference:

➡ DFSVSMxx member of the IMS PROCLIB data set (System Definition)

Monitoring VSAM buffers

If you are using VSAM, you can monitor buffers using the DB-Monitor reports, the IMS Monitor tool, or an IMS command.

With DB-Monitor reports, a VSAM subpool report is produced for each buffer size you define. The VSAM Buffer Pool report tells the number of buffers in the subpool and their size (in the SUBPOOL BUFFER SIZE and TOTAL BUFFERS IN SUBPOOL fields).

The IMS Monitor is a tool that records data about the performance of your DL/I databases in a batch environment.

Use the type-1 /DISPLAY POOL DBAS command to display processor storage utilization statistics for VSAM database buffer pools.

If you dynamically reconfigure your VSAM shared resource pools using the dynamic database buffer pools function, you can issue the type-2 QUERY POOL TYPE(DBAS) command to obtain information about the buffer pools and to confirm that the dynamic updates are completed as intended.

Related concepts:

Chapter 29, "Monitoring databases," on page 635

➡ DB Monitor reports (System Administration)

Related reference:

➡ QUERY POOL command (Commands)

➡ /DISPLAY POOL command (Commands)

Options for improving VSAM buffer performance

You can make a number of adjustments to improve the performance of VSAM buffers.

- If background write is turned on and the number in the NUMBER OF VSAM WRITES TO MAKE SPACE IN THE POOL field is not zero, you probably do not have enough buffers allocated in the subpool. Try allocating more buffers to decrease the number or reduce it to zero.
- If you need to improve performance for a specific application, you can reserve subpools for certain data sets by:
  - Defining multiple local shared resource pools.
  - Dedicating subpools to a specific data set.
  - Defining separate subpools for index and data components of VSAM data sets.
- If VSAM sequential mode processing is not used, the number of VSAM buffers specified in the DFSVSAMP DD statement can dramatically affect performance. This problem occurs when the number of VSAM KSDS indexes that must be read, plus one for the data portion, is equal to or greater than the number of VSAM buffers allocated. This problem can be alleviated either by increasing the number of buffers or by using VSAM sequential mode. With VSAM sequential mode, the need to read indexes above the sequence set is reduced. However, sequential mode can only be obtained in a batch environment with a DBD referenced by a single PCB and with a processing option of LOAD or RETRIEVE only. VSAM sequential mode is not available in data sharing.
- VSAM buffers can take advantage of z/OS Hiperspace buffering.

Related concepts:

Hiperspace buffering parameters:

To use Hiperspace buffering, you must specify one or two optional parameters on the VSRBF subpool definition statement.

The parameters are:

HSO|HSR
Specifies the action IMS takes if Hiperspace buffering requested for a subpool is unavailable.

HSO   Hiperspace buffering is optional. IMS continues to run.

HSR   Hiperspace buffering is required. IMS terminates.

HSn
Specifies the number of Hiperspace buffers to build for a subpool. The number *n* is a 1- to 8-digit number.

Hiperspace parameters are valid only for buffer sizes of 4K or multiples of 4K. Specifying Hiperspace parameters on buffers smaller than 4K causes an error. To use Hiperspace buffering you might need to unload your database and then reload it into 4K or multiples of 4K CI sizes to accommodate Hiperspace requirements.

If you decide to leave intact databases with CI sizes of less than 4K, do not allocate any buffers less than 4K. The CIs that are less than 4K are placed in 4K or larger buffer pools. However, the CIs compete with VSAM data sets already there. This method might be expedient in the short term.

<u>Related Reading:</u> For more information about VSAM buffers, including Hiperspace buffers, see *z/OS DFSMS: Using Data Sets*.

Related concepts:
"Hiperspace buffering" on page 442
Related reference:

▸ Defining VSAM subpools (System Definition)

OSAM buffers

If you are using OSAM, individual subpool buffer reports do exist. However, you can monitor the number of buffers you are using by using the Enhanced OSAM Buffer Subpool statistics function. If you defined your OSAM subpools dynamically, you can also obtain information about the buffer pools by issuing a QUERY command.

The Enhanced OSAM Buffer Subpool statistics function supports the following values:

DBESF
Provides the full OSAM Subpool statistics in a formatted form.

DBESU
Provides the full OSAM Subpool statistics in an unformatted form.

DBESS
Provides a summary of the OSAM database buffer pool statistics in a formatted form.

DBESO
    Provides a the full OSAM database buffer pool statistics in a formatted form for online statistics returned as a result of a /DIS POOL command.

Another way to improve performance, this time for a specific application, is to reserve subpools for use by certain data sets. For example, if you have an index data set with a block size of 512 bytes, reserve a subpool for it that contains 512-byte buffers. You can do this by not defining 512-byte block sizes for any other data sets in the database. (Remember, block sizes are specified *by data set* in the BLOCK= operand in the DATASET statement in the DBD.) If you then allocate enough 512-byte buffers to hold all the blocks in your index, all blocks read into the buffer pool will remain in the buffer pool.

Performance can also be improved through the use of the **co** (caching option) parameter of the IOBF control statement specified either in the DFSVSMxxx member of IMS.PROCLIB or in DFSVSAMP.

You can dynamically reconfigure OSAM subpools, create new buffer pools, and modify existing pools without taking your IMS system offline.

Related concepts:

"Adjusting OSAM and VSAM database buffers"

OSAM buffer pool compatibility definition (System Definition)

Format of enhanced/extended OSAM buffer subpool statistics (Application Programming)

Database-Buffer-Pool report (System Administration)

Related reference:

QUERY POOL command (Commands)

Using the coupling facility for OSAM data caching (System Definition)

Defining OSAM subpools (System Definition)

Adjusting OSAM and VSAM database buffers

You can adjust OSAM and VSAM database buffers by specifying parameters in the DFSVSMxx member of the IMS PROCLIB data set during system definition.

You can also adjust OSAM and VSAM database buffers dynamically by specifying parameters in the DFSDFxxx member of the IMS PROCLIB data set.

Related concepts:

"Overview of dynamic database buffer pools" on page 701

"VSAM buffers" on page 702

IMS buffer pools (System Definition)

Related tasks:

"OSAM buffers" on page 704

Related reference:

DFSVSMxx member of the IMS PROCLIB data set (System Definition)

DFSDFxxx member of the IMS PROCLIB data set (System Definition)

UPDATE POOL command (Commands)

QUERY POOL command (Commands)

Adjusting OSAM and VSAM database buffers in DFSVSMxx

To adjust OSAM and VSAM database buffers, change the control statements that specify buffer size and number and place the control statements in the appropriate member of the IMS PROCLIB data set.

Put the new control statements in the:
- DFSVSAMP data set in batch and utility environments
- IMS.PROCLIB data set with the member name DFSVSMxx in IMS TM and DBCTL environments

Adjusting OSAM database buffers dynamically

To adjust OSAM database buffers while IMS resources are still actively in use, specify new OSAM subpool definitions in the DFSDFxxx member of the IMS PROCLIB data set, and then issue the appropriate UPDATE POOL TYPE(DBAS) command.

Prerequisites: Make sure the following conditions exist:
- The current number or size of OSAM buffer pools is either insufficient or not needed for workload processing for application programs.
- The DFSDFxxx member of the IMS PROCLIB data set exists.
- IMS is configured with at least a minimal Common Service Layer (to support issuing type-2 commands).

1. Update the DFSDFxxx member of the IMS PROCLIB data set with an IOBF statement. Complete one or more of the following steps:
   - Change the number of buffers for an existing OSAM subpool.

     When changing the number of buffers for an existing OSAM subpool, the values for *bufnum*, *fix1*, and *fix2* can be changed.
   - Create an OSAM subpool with a new subpool ID.

     Add an IOBF statement with a value for the *id* parameter that does not exist.

     A subpool can be created with a buffer size which already exists. It would be considered a duplicate subpool and would have to be defined with a unique subpool ID. A database data set must be closed and then re-opened in order for a different subpool to be assigned.

     If the buffer size is the same as an existing subpool and a unique subpool ID is not assigned, you are in effect changing the definitions for an existing subpool.
   - Add a new set of buffers to an existing subpool.

     Add an IOBF statement with the subpool ID that you want to add the new set of buffers to.

     The new values in the IOBF statement do not replace the values that were previously set.

     For example, specifying a different *bufsize* value does not change the size of the buffers in the subpool. It adds a new set of buffers with a different buffer size to the existing subpool. The effect of specifying a different *bufsize* value with an existing subpool ID is to increase the overall size of buffers in the subpool.
   - Delete an OSAM subpool by setting *bufnum* to 0.

     The format of the IOBF statement:

     ```
     IOBF=(bufsize,bufnum,fix1,fix2,id,co)
     ```
2. Issue the following command:

   ```
   UPD POOL TYPE(DBAS) SECTION(OSAMxxx) MEMBER(xxx)
   ```

If the procedure is performed successfully, the IMS system quiesces activities against the target subpools during reconfiguration.

If you specified a number of buffers that is unchanged from the number of buffers in the existing OSAM subpool, the request to update the OSAM subpool is ignored (as if the request was never made).

Determine whether the updated database buffer pools configuration is adequate for your workloads:

1. Issue the following command:

   ```
   QUERY POOL TYPE(DBAS) SUBTYPE(OSAM,VSAM) SIZE() POOLID() SHOW(STATISTICS)
   ```

   The IMS system returns statistics about the OSAM and VSAM buffer pools.
2. Based on the returned statistics, evaluate whether there must be further adjustments to the configuration of OSAM buffer pools.

Related concepts:

"Overview of dynamic database buffer pools" on page 701

Related reference:

➡ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

➡ UPDATE POOL command (Commands)

➡ QUERY POOL command (Commands)

Adjusting VSAM database buffers dynamically

To adjust VSAM database buffers while IMS resources are still actively in use, specify new VSAM shared resource pool definitions in the DFSDFxxx member of the IMS PROCLIB data set, and then issue the appropriate UPDATE POOL TYPE(DBAS) command.

Make sure that the following conditions exist:

- The current number or size of VSAM shared resource pools is either insufficient or not needed for workload processing for application programs.
- The DFSDFxxx member of the IMS PROCLIB data set exists.
- IMS is configured with at least a minimal Common Service Layer (to support issuing type-2 commands).

1. Update the DFSDFxxx member of the IMS PROCLIB data set. Do one or more of the following:

   - Change the number of buffers for an existing VSAM subpool.

     Define the DFSDFxxx member with a POOLID, which identifies the VSAM shared resource pool the subpool belongs to, followed by a VSRBF statement. For example:

     ```
     POOLID=(id,VSRBF=bufsize,bufnum)
     ```

     Set *bufnum* to a different value, indicating that the subpool with the specified buffer size is to change in the number of buffers.
   - Create a subpool for an existing VSAM shared resource pool.

     Update the DFSDFxxx member with a POOLID, which identifies the VSAM shared resource pool the subpool belongs to, followed by a VSRBF statement. For example:

     ```
     POOLID=(id,VSRBF=bufsize,bufnum,type,HSO,HSR,HSn)
     ```
   - Delete a subpool from an existing VSAM shared resource pool.

In the POOLID statement in the DFSDFxxx member, set *bufnum* to 0, indicating that the subpool with the specified buffer size is to be removed. For example:

```
POOLID=(id,VSRBF=bufsize,0)
```

- Change the size of the buffers in a VSAM subpool.

  Create a subpool with a different buffer size than the current subpool, and then delete the current subpool.

- Create a VSAM shared resource pool.

  When adding a new VSAM shared resource pool, update the DFSDFxxx member with a POOLID statement, followed by one or more VSRBF subpool definition statements. For example:

```
POOLID=id,Fixdata=,Fixindex,Fixblock,Stringnm=n
VSRBF=bufsize,bufnum,type,HSO,HSR,HSn
```

2. Issue the following command:

```
UPD POOL TYPE(DBAS) SECTION(VSAMxxx) MEMBER(xxx)
```

If the procedure is performed successfully, the IMS system quiesces activities against the target subpools during reconfiguration.

Determine whether the updated database buffer pools configuration is adequate for your workloads:

1. Issue the following command:

```
QUERY POOL TYPE(DBAS) SUBTYPE(OSAM,VSAM) SIZE() POOLID() SHOW(STATISTICS)
```

The IMS system returns statistics about the VSAM buffer pools.

2. Based on the returned statistics, evaluate whether there must be further adjustments to the configuration of VSAM buffer pools.

Related concepts:

"Overview of dynamic database buffer pools" on page 701

Related reference:

➡ DFSDFxxx member of the IMS PROCLIB data set (System Definition)

➡ UPDATE POOL command (Commands)

➡ QUERY POOL command (Commands)

Usage data for OSAM sequential buffering

If you are using OSAM Sequential Buffering, you can use the Sequential Buffering Summary report and the Sequential Buffering Detail report to see how the SB buffers were used during a your program's execution.

By default, four buffer sets exist in each SB buffer pool. If the reports indicate that a large percentage of random read I/O operations were used, and you know that the program was processing your database sequentially, increasing the number of buffer sets to six or more can improve performance. By increasing the number of buffer sets, it is more likely that a block is still in an SB buffer when requested, and a read I/O operation is not necessary.

If only a few random reads were used during your program's execution, it indicates that the database is very well organized and most requests were satisfied from the SB buffer pool or with sequential reads. If this happens, you can save virtual storage space by decreasing the number of buffer sets in each SB buffer pool to two or three.

Related tasks:

"Tuning OSAM sequential buffering" on page 699

Adjusting sequential buffers

You can change the number of buffer sets allocated to each SB buffer pool in two ways.

You can:

- Code an SBPARM control statement with the BUFSETS keyword
- Use an SB Initialization Exit Routine

Once you have changed the number of buffer sets, you can use the SB Test Utility to reprocess the SB buffer handler call sequence that was issued during your program's execution. Then you can study the resulting //DFSSTAT reports to see the impact of the change.

Related reading:

- The Sequential Buffering Summary report and the Sequential Buffering Detail reports are described and instructions on how to use the SB Test Utility are in *IMS Version 15 Database Utilities*.
- Detailed instructions on how to code an SBPARM control statement are in *IMS Version 15 System Definition*.
- Details on the SB Initialization Exit Routine are in *IMS Version 15 Exit Routines*.

Adjusting VSAM options

For VSAM, you can adjust options specified in the OPTIONS control statement and options specified in the Access Method Services `DEFINE CLUSTER` command.

The only VSAM option you can specifically monitor for is background write. If you are not using background write, you can look at the VSAM Buffer Pool report described in *IMS Version 15 System Administration*. The report, in the Number of VSAM Writes To Make Space in the Pool field, documents the number of times data in a buffer had to be written to the database before the buffer could be used. If you use background write, you might find that you are able to reduce this number and therefore the size of the buffer pool.

If you are already using background write, the VSAM Buffer Pool report tells you how many times background write is invoked in the Number of Times Background Write Function Invoked field. The VSAM Statistics report (another report produced by the DB monitor) tells you in the BKG WTS field if background write was invoked. It also tells you, in the USR WRTS field, among other things, how many times background write was invoked.

Related tasks:

"VSAM options" on page 453

"Changing operating system access methods" on page 712

Adjusting VSAM options specified in the OPTIONS control statement

To adjust VSAM options, change the appropriate parameters in the OPTIONS control statement and put the new control statement in the appropriate data set.

The data set you put the control statement in depends on whether you have a batch or online IMS system.
- In a batch system, use the DFSVSAMP data set
- In an online system, use the IMS.PROCLIB data set with the member name DFSVSMnn

Detailed information on how to code these control statements is in *IMS Version 15 System Definition*.

Adjusting VSAM options specified in the Access Method Services DEFINE CLUSTER command

To adjust these VSAM options, change the appropriate parameters in the `DEFINE CLUSTER` command. What additional things you must do depends on which VSAM parameter you are changing.

Changing the FREESPACE parameter

You can use the reorganization utilities to change the use of free space or to change the percent of free space you have specified. To make this change:

1. Unload your database using the existing DBD and the appropriate unload utility.
2. Recalculate database space. You need to do this because the change you are making will result in different requirements for database space.
3. Delete the old database cluster and define the new database cluster with a change to the FREESPACE parameter.
4. Reload your database, using either the existing DBD (if no changes were made to the DBD) or the new DBD. Use the appropriate reload utility.
5. If the database being reorganized is a secondary index with direct pointers, you must run some of the reorganization utilities before and after reloading to resolve prefix information.

Changing the SPEED / RECOVERY parameter

Do not unload and reload your database merely to change the SPEED/RECOVERY parameter. Rather, if you have RECOVERY specified, change the parameter to SPEED to improve performance when the database is reloaded and restart of the load program is not used. IMS does not support the RECOVERY parameter. Recovery can only be done when the database load program is run under control of UCF.

Because it is assumed you would only change the parameter when making other database changes that require you to unload and reload your database, no procedure for changing it is provided here.

Related concepts:

Chapter 26, "Loading databases," on page 559

Related tasks:

"Estimating the minimum size of the database" on page 559

"Offline reorganization by using the reorganization utilities" on page 646

Adjusting OSAM options

To adjust OSAM options, change the appropriate parameters in the OPTIONS control statement. Then put the new control statement in the appropriate data set.

The appropriate data set depends on whether:
* DFSVSAMP data set in a batch system
* IMS.PROCLIB data set with the member name DFSVSMxx in an online system

The OSAM options you can choose are described in "OSAM options" on page 457. Performance implications of each OSAM option are also discussed there. To improve performance, reread that topic and reassess the original choices you made. You cannot specifically monitor any OSAM options.

Detailed information about how to code the OPTIONS control statement is in *IMS Version 15 System Definition*.

Related tasks:

"Changing operating system access methods" on page 712

Changing the amount of space allocated

Change the amount of space allocated for your database in two situations.

The first is when you are running out of primary space. Do not use your secondary space allocation because this can greatly decrease performance. Also change the amount of space allocated for your database when the number of I/O operations required to process a DL/I call is large enough to make performance unacceptable. Performance can be unacceptable if data in the database is spread across too much DASD space.

One way to routinely monitor use of space is by watching the IWAITS/CALL field in the DL/I Call Summary report. If the IWAITS/CALL field has a relatively high number in it, the high number can be caused by space problems. If you suspect space is the problem, you can verify such problems in two specific ways:
* For VSAM data sets, you can get a report from the VSAM catalog using the `LISTCAT` command. In the report, check CI/CA splits, EXCPs, and EXTENTS.
* For non-VSAM data sets, you can get a report on the VTOC using the `LISTVTOC` command. In the report, check the NOEXT field.

If you decide to change the amount of space allocated for your database, do it with JCL or with z/OS utilities. The reorganization utilities must be run to put the database in its new space. The procedure for putting the database in its new space is as follows:

1. Unload your database, using the existing DBD and the appropriate unload utility.
2. Recalculate database space.
3. Delete the old database space for non-VSAM data sets and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
4. If you are changing the space in the root addressable area of an HDAM database, you might need to adjust other HDAM parameters. In this case, you must code a new DBD before reloading (a new DBD is not needed when a

PHDAM partition is changed). To change the space in the root addressable area of a PHDAM partition, you must use the HALDB Partition Definition utility.

5. Reload your database, using either the existing DBD (if no changes were made to the DBD) or the new DBD. Use the appropriate reload utility.

6. If your non-HALDB database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading the database to resolve prefix information.

Related concepts:

Chapter 29, "Monitoring databases," on page 635

➥ DL/I-Call-Summary report (System Administration)

Related tasks:

"Estimating the minimum size of the database" on page 559

"Offline reorganization by using the reorganization utilities" on page 646

Changing operating system access methods

You can use the reorganization utilities to change access methods from OSAM to VSAM, or from VSAM to OSAM.

To change access methods:

1. Unload the database.

2. Code a new DBD (unless you have already done this as described in Step 1).

3. Delete the old data sets and define the new clusters when changing from non-VSAM to VSAM. Delete the old clusters and define new database data sets when changing from VSAM to non-VSAM.

4. You need to change from OSAM options and buffers to VSAM options and buffers or vice versa.

5. Reload your database, using the new DBD. Remember to make an image copy of your database as soon as it is reloaded.

6. If your non-HALDB database uses logical relationships or secondary indexes, you must run reorganization utilities before and after loading the database to resolve prefix information.

Related concepts:

"Adjusting buffers" on page 701

"Adjusting VSAM options" on page 709

Related tasks:

"Adjusting OSAM options" on page 711

"Offline reorganization by using the reorganization utilities" on page 646

Tuning Fast Path systems

Your objective in tuning the IMS online system when Fast Path applications are present depends upon the importance of the message-driven programs and their criteria for acceptable response time.

The performance analysis studies that you should undertake are:

- Examining the availability of sufficient real storage
- Checking the effectiveness of the balancing groups
- Investigating the number of Fast Path dependent regions and the possibility of parallel processing

- Monitoring of the required frequency of DEDB reorganization to reduce fragmented units of work
- Monitoring of the use of DEDB overflow buffers
- Monitoring the forced serialization of programs that concurrently need to use overflow buffers specified by the EXEC statement DBFX parameter
- Examining the area key ranges and whether the randomizing algorithm can be refined
- Reducing the amount of mixed mode processing

Fast Path performance can also be improved by eliminating unnecessary delays caused by the following:
- Transaction volume to a particular Fast Path application program
- DEDB structure considerations
- Contention for DEDB Control Interval (CI) resources
- Exhaustion of DEDB DASD space
- Utilization of available real storage
- Sync point processing and physical logging
- Contention for output threads (OTHR)
- Overhead resulting from reprocessing
- Dispatching priority of processor-dominant and I/O-dominant tasks
- DASD contention caused by I/O on DEDBs
- Resource locking considerations with block level sharing
- Buffer pool usage and not grouping Fast Path application programs with similar buffer use characteristics together into one or more message classes

Statistics on transaction processing and contention for CIs can be obtained from the output of the Fast Path Log Analysis utility (DBFULTA0), which retrieves (from system log input) data relating to the usage of Fast Path resources.

Related concepts:

"Managing unusable space with IMS tools" on page 217

Related reference:

Fast Path Log Analysis utility (DBFULTA0) (System Utilities)

Transaction volume to a particular Fast Path application program

If a disproportionately high number of transactions are queued to a particular balancing group, consider increasing the number of regions associated with that particular balancing group.

The Fast Path Log Analysis report provides information about balancing group queuing.

DEDB structure considerations

Several characteristics of DEDB usage affect the response time of an application program.

These characteristics include:
- Data replication
- Subset pointers

- Number of areas
- Complexity of hierarchical structure
- Complexity of DL/I calls
- Use of sharing across IMS
- Last child pointers
- Recoverability

The first three characteristics are unique to DEDBs; the last five apply generally to databases. Data replication allows up to seven data sets for an individual area. When reading from an area represented by multiple data sets, performance is not impacted, unless the CI is defective. When updating, up to seven additional writes could be required. Although the physical write is performed asynchronously to transaction processing, there could be delays caused by access paths to a variety of DASD devices.

Up to eight subset pointers allow an application program to separate the children of a parent into groups in a DEDB, with the subset pointer pointing to the start of each group. Use of such pointers can help improve performance by reducing the time needed to access segments whose position is significantly displaced in a chain of sequential dependent segments.

Usage of buffers from a Fast Path buffer pool

The Fast Path buffer pool is used by all Fast Path programs except the DEDB online utilities and the High-Speed Sequential Processing (HSSP) function, both of which have their own buffer pool. The Fast Path buffer pool is used to support the processing of MSDBs and DEDBs.

The Fast Path buffer pools in an IMS system can be either defined automatically by IMS by enabling the Fast Path 64-bit buffer manager or defined manually by specifying the DBBF, DBFX, and BSIZ parameters in the IMS and DBC startup procedures.

Related concepts:

[→] Specifying IMS execution parameters (System Definition)

[→] Database buffer pool tuning (System Administration)

Dynamic definition and allocation of Fast Path buffer pools
When the Fast Path 64-bit buffer manager is enabled, it defines and allocates the Fast Path buffer pools dynamically. If specified, the DBBF, DBFP, DBFX, and BSIZ parameters are ignored.

The Fast Path 64-bit buffer manager defines the Fast Path buffer pools dynamically based on buffer usage and CI size requirements. You do not need to specify either the size or number of Fast Path buffers. If IMS detects that more buffers are required or that a CI size does not match any of the currently allocated buffer pools, the Fast Path 64-bit buffer manager allocates the required buffer pools.

The maximum number of buffers that the Fast Path 64-bit buffer manager can allocate to a dependent region is defined by the combined value of the NBA and OBA dependent region parameters.

To alleviate the usage of the extended common storage area (ECSA), the Fast Path 64-bit buffer manager places DEDB buffer pools in 64-bit private storage.

Tip: The Fast Path 64-bit buffer manager does not manage the utility private buffers for Fast Path high-speed processing. These buffers can be located in ECSA or 64-bit common storage. To locate the buffers in 64-bit common storage for high-speed sequential processing (HSSP), specify the SETU FPBP64U=Y control statement in the DFSCTL DD statement. For the High-Speed DEDB Direct Reorganization (DBFUHDR0) utility, specify FPBP64U Y in the SYSIN DD statement.

To enable the Fast Path 64-bit buffer manager, specify FPBP64=Y in the DFSDFxxx PROCLIB member.

Related reference:

> DFSDFxxx member of the IMS PROCLIB data set (System Definition)

Manual definition of Fast Path buffer pools

If you do not use the Fast Path 64-bit buffer manager, you must define the Fast Path buffer pools by specifying the DBBF, DBFX, and BSIZ parameters in the IMS or DBC startup procedures.

To modify the buffer pools, you must change the values of the DBBF, DBFX, and BSIZ parameters and restart IMS. IMS also places all of the Fast Path buffer pools, including the DEDB buffer pools, in ECSA storage.

These parameters do not affect the utility private buffer pools that are used for Fast Path high-speed processing. To locate the utility private buffer pools in 64-bit common storage and not ECSA, include the SETU FPBP64U=Y control statement in the DFSCTL DD statement for high-speed sequential processing (HSSP) and the FPBP64U Y command in the SYSIN DD statement for the High-Speed DEDB Direct Reorganization (DBFUHDR0) utility.

If you do not use the Fast Path 64-bit buffer manager, the Fast Path buffer pool comprises buffers of a size defined at system startup by the BSIZ parameter. The buffer size selected must be capable of holding the largest CI from any DEDB area that is to be opened. The number of buffers page-fixed is based upon the value of supplied parameters:

- The normal buffer allocation (NBA) value causes the defined number of buffers to be fixed in the buffer pool at startup of the dependent region. (This number can be specified for the dependent region startup procedure using the NBA parameter.) The application program in this dependent region is eligible to receive up to this number of buffers within a given sync interval before one of the following occurs:
  - The buffer manager acquires unmodified buffers from the requesting application program.
  - No more buffers can be acquired on behalf of the requesting application program (a number of buffers equal to NBA have been requested, received, and modified). In this case, the buffer manager must acquire access to the overflow buffer allocation (OBA) if this value was specified for this program. If no OBA was specified, then all resources acquired for this program during sync interval processing to date are released.
- The OBA value is the number of buffers that a program can serially acquire when NBA is exceeded. (This number can be specified for the dependent region startup procedure using the OBA parameter.) The overflow interlock function serializes the overflow buffer access, and only one application program at a time can gain access to the overflow buffer allocation. Therefore, the overflow buffer can be involved in deadlocks.

- The DBFX value, which is a system startup parameter, defines a reserve of buffers that are page-fixed upon start of the first Fast Path application program. These buffers are used when asynchronous OTHREAD processing is not releasing buffers quickly enough to support the requests made in sync interval processing.

It follows that:

- BSIZ should be set equal to the largest DEDB CI that will be online. Because the buffer manager does not split buffers to accommodate multiple control intervals, making all DEDB CIs of a same size will provide more optimum use of storage. Even though large block sizes (up to 28K) can be used, this would cause only partial use of the buffer pool if there were many smaller CI sizes.
- The NBA value should be set approximately equal to the normal number of buffer updates made during a sync interval. The NBA value for inquiry-only programs should be small, because the buffers that are never modified can be reused and will all be released at sync time.
- The OBA should be used only in relation to a limited proportion of sync intervals. OBA is not required for inquiry-only programs. In general, the user should be careful to use the OBA value as intended. It should be used to support sync intervals where application program logic demands a variation in total modified buffer needs, thereby requiring access to OBA on an exceptional basis. With BMPs, OBA values greater than 1 should be unnecessary because the 'FW' status code that is returned when the NBA allocation is exceeded can be used to invoke a SYNC call. Invoking a SYNC call would then release all resources. Such application design reduces the serialization and possible deadlocks inherent in using the overflow interlock function.
- The DBFX value should be set, taking into account the total number of buffers that are likely to be in OTHREAD processing at peak load time. If this value is too low, an excessive number of wait-for-buffer conditions are reflected in the IMS Fast Path Log Analysis report.

To optimize the buffer usage, group message processing application programs with similar buffer use characteristics and assign them to a particular message class, so that the applications share the region's buffers.

Related concepts:

Specifying IMS execution parameters (System Definition)

Related reference:

Parameter descriptions for IMS procedures (System Definition)

Contention for DEDB control interval (CI) resources

Queuing takes place on the DEDB CI resource to maintain serialized access on DEDB data. When two independent application programs concurrently request access to a particular CI, one requestor is required to wait.

When such a wait would cause a deadlock, one of the application programs is selected to have its resources released and its processing returned to the previous sync point. (It should be noted that the overflow buffer interlock can also be involved in a deadlock). The rules for selection of the program to be interrupted because of a deadlock are:

- If the deadlock involves one or more message-driven programs, one of the programs is abnormally terminated, reinstated to its previous sync point, and rescheduled.

- If a BMP deadlocks with another BMP, the BMP that went through sync point last is abnormally terminated, has its resources released, is sent back to its previous sync point, and is given a return code.
- If a deadlock involves a DEDB utility, the other program is terminated and rescheduled. Two utilities cannot be involved in a deadlock, because two utilities cannot concurrently access the same DEDB area.

The number of contention and deadlock situations can be decreased by taking the following steps:
- Ensure that CIs contain no more segments than necessary. (CI size is specified in the DBD.)
- Enable the Fast Path 64-bit buffer manager. The Fast Path 64-bit buffer manager dynamically defines, allocates, and manages Fast Path buffer pools based on buffer usage and CI size requirements and, because of multi-threading, allows multiple dependent regions to access overflow buffers at the same time. The Fast Path 64-bit buffer manager is enabled by specifying FPBP64=Y in the DFSDFxxx PROCLIB member.
- If you are not using the Fast Path 64-bit buffer manager, limit the use of the overflow buffer interlock by increasing the NBA value, which, in conjunction with CI usage can be involved in a deadlock.
- Limit the value of NBA to the value necessary to cope with the majority of cases and use OBA to deal with the exceptional conditions. When the full buffer allocation (NBA or NBA and OBA) for a program has been exceeded, the buffer manager can begin stealing unmodified buffers from this program. When all buffers associated with a CI have been stolen, the CI can be released, providing it is not currently in use by a PCB. The buffer stealing and associated CI releasing is triggered by exceeding the full buffer allocation. Minimizing NBA and OBA will assist the timely release of CIs, thereby reducing CI contention.
- Ensure that BMPs accessing DEDBs issue SYNC calls at frequent intervals. (BMPs could be designed to issue many calls between sync points and so gain exclusive control over a significant number of CIs.)
- BMPs that do physical-sequential processing through a DEDB should issue a SYNC call when crossing a CI boundary (provided it is possible to calculate this point). This ensures that the application program never holds more than a single CI.

Reports produced by the Fast Path Log Analysis utility give statistics about CI contention.

Exhaustion of DEDB DASD space

An out-of-space condition (with consequent stoppage of the DEDB area) can occur in the root addressable and sequential dependent portions of an area. Such situations affect the operation of the system as a whole and can necessitate lengthy recovery procedures.

The number of out-of-space conditions can be decreased by:
- Attempting to restrict the number of uses of independent overflow CIs through randomizing algorithm design or regular reorganization
- Deleting sequential dependent CIs on a regular basis
- Using display commands or DEDB POS calls to track space usage

An out-of-space condition can be relieved without bringing IMS down by following the procedures in "Extending DEDB independent overflow online" on page 807.

Utilization of available real storage

The amount of page-fixed storage defined is a significant consideration in limited storage systems.

Related concepts:

"Insert, delete, and replace rules for logical relationships" on page 287

Synchronization point processing and physical logging

Some 'clustering' of output and release of updated CIs and buffers occurs because DEDB updates are deferred until after physical logging is complete.

In BMPs, it helps to minimize the number of updates performed in any one sync interval, particularly if the program is to be run concurrent with the main bulk of message processing.

It is likely that, for performance reasons, the physical log record will be large, so that the log record might not be written for some time during low logging activity. However, IMS varies the interval between the periodic invoking of physical logging. This interval is directly related to the total logging activity in the IMS system. (Low activity causes a smaller interval to be set.)

The physical logging process can be relatively slow because of small physical log buffers or channel or control unit contention for the WADS/OLDS data sets.

The Fast Path environment can have high transaction rates and logging activity. Therefore, the physical configuration supporting the logging process must also be analyzed and altered for optimum performance.

Contention for output threads

Each OTHR defined provides for the possibility of scheduling a separate service request block (SRB) to control the writing of the modified buffers associated with a particular sync interval.

If the OTHR value is low, then queuing of write buffers waiting for an output thread can occur. In general, it is probably best to have one OTHR for each started dependent region that will cause modification of a DEDB.

Overhead resulting from reprocessing

Overhead will result from the necessity to perform reprocessing in either the message-driven or non-message-driven environments.

The following conditions will necessitate reprocessing:
- Deadlocks involving CIs and (possibly) overflow interlock
- Verify failures at sync point time
- User-initiated rollback caused by such conditions as verify failure at call time

In the case of deadlocks, the application program is pseudo abended for dynamic backout. The program controller subtask is detached, and subsequently, reattached. For verify failures or rollback calls, rescheduling involves only the release of resources held and returned to the application program.

Excessive incidence of the above conditions will add to response time and total overhead. Conditions resulting in abend interception followed by dump and application program reinstatement will add to overhead.

Dispatching priority of processor-dominant and I/O-dominant tasks

Because MSDB processing within a sync interval is processor-dominant, application programs processing solely or mainly MSDBs should be dispatched at a lower priority than those programs processing solely or mainly DEDBs (I/O dominant).

DASD contention due to I/O on DEDBs

As always, I/O contention for DEDB Areas will act as a limitation upon performance.

To minimize this impact:
*   Limit the number of heavily-used Areas per device.
*   Limit the number of application programs accessing any one DEDB area. One possibility here is to design the transaction, input edit/routing exit, and randomizing algorithm combination so that the access to any one area is limited to a particular application program or programs.
*   Limit the incidence and effect of stealing unmodified buffers by appropriate application program design. Buffer stealing can necessitate a second I/O to recover the stolen buffer/control interval. This can happen if the logic of the application program requires processing of a buffer when a significant number of calls have been made following the first retrieval.

Maintaining read performance for multiple area data sets

If you use multiple copies of your area data sets (ADSs), place the first ADS registered in the RECON data set on your fastest DASD for the best read performance.

Subsequent copies of the ADS can reside on slower DASD without affecting overall read performance.

IMS always attempts to read from the first ADS shown in the RECON list. If the first ADS is not available or if it is in a long busy state, IMS attempts to read from each subsequent ADS in the list until an available ADS is found. If all of the ADSs are in a long busy state, IMS uses the first ADS in the list.

Resource locking considerations with block-level data sharing

Resource locking can occur either locally in a non-sysplex environment or globally in a sysplex environment.

In a non-sysplex environment, local locks can be granted in one of three ways:
*   **Immediately** because of either of the following reasons:

    IMS was able to get the required IRLM latches, and there is no other interest on this resource.

    The request is compatible with other holders or waiters.
*   **Asynchronously** because the request could not get the required IRLM latches and was suspended. (This can also occur in a sysplex environment.) The lock is granted when latches become available and one of two conditions exist:

    No other holders exist.

The request is compatible with other holders or waiters.
- **Asynchronously** because the request is not compatible with the holders or waiters and was granted after their interest was released. (This could also occur in a sysplex environment.)

In a sysplex environment, global locks can be granted in one of three ways:
- **Locally by the IRLM** because either of the following two reasons:

    There is no other interest for this resource.

    This IRLM has the only interest, this request is compatible with the holders or waiters on this system, and XES already knows about the resource.
- **Synchronously on the XES CALL** because:

    Either XES shows no other interest for this resource.

    Or XES shows only SHARE interest for the hash class.
- **Asynchronously on the XES CALL** because of one of two conditions:

    Either XES shows EXCLUSIVE interest on the hash class by an IRLM, but the resource names do not match (FALSE CONTENTION by RMF).

    Or the request is incompatible with the other HOLDERs and is granted by the CONTENTION Exit after their interest is released (IRLM REAL CONTENTION).

Resource name hash routine

The Fast Path Resource Name Hash routine generates the hash value used by the IRLM.

You can specify the name of such a routine with the UHASH= startup parameter, but it is ignored. IMS always sets the value of UHASH to DBFLHSH0.

One technique used by the IMS-supplied Fast Path Resource Name Hash routine (DBFLHSH0) increases the range of values implicit with the relative CI numbers by combining parts of the 31-bit CI number with values derived from a database's DMCB number and its area number as follows: Bits 11 through 15 of DMCB number are XOR'd with bits 7, 6, 5, 4, 3 of the area number to give a combination 5-bit position number. (Using the area number's bits in reverse order helps make both DMCB number and area number vary the combination value.)

For the relative CI number (bits 0 through 15 are not used):
- Bits 16 through 20 are XOR'd with the combination value.
- Bits 21 through 25 are XOR'd with the combination value.
- Bits 26 through 29 are used unchanged.
- Bits 30 and 31 are not used—thus a hashed CI number used as a GHT entry represents four CIs.

For the hashed resource name:
- Bits 16 through 29 of the hashed relative CI become bits 18 through 31 of the hash value that is passed to the IRLM.
- Bits 18 through 26 of the hash value are used as the displacement into the resource hash table (RHT).
- Bits 18 through 31 are used as the displacement into the GHT.

Chapter 31. Modifying databases

You can modify your database structure in a variety of ways using the reorganization utilities and other methods.

Over time, user requirements can change, necessitating changes in the database design. Or you might choose to use new or different options or features. Or perhaps you have simply found a more efficient way to structure the database.

When you modify your database, you often make more than a simple change to it. For example, you might need to add a segment type and a secondary index. This topic has procedures to guide you through making each type of change.

If you make more than one change at a time, refer to "Changing the number of data set groups" on page 749, which contains a series of flowcharts that, when used with the individual procedures in this topic, can guide you in making some types of multiple changes to the database.

Attention: If the DBD for an existing MSDB is changed, the header information (BHDR) might change, even though the database segments do not. In this case, the headers in the MSDBCPx data sets are invalid or the wrong length. A change in the MSDB headers causes message DFS2593I. If ABND=Y is specified in the MSDB PROCLIB member, ABENDU1012 is also issued. Correct this problem by using the MSDBLOAD option on a warm start or cold start to load the MSDBs from an MSDBINIT data set.

Related concepts:

"Reorganization utilities" on page 644

"Reorganizing the database" on page 643

Chapter 32, "Converting database types," on page 841

➡ Data sharing in IMS environments (System Administration)

Related tasks:

"Reorganizing databases offline" on page 644

Modifying record segments

You can modify record segments in a variety of ways, including adding deleting, or moving segment types, changing the size of a segment, and changing the data stored in a segment.

Related concepts:

"Changing the hierarchical structure of database records" on page 697

Related tasks:

"Modifications to HALDB record segments" on page 833

Adding segment types

There are several ways to add a segment type to a database.

The ways to add a segment type to a database include:
- Unloading and reloading using the reorganization utilities
- Without unloading or reloading

- Using your own unload and reload program

Unloading and reloading using the reorganization utilities

In some cases, you can add segment types to a database record by using the reorganization utilities.

You can add segment types to a database record using the reorganization utilities if:

- The segment type to be added is at the bottom level of a path in the hierarchy. The following figure shows an existing database record (indicated by solid lines) and the places where a new segment type can be added (indicated by dashed lines).
- The existing relative order of segments in the database record does not change. In other words, the existing parent to child relationships cannot change.
- The existing segment names do not change.
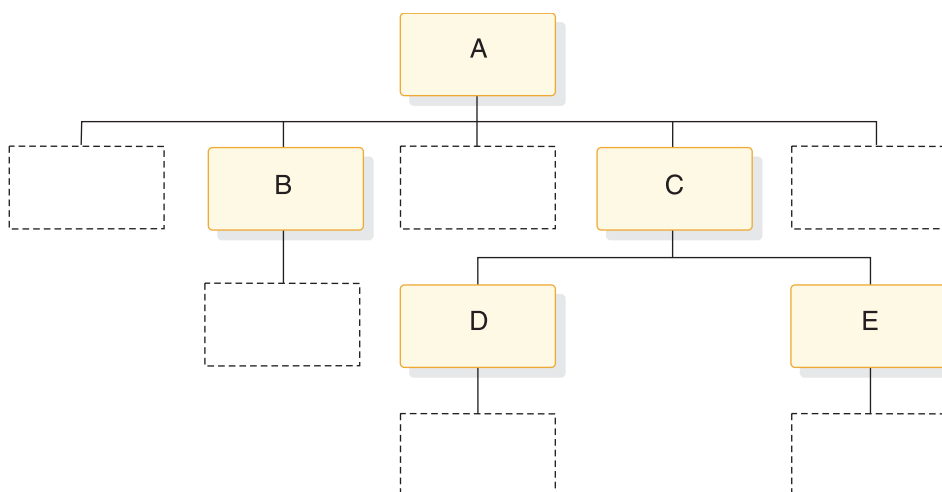
Figure 270. Where segment types can be added in a database record

To use the reorganization utilities to add a segment type to the database:

1. Determine if the change you are making affects the code in any application programs. If the code is affected, make the necessary changes to the application program.
2. Unload your database, using the existing DBD.
3. Code a new DBD. You need to add SEGM= statements to the DBD for the new segment type. No database updates are allowed between unload and reload.
4. If the change you are making affects the code in application programs, make any necessary changes to the PSBs for those application programs.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space. You need to do this because the change you are making will result in different requirements for database space.
7. For non-VSAM data sets, delete the old database space and define the new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Reload your database, using the new DBD. Make an image copy of your database as soon as it is reloaded.

9. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.
10. Code and execute an application program to insert the new segment types into the database.

Related tasks:

"Estimating the minimum size of the database" on page 559

"Offline reorganization by using the reorganization utilities" on page 646

Without unloading or reloading

You can add segment types to a database record without unloading the database under the following circumstances.

- In a HISAM database, the segment type to be added must be the last segment in the hierarchy. In addition, the segment type to be added must fit in the existing logical record.
- In an HD database, the segment type to be added must also be the last segment in the hierarchy. The parent of the new segment type must use hierarchical pointers. Also, the segment type cannot be the largest segment type in the data set group.

To add a segment type to the database without unloading and reloading:
1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Code a new DBD. You need to add a SEGM= statement to the DBD for the new segment type.
3. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
4. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
5. Code and execute an application program to insert the new segment type.

Using your own unload and reload program

If you must unload your database to add a segment type and you cannot use the reorganization utilities, you must write your own unload and reload program.

Generally, you need to write your own unload and reload program if the new segment type changes the hierarchical structure of or relationship between the existing segments in the database.

Deleting segment types

You can delete a segment type from a database by using either the reorganization utilities provided by IMS or by using your own unload and reload program.

You can delete a segment type from a database, using the reorganization utilities, if:

- The existing relative order of segments in the database record does not change. In other words, the existing parent to child relationships cannot change.
- The existing segment names do not change.

To use the reorganization utilities to delete a segment type from the database:

1. Code and execute an application program to delete all occurrences of the segment type being deleted. You must code and execute the application program before the database is unloaded.
2. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
3. Unload your database, using the existing DBD.
4. Code a new DBD. You need to remove SEGM= statements from the DBD for:
   - The segment type being deleted
   - The children of the deleted segment.
5. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
6. Recalculate database space. You need to do this because the change you are making will result in different requirements for database space.
7. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
8. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
9. Reload your database using the new DBD. Remember to make an image copy of your database as soon as it is reloaded.
10. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.

Related tasks:

"Estimating the minimum size of the database" on page 559

"Offline reorganization by using the reorganization utilities" on page 646

Moving segment types

Because segment types cannot be moved using the reorganization utilities, you must write your own unload and reload program to move them.

Changing segment size

Using the reorganization utilities, you can increase or decrease segment size at the end of a segment type.

When increasing segment size, you are adding data to the end of a segment. When decreasing segment size, IMS truncates data at the end of a segment.

If you are increasing the size of a segment, you cannot predict what is at the end of the segment when it is reloaded. Also, new data must be added to the end of a segment using your own program after the database is reloaded.

To increase or decrease segment size:

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload your database, using the existing DBD. If you are changing a HISAM database, you must use the HD UNLOAD/RELOAD utility since the HISAM utilities cannot be used to make structural changes.
3. Code a new DBD. You need to change the BYTES= operand on the SEGM statement in the DBD to reflect the new segment size. If you are eliminating data from a segment for which FIELD statements are coded in the DBD, you

need to eliminate the FIELD statements. If you are adding data to a segment and the data is referenced in the SSA in application programs, you need to code FIELD statements. No database updates are allowed between unload and reload.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than build dynamically.

6. Recalculate database space. You need to do this because the change you are making results in different requirements for database space.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Reload your database, using the new DBD. Make an image copy of your database as soon as it is reloaded.

9. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.

Related tasks:

"Estimating the minimum size of the database" on page 559

"Offline reorganization by using the reorganization utilities" on page 646

Adding or converting to variable-length segments

If you need to change selected segments in your database from fixed to variable length—or convert the entire database to variable-length segments—two ways exist to do it.

Regardless of which way you use, the object in conversion is to put a size field in the segment you need to make variable length and then get the segment defined as variable length in the DBD.

Related concepts:

"Variable-length segments" on page 387

Method 1. Converting segments or a database

Method 1 for converting segments or a database to use variable length segments involves creating a new DBD and writing your own application program to retrieve each segment, add the 2-byte size field, and then insert the segment back into the database.

To convert selected segments or the entire database this way, you must:

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Code and generate a new DBD that identifies the segment types that will be variable length, and their size.

3. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

4. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

5. Write a program that sequentially retrieves from the database all segments that are to be variable length. Your program must add the 2-byte size field to each segment retrieved and then insert the segment back into the database.

Method 2. Converting segments or a database

Method 2 for converting segments or a database to use variable length segments involves creating two DBDs (an interim DBD and a final DBD), unloading the database, and then using a segment edit/compression exit routine to add the 2-byte size field during the reload of the database.

To convert selected segments or the entire database this way, you must:

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload your database, using the existing DBD.
3. Code and generate a new (interim) DBD. This DBD should specify fixed-length segments for all segments being converted to variable length. It should also specify the use of the segment edit/compression exit routine for each segment to be converted. (The interim DBD is used, as explained in Step 9, to add a size field to the existing fixed-length segments.)
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space if necessary. You need to do this when the change you are making results in different requirements for database space.
7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Write an edit routine to which the segment edit/compression exit routine can exit. Your edit routine should add a size field to each segment it receives.
9. Reload the database, using the interim DBD. As each occurrence of a segment type that needs to be converted is presented for loading, your edit routine gets control and adds the size field to the segment. When your edit routine returns control, the segment is loaded into the database. Remember to make an image copy of your database as soon as it is loaded.
10. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.
11. After the database is loaded, code and generate a new DBD that specifies the segment types in the database that are variable, and their size.

Related concepts:

"Segment Edit/Compression exit routine" on page 390

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Changing data in a segment (except for data at the end of a segment)

Data in a segment cannot be increased or decreased in size using the reorganization utilities. To increase or decrease the size of fields, you must write your own unload and reload programs.

Changing the position of data in a segment

You cannot change the position of data in a segment using the reorganization utilities.

To make this kind of change, you must write your own unload and reload program, use field-level sensitivity, or use a tool, such as the IMS High Performance Pointer Checker for z/OS, which includes a DB Segment Restructure utility.

Related Reading: For information about the IMS High Performance Pointer Checker for z/OS, see *IMS High Performance Pointer Checker for z/OS User's Guide*.

Related concepts:

"Field-level sensitivity" on page 399

Changing the name of a segment

If you change the name of a segment without changing anything else about the segment, you do not need to unload and reload the database or otherwise reorganize the database.

To change the name of a segment, you must:

1. Change the segment name in the DBD statement that defines the segment.
2. If the segment is referenced in a logical relationship or secondary index, you must change the segment name in the DBD for the logical or index database.
3. Update the DBDLIB by running the DBD Generation utility.
4. Change the segment name in the PSB statements that reference the segment.
5. Update the PSBLIB by running the PSB Generation utility.
6. Update the ACBLIB by running the ACB Maintenance utility and performing Online Change.
7. Update all application code that references the segment.

Related reference:

� Database Description (DBD) Generation utility (System Utilities)

▸ Program Specification Block (PSB) Generation utility (System Utilities)

▸ Application Control Blocks Maintenance utility (System Utilities)

Adding logical relationships

You can add logical relationships to an existing database.

This topic contains examples and procedures for adding a logically-related database to an existing database. Not all situations in which you might need to add a logical relationship are described in this topic. However, if the examples do not fit your specific requirements, you should be able to gather enough information from them to decide:

- If adding a logical relationship to your existing database is possible
- How to add the relationship

The examples show the logical parent as a root segment, although this is not a requirement. The examples are still valid when the logical parent is at a lower level in the hierarchy.

When adding logical relationships to existing databases, you should always make the change on a test database. Thoroughly test the change before implementing it using production databases.

In the following examples, these conventions are used:

- Existing databases are shown using solid lines.
- The database being added is shown using dashed lines.
- The logical parent and logical child relationship is labeled for the database being added. They are labeled LP and LC.
- The terms DBX, DBY, and DBZ refer to database 1, database 2, and database 3.

Related concepts:

Chapter 17, "Creating logical relationships," on page 245

Related tasks:

"Estimating the minimum size of the database" on page 559

"Writing a load program" on page 574

Examples of adding logical relationships

Each of the examples include a figure to illustrate the addition of a logical relationship. The steps required to add the logical relationship shown in each example are also provided.

Example 1. DBX exists, DBY is to be added

In example 1, DBX must be reorganized to add the counter field to the segment prefix for A.

Example 1 is shown in the following figure.
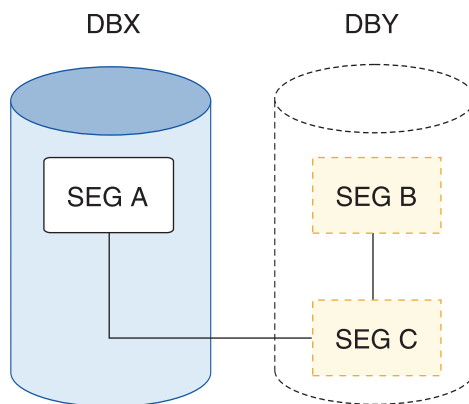
Figure 271. DBX exists, DBY is to be added

DBIL must be specified in the control statement for DBX. In the following steps, the counter field for segment A is updated to show the number of C segments because segment C is loaded with a user load program.

Example 1 procedure

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload DBX, using the existing DBD and the HD Unload utility.
3. Code a new DBD for DBX and DBY.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space for DBX and calculate space for DBY.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Run the Prereorganization utility, specifying DBIL in the control statements for DBX and DBY.

9. Reload DBX, using the new DBD and the HD Reload utility.

10. Load DBY, using an initial load program.

11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.

12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.

13. Remember to make an image copy of both databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Example 2. DBX and DBY exist, DBZ is to be added

In this example, the counter exists in the segment C prefix. DBX and DBY must be reorganized to calculate the new value for the counter in the segment C prefix.

Example 2 is shown in the following figure.
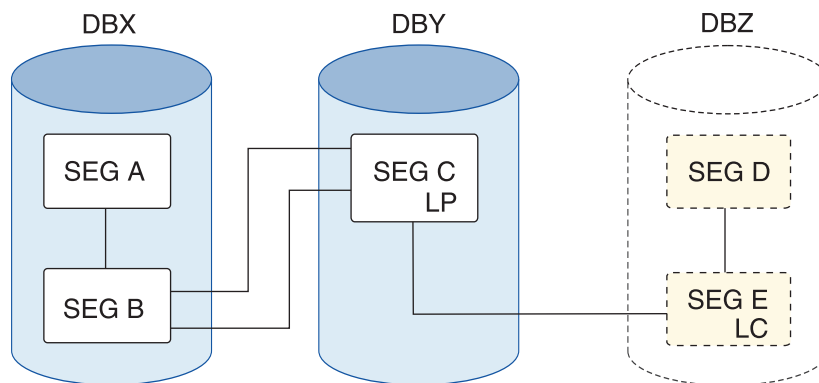
Figure 272. DBX and DBY exist, DBZ is to be added

DBIL must be specified in the control statement for DBX and DBY. In the following steps, the segment A counter field is updated to show the number of C segments because segment C is loaded with a user load program.

Example 2 procedure

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Unload DBX and DBY, using the existing DBDs and HD Unload utility.

3. Code a new DBD for DBY and DBZ.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBX and DBY, and calculate space for DBZ.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Run the Prereorganization utility, specifying DBIL in the control statements for DBX, DBY and DBZ.
9. Reload DBX and DBY, using the new DBDs and the HD Reload utility.
10. Load DBZ, using an initial load program.
11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.
12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.
13. Remember to make an image copy of all three databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Example 3. DBX and DBY exist, DBZ is to be added

In example 3, DBY must be reorganized to add the counter field to the segment C prefix. DBIL must be specified in the control statement for DBY. DBX must be reorganized because an initial load (DBIL) of the logical parent (segment C) assumes an initial load (DBIL of the logical child).

Example 3 is shown in the following figure.
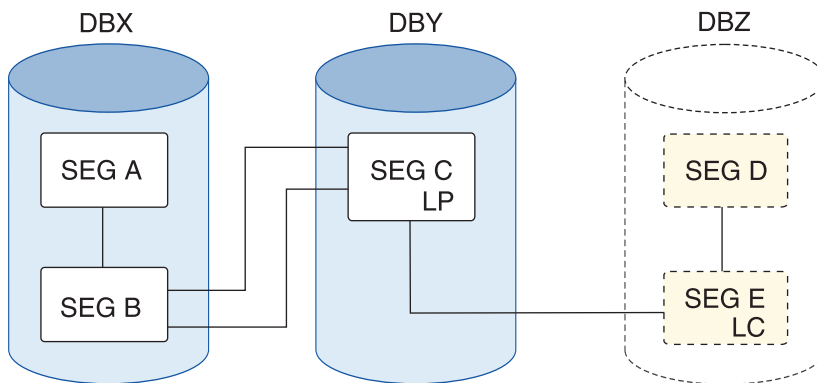
Figure 273. DBX and DBY exist, DBZ is to be added

The procedure for this example (and all conditions and considerations) is exactly the same as example 2.

Example 4. DBX and DBY exist, DBZ is to be added

In example 4, the procedure and all conditions and considerations are exactly the same as for example 2.

Example 4 is shown in the following figure.
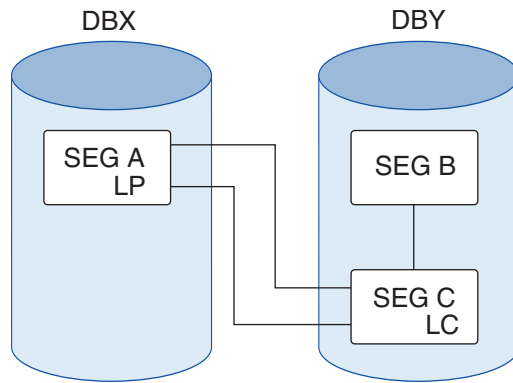
Figure 274. DBX and DBY exist, DBZ is to be added

Example 5. DBX exists, DBY is to be added

In example 5, DBX must be reorganized to add the logical child pointers in the segment A prefix.

Example 5 is shown in the following figure.
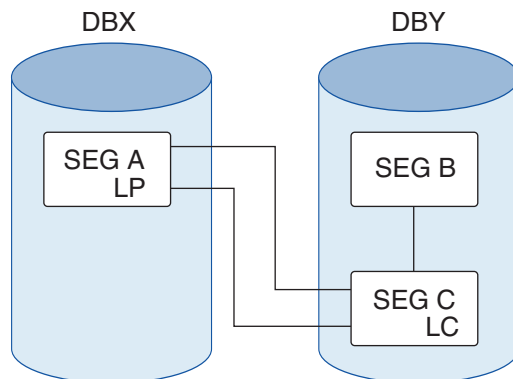
Figure 275. DBX exists and DBY is to be added

Procedure

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload DBX, using the existing DBD and the HD Unload utility.
3. Code a new DBD for DBX and DBY.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space for DBX, and calculate space for DBY.
7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Run the Prereorganization utility, specifying DBR in the control statement for DBX, and DBIL in the control statement for DBY.

9. Reload DBX, using the new DBD and the HD Reload utility.
10. Load DBY, using an initial load program.
11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.
12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.
13. Remember to make an image copy of both databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Example 6. DBX and DBY exist, DBZ is to be added

In example 6, DBY must be reorganized to add the logical child pointers to the segment C prefix.

Example 6 is shown in the following figure.
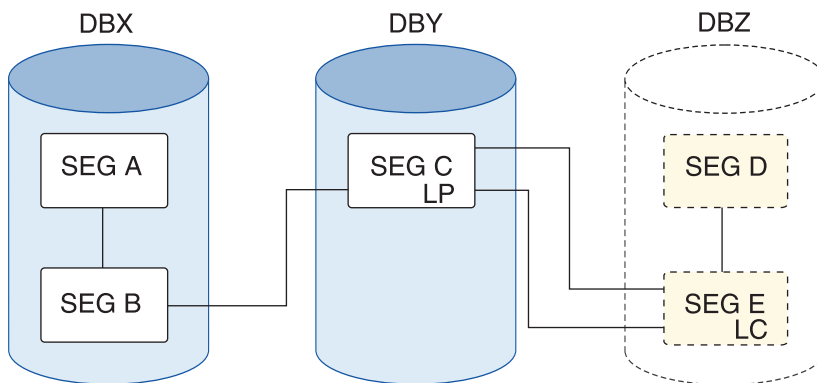
Figure 276. DBX and DBY exist, DBZ is to be added

One of the following three procedures should be used to add the logical child pointers to the segment C prefix:

Procedure when reorganizing DBY (segment B contains a symbolic pointer)

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload DBY, using the existing DBD and HD Unload utility.
3. Code a new DBD for DBY and DBZ.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space for DBY, and calculate space for DBZ.
7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Run the Prereorganization utility, specifying DBR in the control statement for DBY, and DBIL in the control statement for DBZ. (The output from the Prereorganization utility indicates that a scan of DBX is required.)

9. Reload DBY, using the new DBD and the HD Reload utility.

10. Load DBZ, using an initial load program.

11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.

12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.

13. Remember to make an image copy of both databases as soon as they are loaded.

When DBY is reloaded, two type 00 records are produced for each occurrence of segment C. One contains a logical child database named DBZ and matches the type 10 record produced for segment E. The other contains a logical child database named DBX. Because a scan or reorganization of DBX was not done, a matching 10 record was not produced for segment B. The Prefix Resolution utility produces message DFS878 when this occurs. The message can be ignored as long as the printed 00 record refers to DBY and DBX. Any messages for DBY and DBZ should be investigated.

Procedure when reorganizing DBY and scanning DBX (segment B contains a direct pointer)

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Unload DBY, using the existing DBD and HD Unload utility.

3. Code a new DBD for DBY and DBZ.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBY, and calculate space for DBZ.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Run the Prereorganization utility, specifying DBR in the control statement for DBY, and DBIL in the control statement for DBZ. (The output from the Prereorganization utility says that a scan of DBX is required.)

9. Run the scan utility against DBX.

10. Reload DBY, using the new DBD and the HD Reload utility.

11. Load DBZ, using an initial load program.

12. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9, 10, and 11 as input.

13. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 12 as input.

14. Remember to make an image copy of both databases as soon as they are loaded.

Procedure when reorganizing DBX and DBY

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Unload DBX and DBY, using the existing DBDs and HD Unload utility.

3. Code a new DBD for DBY and DBZ.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBX and DBY, and calculate space for DBZ.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Run the Prereorganization utility, specifying DBR in the control statements for DBX and DBY, and DBIL in the control statement for DBZ. (The output from the Prereorganization utility says that a scan of DBX is required.)

9. Reload DBX and DBY, using the new DBDs and the HD Reload utility.

10. Load DBZ, using an initial load program.

11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.

12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.

13. Remember to make an image copy of all three databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Example 7. DBX and DBY exist, DBZ is to be added

In example 7, DBY must be reorganized to add the logical child pointers to the segment C prefix. Logical child pointers from segment C to segment B are not unloaded, therefore, DBX must be reorganized or scanned. DBX must be reorganized to add the logical child pointers in the segment A prefix.

Example 7 is shown in the following figure.
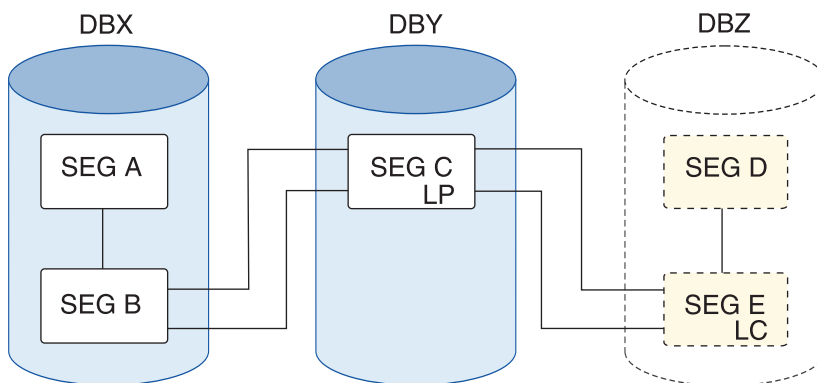
Figure 277. DBX and DBY exist, DBZ is to be added

Procedure using scan

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Unload DBY, using the existing DBD and HD Unload utility.

3. Code a new DBD for DBY and DBZ.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBY and calculate space for DBZ.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Run the Prereorganization utility, specifying DBR in the control statements for DBY, and DBIL in the control statement for DBZ. (The output from the Prereorganization utility indicates that a scan of DBX is required.)

9. Run the scan utility against DBX.

10. Reload DBY, using the new DBDs and the HD Reload utility.

11. Load DBZ, using an initial load program.

12. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9, 10, and 11 as input.

13. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 12 as input.

14. Remember to make an image copy of both databases as soon as they are loaded.

Procedure when reorganizing DBX and DBY

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.

2. Unload DBY and DBY using the existing DBDs and the HD Unload utility.

3. Code a new DBD for DBY and DBZ.

4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.

5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBX and DBY and calculate space for DBZ.

7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

8. Run the Prereorganization utility, specifying DBR in the control statements for DBX and DBY, and DBIL in the control statement for DBZ. (The output from the Prereorganization utility indicates that a scan of DBX is required.)

9. Reload DBX and DBY, using the new DBDs and the HD Reload utility.

10. Load DBZ, using an initial load program.

11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 input.

12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.

13. Remember to make an image copy of both databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Example 8. DBX and DBY exist, DBZ is to be added

In example 8, DBY must be reorganized to add the logical child pointers in the segment C prefix. The procedure for this example and all conditions and considerations are exactly the same as the procedures for example 6.

Example 8 is shown in the following figure.
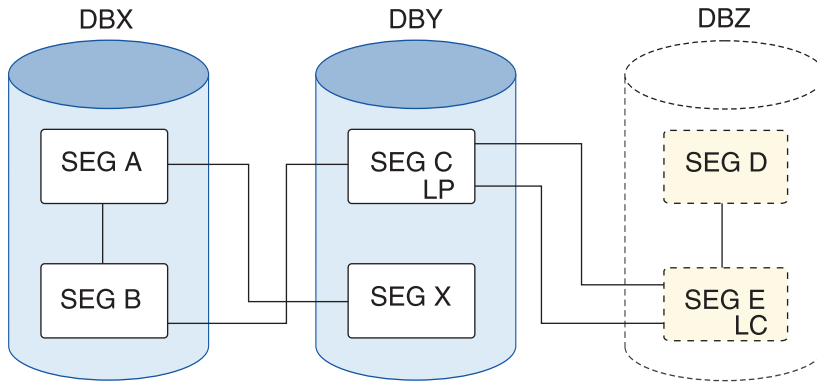
Figure 278. DBX and DBY exist, DBZ is to be added

Example 9. DBY exists, DBZ is to be added

DBY must be reorganized. DBZ must be loaded using an initial load program. DBIL must be specified in the control statement for DBY. Do not specify DBR in the control statement for DBY.

Example 9 is shown in the following figure.
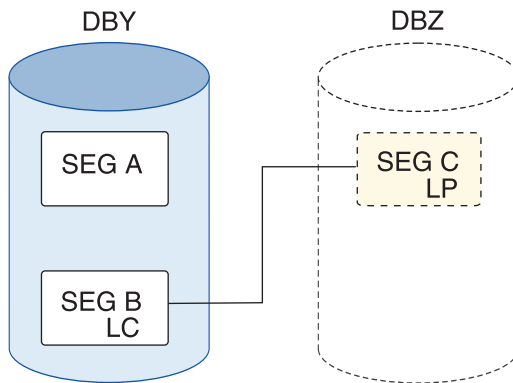
Figure 279. DBY exists, DBZ is to be added

Procedure

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload DBY, using the existing DBD and HD Unload utility.
3. Code a new DBD for DBY and DBZ.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.

6. Recalculate database space for DBY and calculate space for DBZ.
7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Run the Prereorganization utility, specifying DBIL in the control statements for DBY and DBZ.
9. Reload DBY, using the new DBDs and the HD Reload utility.
10. Load DBZ, using an initial load program.
11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.
12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.
13. Remember to make an image copy of both databases as soon as they are loaded.

Example 10. DBY exists, DBZ is to be added

In example 10, segment X might be considered a logical child if the key of segment D is at the correct location in segment X. DBY must be reorganized, because an initial load (DBIL) of the logical parent (segment D) assumes an initial load (DBIL) of the logical child.

In this example, you could use symbolic or direct pointers for segment X. Do **not** under any circumstances specify DBR in the control statement for DBY. If you do, the reload utility will not generate work records for segment D; the logical child pointer in segment D would never be resolved. The procedure for this example (and all conditions and considerations) is exactly the same as the procedures for example 9.

Example 10 is shown in the following figure.
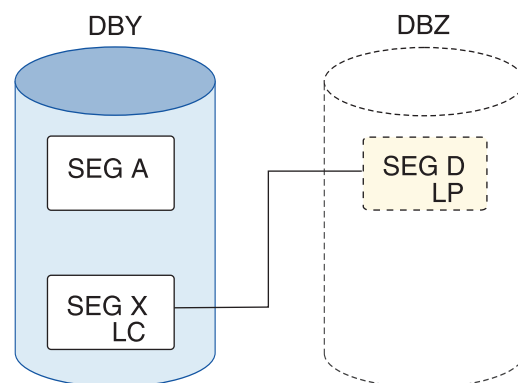
Figure 280. DBY exists, DBZ is to be added

Example 11. DBX and DBY exist, DBZ is to be added

In example 11, DBX and DBY must be reorganized. DBZ must be loaded using an initial load program. Because you must specify DBIL in the control statement for DBZ (a logical parent database), you must also specify DBIL for DBY (a logical child database). DBY is also a logical parent database. Therefore, you must specify DBIL in the control statement for DBX (a logical child database).

The procedure for this example and all conditions and considerations are exactly the same as for Example 2.

Example 11 is shown in the following figure.
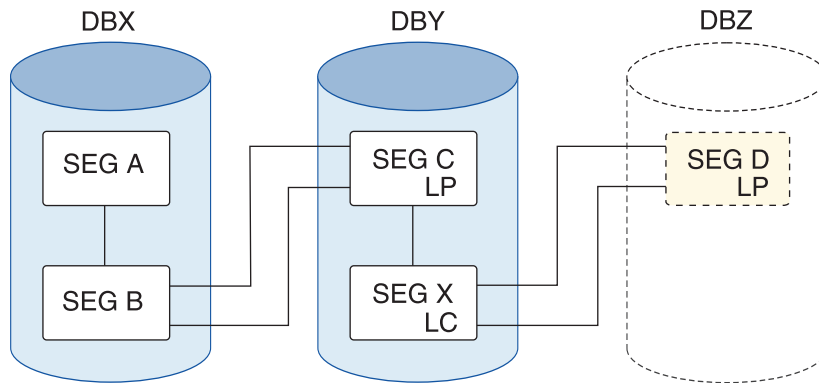
Figure 281. DBX and DBY exist, DBZ is to be added

Example 12. DBX and DBY exist, DBZ is to be added

In example 12, segment B has a symbolic pointer. The procedure for this example and all conditions and considerations are exactly the same as for example 2.

Example 12 is shown in the following figure.
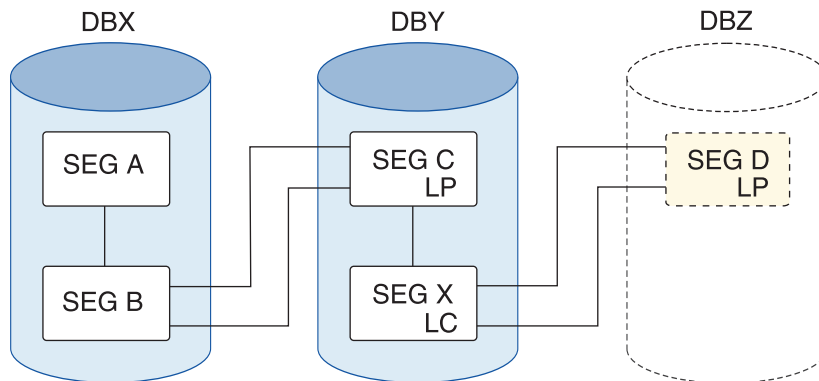
Figure 282. DBX and DBY exist, DBZ is to be added

Example 13. DBX and DBY exist, segment Y and DBZ are to be added

Example 13 is shown in the following figure.

Before

After
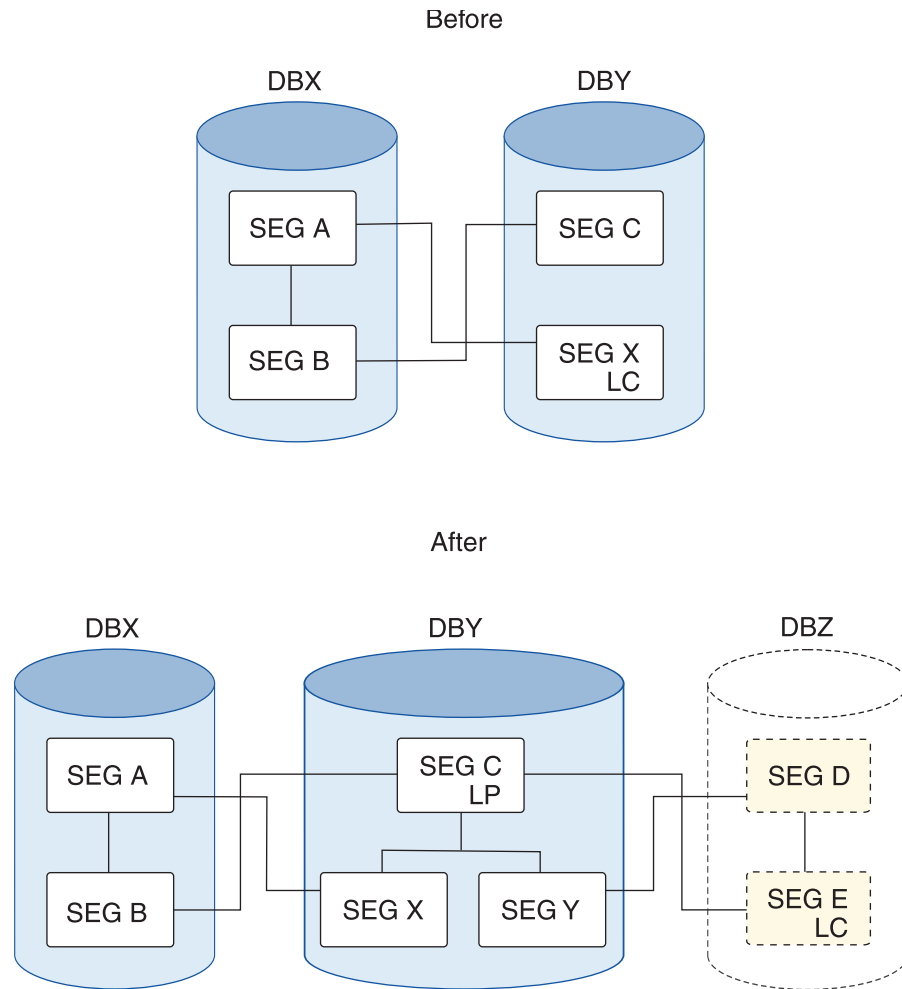
Figure 283. DBX and DBY exist, segment Y and DBZ are to be added

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload DBX, using the existing DBD and HD Unload utility.
3. Code a new DBD for DBY and DBZ.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for these application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space for DBX and DBY, and calculate space for DBZ.
7. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
8. Run the Prereorganization utility, specifying DBIL in the control statements for DBX, DBY and DBZ.
9. Reload DBX, using the new DBD and the HD Reload utility.
10. Load DBY and DBZ, using an initial load program.
11. Run the Prefix Resolution utility, using the DFSURWF1 work files that are output from Steps 9 and 10 as input.

12. Run the Prefix Update utility, using the DFSURWF3 work file that is output from Step 11 as input.
13. Remember to make an image copy of both databases as soon as they are loaded.

Related tasks:

"Specifying logical relationships in the logical DBD" on page 277

Altering IMS logical relationships

The steps in the table below outline the steps required to add or modify logical relationships by reorganizing a database.

The table below shows you what you need to do based on the logical relationships and pointers you are starting from in column 1, the databases you are reorganizing in column 2, and the logical relationships and pointers you are adding in columns 3 to 6. Column 1 is the "FROM" column. Columns 3 to 6 are the "TO" columns.

At the intersection between the existing and new logical relationships and pointers, the table provides the following information:

- When a logically related database must be scanned
- When both sides of a logical relationship must be reorganized
- When the Prefix Resolution and Prefix Update utilities must be run

If the Prefix Resolution and Prefix Update utilities do not need to be run, the table cell contains "finished".

The figure applies to reorganizations only. When initially loading databases, you must run the Prefix Resolution and Update utilities whenever work data sets are generated.

The following table covers all reorganization situations, whether or not database pointers are being changed. In using the figure, a bidirectional physically paired relationship should be treated as two unidirectional relationships. Unless otherwise specified, DBR should be specified for the reorganized databases when the Prereorganization utility is run.

The following two examples guide you in use of the table.

Example 1: Reorganizing a database with unidirectional symbolic pointers without changing pointers

Assume your database has unidirectional symbolic pointers and you are not changing pointers. On the left side of the table below, in the FROM column, find unidirectional symbolic pointers and follow the row across to the right until it intersects with the TO column of unidirectional symbolic pointers. The figure tells you what you must do to reorganize with one of the following:

- The database containing the logical parent
- The database containing the logical child
- Both databases, if necessary

In all three situations, it is not necessary to run the Prefix Resolution or Update utilities (this is what is meant by "finished").

Example 2: Changing bidirectional symbolic pointers to bidirectional direct pointers during a reorganization

Assume your database has bidirectional symbolic pointers and you need to change to bidirectional direct pointers. In the FROM column, find the row for bidirectional symbolic pointers. In the TO columns, find bidirectional direct pointers. Where the row and column intersect, the table below shows that:

- Reorganizing only the logical parent database cannot be done, because a logical parent pointer must be created in the logical child segment in the logical child database.
- Reorganizing the logical child database can be done. To scan the logical child database, you must scan the logical parent database. The control statements for the Prereorganization utility must specify DBIL for the logical child database. Also, the Prefix Resolution and Update utilities must be run.
- Reorganizing both databases can also be done. In this case, the control statements for the Prereorganization utility must specify DBIL for the logical child database and DBR for the logical parent database. Again, the Prefix Resolution and Update utilities must be run.

Table 83. Steps required to convert logical relationships and pointers during a database reorganization.

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	Unidirectional direct pointers	Bidirectional symbolic pointers	Bidirectional direct pointers
Unidirectional with symbolic pointers	Logical parent database only	Finished[1]	Not valid, because symbolic LP pointers exist now and direct LP pointers must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** Logical child segment will not contain LT pointers unless it is reorganized.	Not valid, because direct LP and LT pointers must be put in the logical child database.
	Logical child database only	Finished	1. Scan logical parent database. 2. Run prefix resolution and update. Specify DBIL for the logical child database.	Not valid, because a counter exists now and LCF/LCL pointers must be put into the logical parent database.	Not valid, because a counter exists now and LCF/LCL pointers must be put into the logical parent database.
	Both databases	Finished[2]	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBR for both databases.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.

Table 83. Steps required to convert logical relationships and pointers during a database reorganization (continued).

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	Unidirectional direct pointers	Bidirectional symbolic pointers	Bidirectional direct pointers
Unidirectional with direct pointers	Logical parent database only	Not valid, because a direct LP pointer exists now and symbolic LP pointers must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update.	Not valid, because a direct LP pointer exists now and symbolic LP pointers must be added to the logical child database. LT pointers must also be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** Logical child segment will not contain LT pointers unless database is reorganized.
	Logical child database only	Finished	Finished	Not valid, because LCF/LCL pointers must be put in the logical parent database.	Not valid, because LCF/LCL pointers must be put in the logical parent database.
	Both databases	Finished[2]	Run prefix resolution and update.	Run prefix resolution and update.	Run prefix resolution and update.
Bidirectional with symbolic pointers	Logical parent database only	Not valid, because the counter in the logical parent database will not be resolved and LT pointers exist now in the logical child database.	Not valid, because symbolic LP and LT pointers exist now and a direct LP pointer must be added to the logical child database.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** LCF/LCL pointers are not unloaded and reloaded.	Not valid, because a symbolic LP pointer exists now and a direct LP pointer must be added to the logical child database.
	Logical child database only	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	1. Scan logical parent database. 2. Run prefix resolution and update.	1. Scan logical parent database. 2. Run prefix resolution and update. 3. Specify DBIL for the logical child database.
	Both databases	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.

FROM: existing logical relationship and pointers	Databases to be reorganized	TO: new logical relationship and pointers			
		Unidirectional symbolic pointers	Unidirectional direct pointers	Bidirectional symbolic pointers	Bidirectional direct pointers
Bidirectional with direct pointers	Logical parent database only	Not valid, because direct LP and LT pointers exist in the logical child database and symbolic LP pointers must be added.	Not valid, because the counter in the logical parent database will not be resolved and LT pointers will not be removed from the logical child database.	Not valid, because a direct LP pointer exists in the logical child database and the change is to symbolic LP pointers.	1. Scan logical child database. 2. Run prefix resolution and update. **Note:** LCF/LCL pointers are not unloaded and reloaded.
	Logical child database only	Not valid, because LCF/LCL pointers exists in the logical parent database and a counter must be added to the logical parent database.	Not valid, because LCF/LCL pointers exist now in the logical parent database and a counter must be added to the logical parent database.	1. Scan logical parent database. 2. Run prefix resolution and update.	1. Scan logical parent database. 2. Run prefix resolution and update.
	Both databases	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update. Specify DBIL for the logical child database and DBR for the logical parent database.	Run prefix resolution and update.	Run prefix resolution and update.

Note:

1. The Prereorganization utility says to scan the logical child database and the DFSURWF1 records will be produced if scan is run.

2. DFSURWF1 records are produced; however, the prefix resolution and update utilities need *not* be run.

Related concepts:

"Defining sequence fields for logical relationships" on page 271

Some restrictions on modifying existing logical relationships

In some cases, the IMS utilities cannot be used to modify an existing logical relationship. When an existing logical relationship cannot be modified, you must write your own program.

The topics below provide examples.

Example 1: changing from bidirectional virtual to bidirectional physical pairing

The following example shows the change in pairing from virtual to physical.
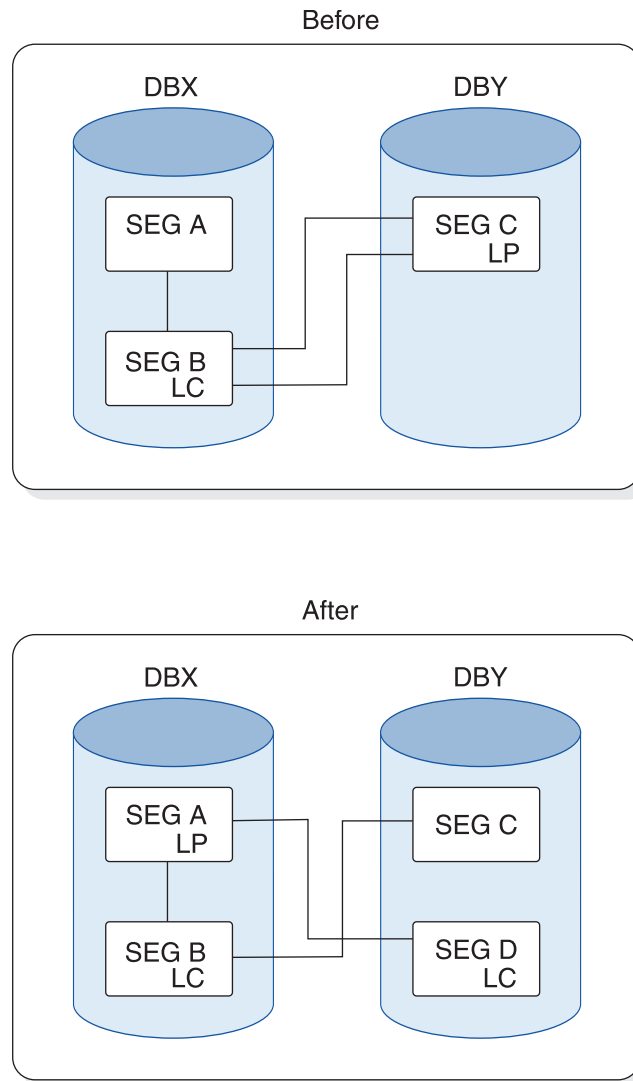
Figure 284. The change in pairing from virtual to physical

Example 2: changing the location of the real logical child in a bidirectional logical relationship

The following figure shows the position change of a real logical child from one logically related database to another.

Before

After
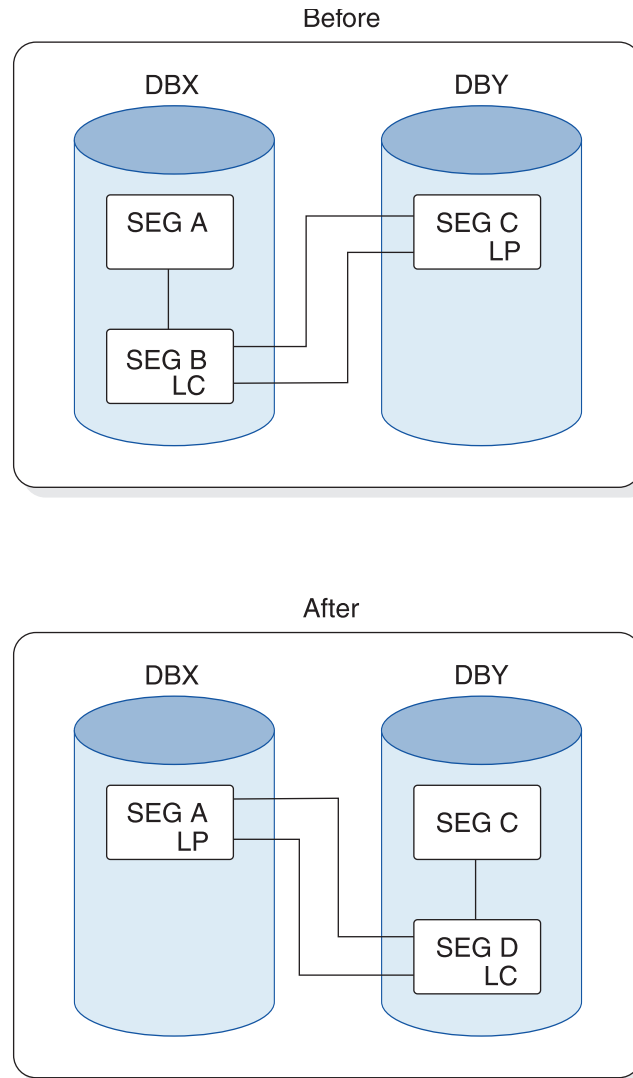
Figure 285. The position change of a real logical child from one logically related database to another

In both of these "before" examples, occurrences of segment B can exist that are physically, but not logically, deleted. The logical child can be accessed from the logical path but not the physical path. When unloading DBX, the HD Unload utility cannot access occurrences of segment B that are physically, but not logically, deleted. Therefore, you must write your own program to do this type of reorganization.

Summary on use of utilities when adding logical relationships

The following points summarize adding logical relationships with the IMS utilities.

- Counters are increased by counting logical children loaded using an initial load program or, when logically related databases are reorganized, by using DBIL in the control statement.
- Counter problems can be corrected by reorganizing databases. When correcting counter problems, DBIL must be specified in the control statement for the databases involved.

- LCF and LCL pointers are not unloaded and reloaded. They must be recreated by the Prefix Resolution and Update utilities.
- Unless DBIL is specified for all its logical child databases, never specify DBIL in the control statement for a logical parent database.
- To change from symbolic to direct pointers, specify DBIL on the control statement for the logical child database.

Converting a logical parent concatenated key from virtual to physical or physical to virtual

You can convert a logical parent concatenated key from virtual to physical or from physical to virtual by using DBDGEN and the HD reorganization utilities.

To convert a logical parent concatenated key from virtual to physical or from physical to virtual:

1. Unload your database, using the existing DBD.
2. Code a new DBD, changing the concatenated key physical/virtual specification.
3. If you use prebuilt ACBs rather than dynamically built ACBs, rebuild the ACB.
4. Recalculate the database space. You need to do this because the change you are making changes database space requirements.
5. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
6. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.
7. Reload your database using the new DBD. Remember to make an image copy of your database as soon as it is reloaded.
8. If required, run reorganization utilities to resolve prefix information.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Altering IMS indexes

You can add or remove a secondary index from a full-function or a DEDB database.

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

"How secondary indexes restructure the hierarchy of databases" on page 342

Related tasks:

"Dropping an index" on page 749

Adding a secondary index to a full-function database

You can add a secondary index to a full-function database to process a segment type in a sequence other than the one defined by the key of the segment.

To add a secondary index:

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, ensure that the code is changed.
2. Unload the database by using the existing DBD and the HD Reorganization Unload utility.

3. Code 2 new DBDs. one for the existing database and one for the new secondary index database.
4. If the change you are making affects the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Optional: Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Delete the old database space and define new database space (non-VSAM), or delete the space allocated for the cluster and define space for the new cluster. In addition, define space for the secondary index.
7. Reload the existing database, using the new DBD and the HD Reorganization Reload utility.
8. Run the Database Prefix Resolution utility, using the DFSURWF1 work file that is output from Step 7 as input.
9. Run the HISAM Reorganization Unload utility, using the DFSURIDX work file that is output from Step 8 as input. Indicate in the utility control statement that HISAM unload is being used to build a secondary index.
10. Run the HISAM Reorganization Reload utility using as input the output from HISAM unload in Step 9.
11. Change your JCL to add a DD statement for the secondary index data set even when you are not using the secondary index to process the main database.
12. Change your reorganization procedures when adding a secondary index.
13. When you reorganize the data set the secondary index points to, you must execute the reorganization utilities to rebuild the secondary index.

Related concepts:

Related reference:

Examples with secondary indexes (System Utilities)

Adding a secondary index to a new primary DEDB

You can add a secondary index to a new primary DEDB database to process a segment type in a sequence other than the one defined by the segment's key.

To add a secondary index:

1. Create a DBD definition for a DEDB primary database that has a secondary index defined.
2. Create a DBD definition for a Fast Path secondary index database.
3. Add the PROCSEQD= parameter with a Fast Path secondary index DBD name in the PCB for the primary DEDB database that will be accessed through the Fast Path secondary index database.
4. Run the DBDGEN utility with the definitions with Fast Path secondary index defined.
5. Run the PSBGEN utility with the definitions with Fast Path secondary index defined.
6. Run the ACB Maintenance utility with the definitions with Fast Path secondary index defined.
7. Allocate a new data set for the area.

8. Optional: Register new DB/DBDS/ADS for the new DEDB database to DBRC.

9. Format the new area data set using DEDB Initialization utility (DBFUMIN0).

10. Create a DMB directory entry (DDIR) for the new DEDB and its secondary index database using one of the following methods:
    - Perform a MODBLKS system definition and use the ACBLIB online change process.
    - If you are using dynamic resource definition, issue the CREATE DB command and then perform an ACBLIB or member online change.

11. Take an image copy of the new primary DEDB database.

12. Optional: For data-sharing Fast Path secondary index databases, register the new Fast Path secondary index databases in DBRC.

13. Create an application program to load the primary DEDB database with a Fast Path secondary index defined.

14. Run the application program to build and load both the primary DEDB database and its Fast Path secondary index database.

Both the primary DEDB database and its Fast Path secondary index database are available for IMS online access.

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

Related reference:

➡ Examples with secondary indexes (System Utilities)

Adding a secondary index to a DEDB

You can add a secondary index to an existing DEDB database to process a segment type in a sequence other than the one defined by the segment's key.

1. Modify the DBD definition for a DEDB primary database to add secondary index information.

2. Create a DBD definition for a Fast Path secondary index database.

3. Add the PROCSEQD= parameter with a Fast Path secondary index DBD name in the PCB for the primary DEDB database that will be accessed through the Fast Path secondary index database.

4. Run the DBDGEN utility with the definitions with Fast Path secondary index defined.

5. Run the PSBGEN utility with the definitions with Fast Path secondary index defined.

6. Run the ACBGEN utility with the definitions with Fast Path secondary index defined.

7. For data-sharing Fast Path secondary index databases, register the new Fast Path secondary index databases in DBRC.

8. Change the access of the primary DEDB database to "Read Only" or take the primary DEDB database offline in preparation to build its Fast Path secondary index database.

9. Build and load a Fast Path secondary index database. Use a tool that is offered by IMS Tools vendors.

10. If you have not already done so, take the primary DEDB database offline before performing an online change.

11. Create a DMB directory entry (DDIR) for the new Fast Path secondary index database. You can create the DDIR by one of the following methods:

- Perform a MODBLKS system definition and use the ACBLIB online change process.
- If you are using dynamic resource definition, issue the CREATE DB command to create the DDIR for the new Fast Path secondary index database and then perform an online change (ACBLIB online change or member online change).

12. Start the primary DEDB database and its Fast Path secondary index database.

Both the primary DEDB database and its Fast Path secondary index database are available for IMS online access.

Related concepts:

"How secondary indexes restructure the hierarchy of DEDB databases" on page 344

Chapter 18, "Creating secondary indexes," on page 335

Related reference:

Examples with secondary indexes (System Utilities)

Dropping an index

You can remove a secondary index from a DEDB database if it is no longer necessary.

To remove a secondary index:
1. Remove the LCHILD and XDFLD statements in the DBD for the primary DEDB database.
2. Remove or change the FIELD statement that specifies a /CK name.
3. Delete the DBD definition for its associated Fast Path secondary index database.
4. Delete any PSB definitions that have the PROCSEQD= parameter for the Fast Path secondary index database.
5. Run the DBDGEN utility with the new definitions without the Fast Path secondary index defined.
6. Run the PSBGEN utility with the new definitions without the Fast Path secondary index defined.
7. Run the ACBGEN utility with the new definitions without the Fast Path secondary index defined.
8. Take offline the primary DEDB database and its Fast Path secondary index database.
9. Perform an online change to revert changes in the primary DEDB database definition without secondary index defined.
10. Start the database on the primary DEDB database to make the database available for IMS online access.

Related concepts:

Chapter 18, "Creating secondary indexes," on page 335

Changing the number of data set groups

Normally, a database is physically stored on one data set or, as in HISAM, on a pair of data sets. However, databases can be physically stored on more than one data set or pair of data sets. If so, each data set or pair of data sets is called a data set group.

You can change to multiple data set groups to tune your database. It is not possible for you to specifically monitor your database to determine whether multiple data set groups will improve performance or better utilize space. Rather, knowledge of your application's requirements along with many types of statistics about database use might help you make this decision.

You can use the database reorganization/load processing utilities (that is, the HISAM Unload/Reload, HD Unload/Reload, Prefix Resolution and Prefix Update utilities) when you change to multiple data set groups and you can use the utilities on one or more databases concurrently. For example, you can reorganize one or more existing databases at the same time that other databases are being initially loaded. Any or all of the databases being operated on can be logically interrelated. A database operation is defined as an initial database load, a database unload/reload (reorganization), or a database scan.

If you are making additional structural changes to a HISAM database other than introducing multiple data set groups (for example, changing the access method from HISAM to HDAM, pointer changes, additional segments, or adding a secondary index), you must follow a procedure similar to that shown in "Example flow for HD databases with logical relationships or secondary indexes" on page 753.

To change the number of data set groups in your database:
1. Unload your database using the existing DBD.
2. If your database is PHDAM or PHIDAM, delete the database definition from the DBRC RECON data sets using the HALDB Partition Definition Utility.
3. Code a new DBD.
4. Recalculate database space. You need to recalculate database space because the change you are making will result in different requirements for database space.
5. Delete the old database space and define new database space for non-VSAM data sets. Delete the space allocated for the old clusters and define space for the new clusters for VSAM data sets.
6. If your new database is PHDAM or PHIDAM, run the HALDB Partition Definition utility to define the partition data sets for the database.
7. Reallocate data sets because the number and size of data sets you are using will change.
8. Reload your database using the new DBD. Take an image copy of your database as soon as the database is reloaded.
9. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.

Related concepts:
"Multiple data set groups" on page 410

Allocating OSAM data sets (System Definition)

Related tasks:
"Estimating the minimum size of the database" on page 559
"Allocating database data sets" on page 567
"Offline reorganization by using the reorganization utilities" on page 646

Example flow for simple HD databases

The process flow for making changes to simple HD databases by using the reorganization utilities is simple compared to the process flow for HD databases that use logical relationships or secondary indexes.

The following figure illustrates the process flow for modifying a database to use multiple data set groups.
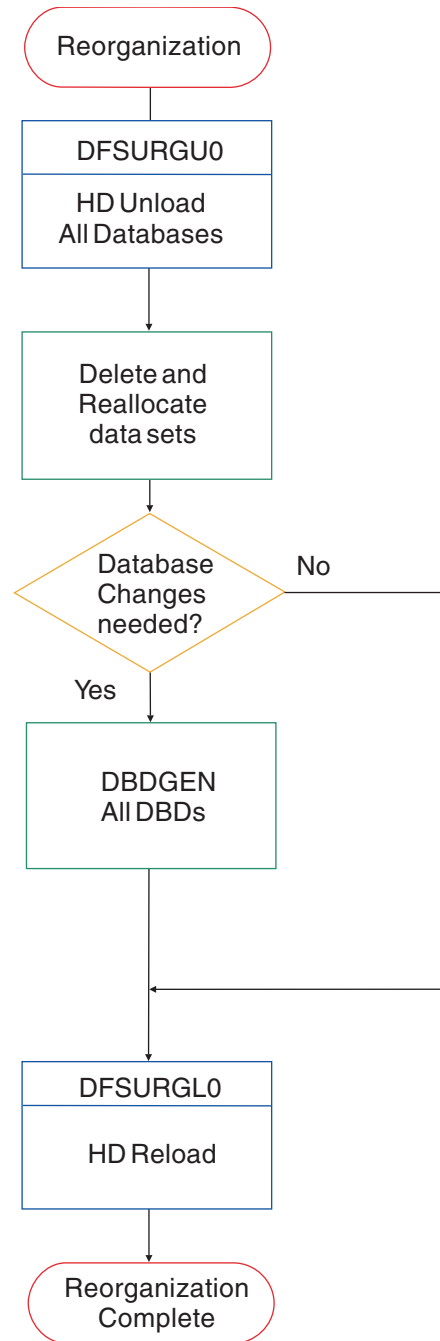
Figure 286. Flow chart of process for modifying a database to use multiple data set groups

Example flow for modifying HISAM databases with the reorganization utilities

The following flow chart illustrates the process for using the reorganization utilities to modify a database to use multiple data set groups.
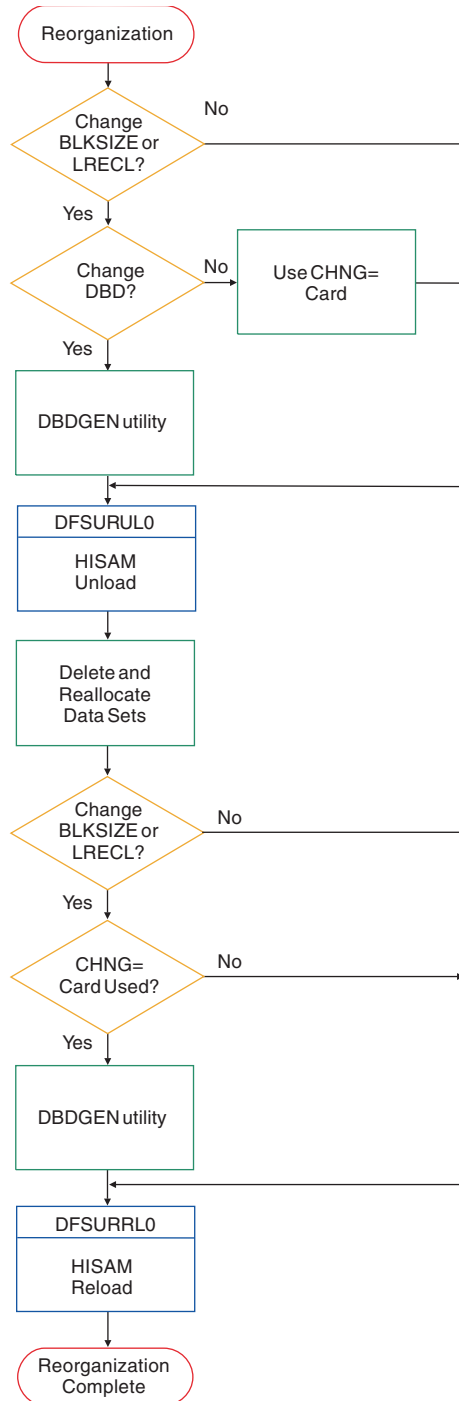
Figure 287. Flow chart of process for modifying a HISAM database to use multiple data set groups

Example flow for HD databases with logical relationships or secondary indexes

If your HD database has logical relationships or secondary indexes or if you are making additional structural changes to your database when adding multiple data set groups, you will follow a process similar to that shown in the following figure.

The following figure illustrates the process flow for modifying a database that uses logical relationships or secondary indexes to use multiple data set groups.
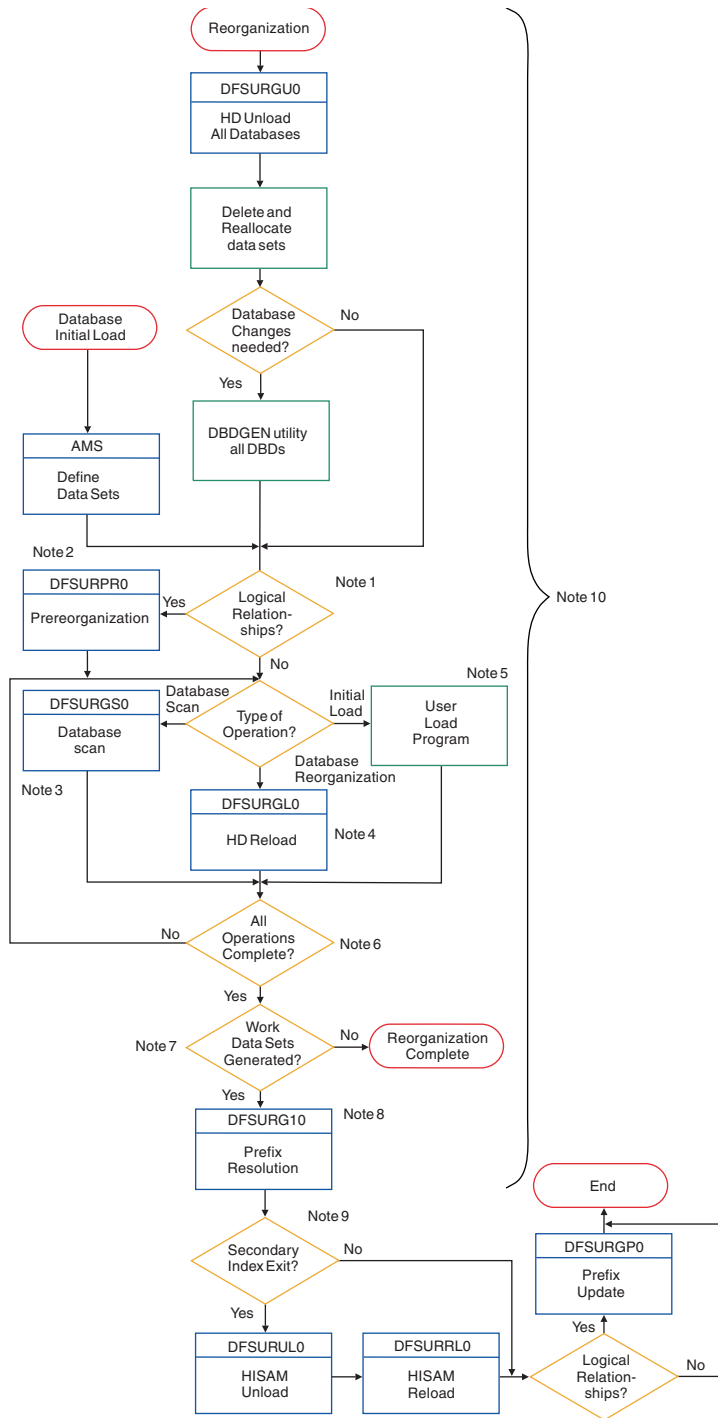
Figure 288. Flow chart of process for modifying a database with logical relationships or secondary indexes to use multiple data set groups

Notes to the preceding flow chart:

1. If one or more segments in any or all of the databases being operated upon is involved in either a logical relationship or a secondary index relationship, the YES branch must be taken. You can also use the Prereorganization utility to determine which database operations must be performed.

2. Based upon the information given to it on control statements, the database Prereorganization utility provides a list of databases that must be initially

loaded, reorganized, or scanned. You must not change the number and sequence of databases specified on the prereorganization control statement between reload and prefix resolution.

3. You must run the DB Scan utility before a database is unloaded when logical parent concatenated keys are defined as virtual in the logical child database to be unloaded.

   This program should be executed against each database listed in the output of the Prereorganization utility. A work data set can be generated for each database scanned by this utility. Databases for scanning are listed after the characters "DBS=" in one or more output messages of the Prereorganization utility.

4. The HD Reorganization Reload utility can cause the generation of a work data set to be later used by the Database Prefix Resolution utility. Databases to be reorganized using the HD Reorganization Unload utility and the HD Reorganization Reload utility are listed after the character "DBR=" in one or more output messages of the Prereorganization utility.

5. The user-provided initial database load program can automatically cause the generation of a work data set to be later used by the Prefix Resolution utility. You do not need to add code to the initial load program for work data set generation. Code is added automatically by IMS through the user program issuing ISRT requests. You must, however, provide a DD statement for this data set along with the other JCL statements necessary to execute the initial load program. Databases for initial loading are listed after the characters DBIL= in one or more output messages of the Prereorganization utility.

6. You must ensure that all operations indicated by the Prereorganization utility (if it was executed) are completed prior to taking the YES branch.

7. If any work data sets were generated during any of the database operations that were executed by you, the YES branch must be taken. The presence of a logical relationship in a database does not guarantee that work data sets will be generated during a database operation. The reorganization/load processing utilities determine the need for work data sets dynamically, based upon the actual segments presented during a database operation. If any segments that participate in a logical relationship are loaded, work data sets will be generated and the YES branch must be taken.

   If for any specific database operation no work data set was generated for the database, processing of that database is complete and ready to use.

   When a HIDAM database is initially loaded or reorganized, its primary index will be generated at database load time.

8. The Database Prefix Resolution utility combines the work file output from the Database Scan utility and either the HD Reorganization Reload utility or the user's initial database load execution, but not both, to create an output data set, DFSURWF3, for use by the Prefix Update utility. The Prefix Update utility then completes all logical relationships defined for the databases that were operated upon.

9. If a secondary index needs to be created or if two secondary indexes need to be combined, you must run the HISAM Unload/Reload utilities. After the HISAM Unload/Reload utilities are run, if logical relationships exist in the database, you must execute the Prefix Update utility before the reorganization or load process is considered to be complete.

10. This area of the flowchart must be followed once for each database to be operated upon, whether the operation consists of an initial load, reorganization, or scan. The operations can be done for all databases concurrently, or one database at a time. If the various database operations are

performed sequentially, work data set storage space can be saved and processing efficiency increased if DISP=(MOD,KEEP) is specified for the DFSURWF1 DD statement associated with each database operation. The attributes of the work data set for the database initial load, reorganization, and scan programs must be identical.

When using the HD Reorganization Reload utility, first do all unloads and scans of logically related databases if logical parent concatenated keys are defined as virtual in the logical child.

Converting to the Segment Edit/Compression exit routine

You might need to make changes to your database before you can use a Segment Edit/Compression exit routine, such as the IMS sample exit routine DFSCMPX0.

To convert an existing database to support a Segment Edit/Compression exit routine, follow these steps:

1. Determine whether the change you are making affects the code in any application programs. If the code is affected, make sure it gets changed.
2. Unload your database, using the existing DBD and the HD Unload utility.
3. Code a new DBD. The new DBD must specify the name of your edit routine for the segment types you need edited.
4. If the change you are making affected the code in application programs, make any necessary changes to the PSBs for those application programs. If you have the DB/DC Data Dictionary, it can help you determine which application programs and PCBs are affected by the DBD changes you have made.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. Recalculate database space. You need to do this because the change you are making results in different requirements for database space.
7. Delete the old database space and define new database space. If you are using VSAM, use the Access Method Services DEFINE CLUSTER command to define VSAM data sets.
8. Reload the database, using the new DBD. Remember to make an image copy of your database as soon as it is reloaded.
9. If your database uses logical relationships or secondary indexes, you must run reorganization utilities before and after reloading to resolve prefix information.

Related concepts:

"Segment Edit/Compression exit routine" on page 390

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Related reference:

⬉ Segment edit/compression exit routines (Exit Routines)

Converting databases for Data Capture exit routines and Asynchronous Data Capture

You can convert a database for a data capture exit routine or asynchronous data capture in for basic steps.

This topic contains General-use Programming Interface information.

To convert an existing database for use with Data Capture exit routines or Asynchronous Data Capture:

1. Determine whether the change requires revisions to the logical delete rules in a database. If so, change the delete rules, which might require reorganizing your database.
2. Code a new DBD. On the DBD or SEGM statements, specify the name of each exit routine you need called against a segment in the database.
3. Run DBDGEN.
4. If you use prebuilt ACBs rather than dynamically built ACBs, rebuild the ACB.

Related concepts:

"Data Capture exit routines" on page 393

Related reference:

➡ Database Description (DBD) Generation utility (System Utilities)

➡ Database Manager exit routines (Exit Routines)

Modifying online databases

IMS provides several methods for modifying IMS databases without taking them offline, including dynamic resource definition support, which uses type-2 commands, the online change function, which uses the type-1 commands, HALDB and DEDB alter functions, and when the IMS management of ACBs is enabled, DDL or the IMPORT DEFN SOURCE(CATALOG) command.

Each of the methods has different prerequisites and steps, and each of the methods supports different types of database modifications.

For example, the online change function requires a MODBLKS system definition and dynamic resource definition does not. To modify a database by using DDL statements, the IMS management of ACBs must be enabled. You start the HALDB Alter function by issuing the INIT OLREORG command, but you start the DEDB Alter function by running the DEDB Alter utility (DBFUDA00).

Related tasks:

"Introducing databases into online systems" on page 547

Altering the definition of an online HALDB database

By specifying the ALTER option of the INITIATE OLREORG command, you can apply certain segment or field definition changes to an online HALDB database without stopping access to the database.

During the reorganization process, application programs that use the existing database structure can continue to access the database. Application programs that use the altered database structure cannot access the database until the reorganization process is complete and you perform one of the following actions to enable access to the new segment and field definitions:

- If you use ACB libraries, the online change function.
- If the IMS management of ACBs is enabled and you use the IMS catalog, issue the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command.

The ALTER option is currently supported by the following database access types:

- PHDAM
- PHIDAM

You can use the ALTER option to make only the following changes to a database definition:

- An increase to the length of a segment.
- The addition of new fields to existing, undefined space at the end of a segment.

When the ALTER option is specified, the database is altered by the integrated HALDB online reorganization function. Therefore, unless otherwise noted, the procedures, guidelines, requirements, and so forth that are documented for the integrated HALDB online reorganization function apply when altering a HALDB database.

Related concepts:

"HALDB online reorganization" on page 669

Configuration requirements for altering an online HALDB database

Before you can alter an online HALDB database, your IMS system must meet a few configuration requirements.

The IMS system must be configured with the Operations Manager (OM) and Structure Call Interface (SCI) components of the IMS Common Service Layer (CSL) to enable a type-2 command environment. A HALDB alter operation can be initiated only by the type-2 command INITIATE OLREORG NAME(*haldbmaster*) OPTION(ALTER).

The MINVERS value of the RECON data set must be set to a version of IMS that supports altering an online HALDB database. Consequently, a RECON data set that supports an IMS system that uses the HALDB alter function cannot be shared by versions of IMS that do not support the HALDB alter function.

Steps for altering an online HALDB database

The steps required for altering the structure of an online HALDB database differ depending on whether your system uses IMS-managed ACBs or an ACB library data set.

Altering an online HALDB database when IMS manages ACBs:

Altering the structure of an online HALDB database when IMS manages ACBs requires entering the changes into the staging data set of the IMS catalog, issuing the INITIATE OLREORG OPTION(ALTER) command to apply the changes to the database, and then issuing the IMPORT DEFN SOURCE(CATALOG) command to activate the database alterations in the online system.

1. Enter the database alterations into the IMS catalog either by submitting SQL DDL statements to IMS or by running one of the IMS catalog population utilities with the `MANAGEDACBS=STAGED` control statement referenced by the SYSINP DD statement. To use one of the IMS catalog population utilities, you must also use the DBD Generation utility, the PSB Generation utility, and, unless you use the ACB Generation and Catalog Population utility (DFS3UACB), the ACB Maintenance utility.
2. If necessary, set the ALTERSZE values in the RECON data set to alter the OSAM block or VSAM CI sizes of the output data sets.
3. Issue the IMS type-2 INITIATE OLREORG OPTION(ALTER) command to apply the changes to the online database. After the command is issued,

application programs that use the old database structure can continue to access the database until the IMPORT DEFN SOURCE(CATALOG) command is issued.

4. After alter processing is complete, activate the database changes in the IMS system by issuing the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command.

Related concepts:

➡ IMS management of ACBs (System Definition)

➡ SQL programming reference (Application Programming APIs)

Related tasks:

"Modifying block or CI sizes with HALDB alter" on page 762

Related reference:

➡ IMPORT DEFN SOURCE(CATALOG) command (Commands)

➡ INITIATE OLREORG command (Commands)

➡ CHANGE.PART command (Commands)

Altering an online HALDB database when ACB libraries are used:

Altering the structure of an online HALDB database in an IMS system that uses an ACB library requires coding the changes to the database and program view definitions in utility macro instructions, performing DBD and ACB generations, issuing the INIT OLREORG OPTION(ALTER) command, and performing the Online Change (OLC) procedure.

The HALDB alter function uses dynamic allocation to allocate the staging ACB library. If dynamic allocation of the staging ACB library is not already enabled, you can enable it by using the DFSMDA macro and the IMSDALOC procedure to build the dynamic allocation member named IMSACB.

The high-level steps required to apply database changes with the ALTER option include:

1. Code the changes to the database structure in the SEGM and, if necessary, the FIELD input statements of the Database Generation (DBDGEN) utility.

2. Generate the DBD members by running the DBDGEN utility with the changed macros.

3. Generate the PSB members for the application programs that access the database. If database versioning is enabled, you are not required to update the PSBs of the application programs that access only prior versions of the database.

4. Generate the ACB members into the ACB staging library by using one of the following methods:
   - If the IMS catalog is enabled in your system, run the ACB Generation and Catalog Populate utility (DFS3UACB) to generate the ACB members and populate the IMS catalog in a single job step.
   - If the IMS catalog is not enabled in your system, run the ACB Maintenance utility.

5. If necessary, set the ALTERSZE values in the RECON data set to alter the OSAM block or VSAM CI sizes of the output data sets.

6. Issue the IMS type-2 INITIATE OLREORG OPTION(ALTER) command to apply the changes to the online database. After the command is issued,

application programs that use the old database structure can continue to access the database until online change is performed.

7. Enable access to the new database structure by using the member online change procedure. A HALDB online alter operation is not complete until the ACB members for the altered database that are in the staging library are activated.

Related concepts:

"Coding database descriptions as input for the DBDGEN utility" on page 519

"Building the application control blocks (ACBGEN)" on page 528

➡️ Dynamically allocating the ACB staging library for ACBLIB member online change (System Definition)

Related tasks:

"Activating database changes by using the online change function" on page 795

"Modifying block or CI sizes with HALDB alter" on page 762

Related reference:

➡️ IMSDALOC procedure (System Definition)

➡️ DFSMDA macro (System Definition)

➡️ INITIATE OLREORG command (Commands)

➡️ INITIATE OLC command (Commands)

➡️ CHANGE.PART command (Commands)

➡️ Database Description (DBD) Generation utility (System Utilities)

➡️ Application Control Blocks Maintenance utility (System Utilities)

ACB library and HALDB alter

If you use ACB libraries, before you start an alter operation you must generate the ACB members that define the new structure of the database into the staging ACB library. If you enable the IMS management of ACBs, you do not need to use staging and active ACB libraries to alter the structure of a HALDB database.

If you have not enabled the IMS management of ACBs and you use ACB libraries, the ACB members must be maintained in the staging library until the online change function clears the ALTER IN PROGRESS status from the database partition records in the RECON data set.

The staging ACB library must be available to all IMS systems that share the database and any batch application programs or utilities that access the database. You can use a single, shared staging ACB library or multiple staging ACB libraries with cloned copies of the ACB members that contain the database changes. You must enable dynamic allocation and deallocation of the staging ACB library by defining TYPE=IMSACB DFSMDA macros.

If the ACB members for the altered database are modified or deleted from the staging library while the partition records have an ALTER IN PROGRESS status, application programs and utilities cannot access the database and the online change function cannot activate the new database structure. If the ACB members are modified or deleted, you can restore access to the database and complete online change by restoring the staging ACB library to its state at the time that alter processing began.

After all of the partition data sets are altered, the ACB members in the staging library must be added to the active ACB library to enable access to the new database structure.

To add the ACB members to the active ACB library, use the ACB member online change function. The member online change function copies from the staging ACB library to the active ACB library only the ACB members of the altered database.

If necessary, you can use either the local online change function or the global online change function. However, both of these methods require you to copy the ACB members from the staging library to the inactive ACB library before you start the online change process. These methods also replace the entire ACB library instead of only the altered ACB members.

Unless database versioning is enabled to allow certain application programs to continue accessing the database without modifications, you must also activate application programs that reference the altered segments in the database when you activate the new ACB members.

After the ACB members are activated, the staging library is no longer required.

Related concepts:

⇨ ACB library member online change (System Administration)

⇨ Dynamically allocating the ACB staging library for ACBLIB member online change (System Definition)

Related tasks:

⇨ Changing or adding IMS.ACBLIB members online (System Administration)

IMS catalog and HALDB alter

When you alter a HALDB database in an IMS-managed ACBs environment, IMS reads the ACB members that are in the staging data set of the IMS catalog. The ACB members define the new structure of the database. These ACB members must be maintained in the staging data set of the IMS catalog until the ALTER IN PROGRESS status is cleared from the database partition records in the RECON data set.

If the ACB members for the altered database are modified or deleted from the IMS catalog while the partition records have an ALTER IN PROGRESS status, application programs and utilities cannot access the database and IMS cannot activate the new database structure when you issue the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command. If the ACB members are modified or deleted, you can restore access to the database and activate the database changes by restoring the staging data set of the IMS catalog to its state at the time that alter processing began.

Unless database versioning is enabled to allow certain application programs to continue accessing the database without modifications, you must also activate application programs that reference the altered segments in the database when you activate the new ACB members.

Online change and HALDB alter

If you use ACB libraries, the IMS online change function is required to enable access to the new structure of a HALDB database after alter processing completes.

If you are altering a HALDB database in an IMS-managed ACBs environment, instead of using the online change function, you must issue the IMPORT DEFN SOURCE(CATALOG) command after alter processing completes to enable access to the new structure of the database.

In addition to activating the ACB members that contain the new database structure, the online change function clears various flags and counters in the RECON data set. Activating the ACB members by a means other than the online change function does not clear the flags and counters automatically.

Until the ACB members are activated and the flags and counters are cleared, the alter procedure is not complete and the new database structure cannot be used.

Before you start the online change procedure to complete an alter operation, you must stop access to the HALDB database by issuing either a /DBRECOVERY DB *HALDB_master_name* command or an UPDATE DB NAME(*HALDB_master_name*) STOP(ACCESS) command.

If you use ACB libraries, use the member online change function to complete the alter procedure. The member online change function reads directly from the staging ACB library and can process only the specific ACB members that require activation. The local and global online change functions require you to copy the ACB members into the inactive ACB library. They also process the entire ACB library, instead of just the ACB members that contain the new database changes.

In a data sharing environment that uses ACB libraries, if you use local online change you must coordinate it across all IMS systems that share the database. Because the first IMS system in which the local online change is performed resets all of the flags and counters in the RECON data set, you should issue the /DBRECOVER command for the database in all IMS systems first and then do the local online change in each system.

Related concepts:

"Online change and HALDB databases" on page 819

➥ ACB library member online change (System Administration)

Related tasks:

"Activating database changes by using the online change function" on page 795

➥ Changing or adding IMS.ACBLIB members online (System Administration)

Related reference:

➥ INITIATE OLC command (Commands)

Modifying block or CI sizes with HALDB alter

If you are increasing the size of a segment when you are altering an online HALDB database, you might also need to increase the OSAM block size or VSAM CI size of the output database data set that holds the altered segment.

Note: The INITIATE OLREORG command fails if the ALTER option is specified, but no changes to the database structure are detected in the input database member. Consequently, you cannot change the block or CI sizes by using the ALTER option unless you are also making a structural change to the database.

New block or CI sizes are applied to the output data sets at the start of alter processing, but must be stored in the RECON data set before the INIT OLREORG OPTION(ALTER) command is issued.

New block or CI sizes are stored in the RECON by specifying them on the ALTERSZE keyword of the CHANGE.PART command.

For VSAM data sets, if output data sets for alter processing exist, the output data sets that require a new CI size must be deleted before initiating the alter process. Alter processing automatically re-creates the required output data sets with the new CI sizes, unless you create them yourself. If you create the output data sets yourself, the CI sizes must match their corresponding ALTERSZE values. If no ALTERSZE value is specified for a given VSAM data set group and an output data set exists, the CI size of the output data set is used. If no ALTERSZE value is specified and an output data set does not exist, the CI size of the input data set is used.

For OSAM data sets, if no ALTERSZE value is set, the block size of the input data set is used, even if an output data set exists.

When you change a block or CI size, you might also need to change the size of the buffers. If the new block or CI size does not fit into the current buffer subpool, IMS tries to find a larger subpool among the available subpools. If none of the available subpools are large enough to hold the new block or CI size, the output data set fails to open. To check buffer sizes, issue the type-2 command QUERY POOL TYPE(DBAS).

Related concepts:

"Adjusting buffers" on page 701

Related tasks:

"Steps for altering an online HALDB database" on page 758

Related reference:

➡ CHANGE.PART command (Commands)

➡ LIST.DB command (Commands)

➡ QUERY POOL command (Commands)

Setting ALTERSZE values for HALDB alter:

To increase the OSAM block size or VSAM CI size of the database data sets when you are modifying the structure of a database with the HALDB alter function, you must set ALTERSZE values for each data set group that is changing in each partition record in the RECON data set.

To set the ALTERSZE values, you can use either DBRC command CHANGE.PART or the HALDB Partition Definition utility (%DFSHALDB).

If you use the CHANGE.PART command to set the ALTERSZE values, the values must be specified as positional, comma-separated values. The value in the first position applies to the first data set group. The value in the second position applies to the second data set group, and so on.

For example, the following ALTERSZE specification sets a new block or CI size for the third data set group, but leaves the sizes unchanged for the first and second data set groups, as well as for the fourth through tenth data set groups, if they exist: ALTERSZE(,,4096).

You can determine the position in which to enter a size for a data set group by looking at the DSGROUP keyword in the SEGM statement that defines the

segment that you are altering. DSGROUP=A indicates the first position, DSGROUP=B, the second position, and so on, up to DSGROUP=J, which indicates the tenth position.

To set ALTERSZE values for the output database data sets of an alter operation by using batch DBRC commands, specify the following command.

```
                                          ┌─,─────┐
►►──CHANGE.PART──DBD(name)──PART(name)──ALTERSZE(─▼─nnnnn─┴──)───────────────►◄
```

After the command is successfully processed, the block or CI sizes to be used by the alter process are listed under "ALTER BLOCK SIZE" in the RECON record for a partition, which can be displayed by issuing the DBRC command LIST.DB DBD(*partitionname*).

After the alter size values are set, you can start the alter process by issuing the IMS type-2 command INIT OLREORG NAME(*masterdb*) OPTION(ALTER).

Related reference:

➡ CHANGE.PART command (Commands)

➡ LIST.DB command (Commands)

Changing ALTERSZE values after they are set:

To correct an ALTERSZE value that is already set, replace the incorrect value with the correct value by using either the CHANGE.PART command or the HALDB Partition Definition utility (%DFSHALDB).

For OSAM data sets, if you change an ALTERSZE value back to the original block size of the input data set, the ALTERSZE value displays as 0 to indicate that the block size is not changing. If all of the ALTERSZE values are restored to the original block sizes of the input data sets, the **ALTER SIZE** field is omitted from the output of the LIST.DB command.

For VSAM data sets, if you change an ALTERSZE value back to the original CI size, the original CI size is displayed. If all of the ALTERSZE values are restored to the original CI sizes of the input data sets, the **ALTER SIZE** field is displayed with the last values that you entered.

You can clear all of the ALTERSZE values for a partition by specifying CHANGE.PART PART(name) NOALTRSZ command. For both OSAM and VSAM data sets, when the NOALTRSZ keyword is used to clear all ALTERSZE values, the **ALTER SIZE** field is omitted when the partition record is displayed.

After the command is successfully processed, the block or CI sizes to be used by the alter process are listed under **ALTER BLOCK SIZE** in the RECON record for a partition, which can be displayed by the issuing DBRC command LIST.DB DBD(*partitionname*).

After the alter size values are corrected, you can start the alter process by issuing the IMS type-2 command INIT OLREORG NAME(*masterdb*) OPTION(ALTER).

Related reference:

➡ CHANGE.PART command (Commands)

➡ LIST.DB command (Commands)

Online reorganization processing when the ALTER option is specified

When the ALTER option is specified, the INIT OLREORG command initiates a reorganization of an entire HALDB database to apply the database changes to all of the database partitions.

Upon receiving the INIT OLREORG OPTION(ALTER) command, an IMS system can alter a maximum of 10 partitions concurrently. If there are more than 10 partitions, those that are processed immediately are queued and altered as soon as processing completes for one of the partitions that were altered first.

While an IMS system reorganizes and alters a partition, the IMS system has ownership of the partition. The subsystem ID of the IMS system that owns a partition for alter processing is recorded in the OLRIMSID field of the partition record.

In a data-sharing environment, ownership of a partition is granted to the first IMS system that is available to alter the partition. If one IMS system is available to process ten partitions before any other IMS system becomes available, all ten partitions are processed by the single IMS system. If a partition is queued for alter processing, the partition will be processed by the first IMS system that can obtain ownership of the partition.

When alter processing starts, IMS updates the RECON records for every partition in the database with ALTER IN PROGRESS=YES, including for those partitions that are queued. While an IMS system is actively altering a partition, the partition record shows OLREORG CURSOR ACTIVE=YES.

After alter processing is complete for a partition, the partition record to shows PARTITION ALTERED=YES.

Only after all partitions in the database have a status of PARTITION ALTERED=YES can you activate the new database structure. You activate the new database structure by using one of the following methods:

- If the IMS management of ACBs is enabled, by issuing the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command.
- If you use ACB libraries, by using the online change function.

When you use the online change function or the IMPORT DEFN SOURCE(CATALOG) command, both the ALTER IN PROGRESS field and the PARTITION ALTERED fields are reset to NO.

Related concepts:

"HALDB online reorganization" on page 669

Related reference:

➡ INITIATE OLREORG command (Commands)

➡ DB (HALDB) record fields (Commands)

➡ DB (PART) record fields (Commands)

➡ LIST.DB command (Commands)

HALDB alter and offline reorganization

After starting HALDB alter on a database, if the alter operation is terminated, you can complete the application of the DBD changes to the database by using the offline reorganization process.

The method that you use to reorganize a HALDB database offline depends on whether the IMS management of ACBs is enabled.

Related tasks:

"HALDB offline reorganization" on page 664

"Offline Reorganizations after HALDB Online Reorganizations" on page 690

Related reference:

➡ HD Reorganization Unload utility (DFSURGU0) (Database Utilities)

➡ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

HALDB alter and offline reorganization when IMS manages ACBs:

If you perform the offline reorganization process to alter a HALDB database when IMS manages ACBs, the HD Reorganization Unload utility (DFSURGU0) and the HD Reorganization Reload utility (DFSURGL0) retrieve the active DBDs from the directory data set of the IMS catalog. You must also issue the IMPORT DEFN SOURCE(CATALOG) command.

Before you start the offline reorganization, access to the database must be stopped by issuing either the IMS type-2 command UPDATE DB NAME(*partition_name*) STOP(ACCESS) command or the IMS type-1 command /DBRECOVERY.

An offline reorganization is performed by unloading the database with the HD Reorganization Unload utility (DFSURGU0) and then reloading the database with the HD Reorganization Reload utility (DFSURGL0).

The HD Reorganization Unload utility automatically unloads the records from the active portions of both the A–J and M–V data sets.

The HD Reorganization Unload utility automatically retrieves the active database records from the directory data set of the IMS catalog. When you use the HD Reorganization Reload utility to reload the database with the changed definition, by default, the reload utility obtains the active database from the directory data set of the IMS catalog. The altered database is in the staging data set of the IMS catalog. You can specify the STAGING keyword on the DFSACBPD DD statement in the execution JCL to obtain the pending database from the staging data sets.

The IMS DD statement in the execution JCL of both the HD Reorganization Unload utility and the HD Reorganization Reload utility becomes optional and is ignored by the IMS system.

The HD Reorganization Reload utility loads all records into the A–J data sets. If the HALDB alter function was writing to the M–V data sets, the characteristics of the A–J data sets might not be correct. Before you reload the database, ensure that the A-J data sets are allocated enough storage for the altered segment sizes and, if you are altering VSAM data sets, that the VSAM CI sizes of the A–J data sets are correct.

After the database is reloaded, issue the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command to enable the altered database in the online system and to reset the HALDB alter flags and counters in the RECON data set.

After you issue the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command, you can restart the database.

HALDB alter and offline when ACB libraries are used:

If you perform the offline reorganization process to alter a HALDB database ACB libraries are used, the HD Reorganization Unload utility (DFSURGU0) and the HD Reorganization Reload utility (DFSURGL0) retrieve the active DBDs from the DBDLIB member that is specified on the IMS DD statement. You must also use the online change function.

Before you start the offline reorganization, access to the database must be stopped by issuing either the IMS type-2 command UPDATE DB NAME(*partition_name*) STOP(ACCESS) command or the IMS type-1 command /DBRECOVERY.

An offline reorganization is performed by unloading the database with the HD Reorganization Unload utility (DFSURGU0) and then reloading the database with the HD Reorganization Reload utility (DFSURGL0).

The HD Reorganization Unload utility automatically unloads the records from the active portions of both the A–J and M–V data sets. The IMS DD statement in the execution JCL of the HD Reorganization Unload utility must reference the DBD library that contains the original, unaltered DBD of the database, regardless of whether some or all of the database segments have been altered by the HALDB alter function.

The execution JCL for the HD Reorganization Reload utility (DFSURGL0) also must contain an IMS DD statement that references the DBD library that contains original, unaltered DBD of the database.

The altered DBD is in the staging ACB library. If you have dynamic allocation enabled for the staging ACB library, a DD statement for the staging ACB library is not required for either utility.

The HD Reorganization Reload utility loads all records into the A–J data sets. If the HALDB alter function was writing to the M–V data sets, the characteristics of the A–J data sets might not be correct. Before you reload the database, ensure that the A-J data sets are allocated enough storage for the altered segment sizes and, if you are altering VSAM data sets, that the VSAM CI sizes of the A–J data sets are correct.

After the database is reloaded, perform the online change function to enable the altered database in the online system and to reset the HALDB alter flags and counters in the RECON data set.

After online change is performed, you can restart the database.

Altering shared HALDB databases

In a data-sharing environment, alter processing can be distributed among multiple IMS systems by routing the INIT OLREORG OPTION(ALTER) command to each of the participating IMS systems.

The process that is used to alter shared HALDB databases differs depending on whether IMS-managed ACBs or ACB libraries are used.

Altering shared HALDB databases when IMS manages ACBs:

When IMS manages ACBs, all of the IMS systems that participate in the alter operation read the database changes from members in the staging data set of the IMS catalog.

The directory data set of the IMS catalog can be shared or each of the IMS systems can have its own directory data set.

When the alter operation begins, all of the IMS systems in the IMSplex issue message DFS3197I to acknowledge the start of alter processing, even if the IMS systems do not access the database that is being altered.

During the alter operation, no IMS system can initiate either another alter operation of the database that is being altered or a HALDB online reorganization of the database that is being altered.

Each IMS system issues message DFS3198I when it stops altering the partitions that it started. In a data sharing environment, IMS systems can issue one of two versions of DFS3198I: DFS3198I HALDB ALTER ENDED... or DFS3198I HALDB ALTER COMPLETED.... An IMS system issues DFS3198I HALDB ALTER ENDED... if either alter processing was interrupted or one or more of the partitions in the HALDB database were altered by another IMS system. The other IMS systems might or might not have finished altering the partitions that they started. An IMS system issues DFS3198I HALDB ALTER COMPLETED... if it altered all of the partitions in the HALDB database itself and alter processing is complete for all of the partitions in the database.

Issue the IMS type-2 IMPORT DEFN SOURCE(CATALOG) command to activate the ACB members that are in the staging data set of the IMS catalog. The ACB members are activated for all of the IMS systems that share the HALDB database.

Related concepts:

➡ Activation when the IMS catalog is shared (System Administration)

Related tasks:

"Altering an online HALDB database when IMS manages ACBs" on page 758

Altering shared HALDB databases when ACB libraries are used:

When ACB libraries are used, all of the IMS systems that participate in the alter operation read the database changes from members in the staging ACB library.

The staging ACB library can be shared or each of the IMS systems can have its own staging ACB library. However, if the staging ACB library is not shared, the ACB members that are used by each IMS system must be copies that are created by the same execution of the ACB Maintenance utility.

Recommendation: If possible, have each participating IMS system allocate the same shared staging ACB library.

When the alter operation begins, all of the IMS systems in the IMSplex issue message DFS3197I to acknowledge the start of alter processing, even if the IMS systems do not access the database that is being altered.

During the alter operation, no IMS system can initiate either another alter operation of the database that is being altered or a HALDB online reorganization of the database that is being altered.

Each IMS system issues message DFS3198I when it stops altering the partitions that it started. In a data sharing environment, IMS systems can issue one of two versions of DFS3198I: `DFS3198I HALDB ALTER ENDED...` or `DFS3198I HALDB ALTER COMPLETED...`. An IMS system issues `DFS3198I HALDB ALTER ENDED...` if either alter processing was interrupted or one or more of the partitions in the HALDB database were altered by another IMS system. The other IMS systems might or might not have finished altering the partitions that they started. An IMS system issues `DFS3198I HALDB ALTER COMPLETED...` if it altered all of the partitions in the HALDB database itself and alter processing is complete for all of the partitions in the database.

Use member online change to activate the ACB members in the staging ACB library for all of the IMS systems that share the database.

If you use either global or local online change, after alter processing is complete and before online change is performed, the ACB members that contain the database changes must be copied into the inactive ACB library that is used by each IMS system.

Recommendation: Do not use local online change in a data sharing environment to activate the ACB members after alter processing.

HALDB alter and batch processing

Batch application programs and batch utilities can access a HALDB database while the database is being reorganized by the HALDB alter function.

Related reference:

- ⤷ DBBBATCH procedure (System Definition)

- ⤷ DLIBATCH procedure (System Definition)

- ⤷ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

HALDB alter and batch processing when IMS manages ACBs:

When IMS manages ACBs, batch application programs and batch utilities can access a HALDB database while the database is being reorganized by the HALDB alter function.

Ensure that the directory data set of the IMS catalog is available to any batch application programs or utilities that access the HALDB database that is being altered.

Batch application programs

Batch application programs can access a HALDB database while a HALDB alter operation is in progress.

When IMS manages ACBs, the offline batch application programs read the ACBs from the staging data set of the IMS catalog. Any specification of the IMS DD statement for the DLIBATCH procedure or the IMSACB, IMSACBA, and IMSACBB DD statements for the DBBBATCH procedure are ignored and can be omitted from the JCL.

Batch utilities

Batch utilities can access a HALDB database while a HALDB alter operation is in progress.

To alter a HALDB database offline, use the HD Reorganization Unload utility (DFSURGU0) and the HD Reorganization Reload utility (DFSURGL0). The HD Reorganization Unload utility automatically retrieves the active database records from the directory data set of the IMS catalog. When you use the HD Reorganization Reload utility to reload the database with the changed definition, by default, the reload utility obtains the active database from the directory data set of the IMS catalog. The altered database is in the staging data sets of the IMS catalog. You can specify the STAGING keyword on the DFSACBPD DD statement in the execution JCL to obtain the pending database from the staging data sets.

HALDB alter and batch processing when ACB libraries are used:

When ACB libraries are used, batch application programs and batch utilities can access a HALDB database while the database is being reorganized by the HALDB alter function.

Batch application programs

Batch application programs can access a HALDB database while a HALDB alter operation is in progress without requiring any changes to their existing execution JCL. However, until the online change function completes the alter operation when ACB libraries are used, IMS still requires the original, unaltered database definition for comparison purposes, even if all of the segment data in the database already conforms to the new database definition. Consequently, DLIBATCH application programs must reference the DBD library that contains the original, unaltered DBD on the IMS DD statement in the execution JCL. DBBBATCH application programs must reference the active ACB library that contains the original, unaltered database definition on the IMSACB DD statement. If you have dynamic allocation of the staging ACB library enabled, a DD statement for the staging ACB library that contains the new, altered database definition is not required.

Batch utilities

Batch utilities can access a HALDB database while a HALDB alter operation is in progress. However, until the online change function completes the alter operation when ACB libraries are used, IMS requires the original, unaltered database definition for comparison purposes, regardless of whether some or all of the segment data in the database already conforms to the new database definition.

Consequently, the IMS DD statement in the execution JCL for the utility must reference the DBD library that contains the original, unaltered database definition until online change is performed.

If you have dynamic allocation of the staging ACB library enabled, a DD statement for the staging ACB library that contains the new, altered database definition is not required.

If the HD Reorganization Reload utility is used to complete the reorganization of one or more partitions in a HALDB database for which the HALDB alter function was started, but the online change function has not been performed to complete the alter process, the execution JCL of the HD Reorganization Reload utility must

include an IMS DD statement that references the DBD library that contains the original, unaltered DBD.

Related reference:

➔ DBBBATCH procedure (System Definition)

➔ DLIBATCH procedure (System Definition)

➔ HD Reorganization Reload utility (DFSURGL0) (Database Utilities)

Querying the status of ALTER processing for HALDBs

To determine whether alter processing is complete when altering the structure of an online HALDB database, you can use the IMS type-2 command QUERY OLREORG NAME(*partname* | *) STATUS(ALTER | ALTINPRG | ALTDONE).

You can issue the command for one, some, or all partitions in a HALDB database.

If you specify NAME(*) on the command with STATUS(ALTER), IMS returns the alter status of all of the partitions in the database. If you specify NAME(*) with either STATUS(ALTINPRG) or STATUS(ALTDONE), IMS returns only the partitions in the database with the specified status.

If you specify the name of one or more partitions, IMS returns the alter status of only the specified partitions.

A partition in a HALDB that is being altered can be in the following states:

ALTINPRG
> Indicates that the partition is either currently being altered or is queued for altering.

ALTDONE
> Indicates that alter processing is complete for the partition. The ALTER COMPLETE status of a partition remains in effect until Online Change is performed to activate the ACB members for the new database structure.

ALTINQUE
> Indicates that alter processing has not yet started for the partition.

To query the status of an alter operation on an online HALDB database, issue the type-2 command QUERY OLREORG NAME(*partname* | *) STATUS(ALTER | ALTINPRG | ALTDONE).

Stopping alter processing before completion

You can stop alter processing of a HALDB database before it is complete by issuing the TERMINATE OLREORG command.

You can issue either the type-1 or type-2 version of the TERMINATE OLREORG command; however, only the type-2 version of the command can be issued to multiple IMS systems. The type-1 command is processed by only the local IMS system on which it is issued.

The TERMINATE OLREORG command does not support the specification of the name of a HALDB master database. To stop alter processing for the entire HALDB database, you specify a wildcard character in place of the partition names or you can specify the names of all of the partitions in the database explicitly. If multiple IMS systems are altering the database, you must use the type-2 TERMINATE OLREORG command to stop all of the IMS systems at once or issue the type-1 command separately on each IMS system.

When alter processing is stopped for a subset of the partitions in the database, alter processing continues for the other partitions that are not contained within the specified subset.

When alter processing is stopped, the data in the partition might be physically stored in both the input and output data sets. The output data sets conform to the altered database structure. The input data sets conform to the old database structure. However, where the data is physically stored is not apparent to application programs. Until alter processing completes and online change is performed, application programs can access the data only in the old database structure.

Until all of the partitions in the HALDB database are altered and online change is performed, only application programs that use the unaltered database structure can access the database.

- To stop alter processing for one or more partitions of a HALDB database, issue the TERMINATE OLREORG NAME(*partnm* | *) command.
- To resume alter processing, issue the INITIATE OLREORG OPTION(ALTER) command.

Related concepts:

"The termination phase for HALDB Online Reorganization" on page 673

Related tasks:

"Stopping HALDB Online Reorganization" on page 683

Related reference:

➡ INITIATE OLREORG command (Commands)

➡ TERMINATE OLREORG command (Commands)

Altering a HALDB database that has logical relationships

If a HALDB database is logically related to another database, you can use the HALDB online alter function to modify any segment in the database except for logical parent or logical child segments.

If you need to alter a segment that is a logical parent or logical child, you must take both databases offline so that you can apply the changes to both databases at the same time by using the HD Reorganization Unload utility (DFSURGU0) and the HD Reorganization Reload utility (DFSURGL0). The online alter function for HALDB databases can apply changes to only one database at a time.

Altering a HALDB database that has a secondary index

If a HALDB database is indexed by one or more secondary indexes, you can use the HALDB online alter option to modify the database; however, you cannot use the HALDB online alter option to alter the secondary indexes.

Altering the definition of an online DEDB database with the DEDB Alter utility

You can alter the database or area definitions of DEDB databases that use two-stage randomizers while the DEDB area or the DEDB database remains accessible online with the DEDB Alter utility (DBFUDA00).

If you are to use the DEDB Alter utility, set ALTERHLQ as the default in the Fast Path section of the DFSDFxxx member of the PROCLIB data set.

If more than one IMS is involved, you must set ALTERGRP in the Fast Path section of the DFSDFxxx member of the PROCLIB data set.

If you enable ACB libraries by specifying ACBMGMT=ACBLIB in the DFSDFxxx member of the IMS PROCLIB data set, the DEDB Alter utility supports both FDBR and XRF for ACBSHR=Y and ACBSHR=N.

If you enable IMS to manage ACBs by specifying ACBMGMT=CATALOG in the DFSDFxxx member, the DEDB Alter utility supports IMS-managed ACBs for both ACBSHR=Y and ACBSHR=N. When IMS manages ACBs, the DEDB Alter utility supports only FDBR.

Restriction: Because the IMS management of ACBs does not support XRF environments, the DEDB Alter utility does not support XRF when IMS manages ACBs.

If you use ACB libraries and the DEDB Alter utility successfully alters the definition of an online DEDB database, the DEDB definition that was used to alter the DEDB remains in the staging ACB library data set. If you use IMS managed ACBs, the DEDB definition is deleted from the IMS staging directory after the DEDB Alter utility successfully completes. If IMS manages ACBs and the DEDB Alter utility fails, pending DEDB definitions remain in the IMS catalog and you do not need to prepare the DEDB definition in the IMS catalog.

Requirements:
- A two-stage randomizer must be used, which enables areas to be processed individually.
- All IMS data sharing systems must be at IMS 15 or later.
- The DEDB databases must be registered to DBRC.
- If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

The DEDB Alter function supports both single and multiple area data sets (ADS). You have control over the output of the DEDB Alter function as either single or multiple ADSs depending on the number of shadow area data sets that you define to DBRC for the DEDB database.

The DEDB Alter function does not support DEDB databases that are in Fast Path Virtual Storage Option (VSO) or Shared Virtual Storage Option (SVSO) mode. DEDB databases using the VSO or SVSO option must first be unloaded to DASD using the /VUNLOAD command before running the DEDB Alter utility.

Related reference:

▶ DEDB Alter utility (DBFUDA00) (Database Utilities)

Altering the size attributes of an active DEDB area with the DEDB Alter utility

You can alter the DEDB area definitions of the SIZE, UOW, and ROOT parameters for an active DEDB area to increase the size of the DEDB area while the DEDB area remains accessible online by using the DEDB Alter utility (DBFUDA00).

The DEDB Alter utility's ALTERAREA function supports DEDB databases with and without sequential dependent (SDEP) segments defined.

You can also use the ALTERAREA function to add new fields to existing free space that is in a segment of an active DEDB area.

1. Update the size attributes of the active DEDB area.
   - If you use the DBDGEN utility, update the AREA statement with any changes to the SIZE, UOW, and ROOT parameters, as necessary.
   - If you use DDL, use the ALTER TABLESPACE statement with the SIZE PRIMARY, UOW, and ROOT parameters, as necessary.

2. If you are adding one or more fields to existing free space that is in a segment of an active DEDB area, use one of the following methods to add the fields:
   - If you use the DBDGEN utility, add one or more FIELD statements under the SEGM statement of the segment that you are altering.
   - If you use DDL, use the ALTER TABLE statement with the INTERNALNAME parameter.

3. If you are replacing the randomizer or adding a Segment Edit/Compression exit routine, assemble and link edit the randomizer and the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

   If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

4. If you use the IMS generation utilities, run the DBDGEN utility with the updates to the AREA and FIELD statements to alter the new DEDB DBD definitions for the active DEDB area.

5. If you use the IMS generation utilities, run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

   If you are using the IMS catalog, use one the following methods to update it after the DEDB Alter utility finishes:
   - Run both the ACB Maintenance utility and the IMS Catalog Populate utility (DFS3PU00).
   - Run the IMS catalog ACBGEN utility (DFS3UACB), which builds the ACBs and populates the IMS catalog in a single job step.

6. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:
   - By using the staging ACBLIB that you created in step 4 as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.
   - Use DDL.

7. Allocate one or more shadow DEDB area data sets for an active DEDB area to be altered:
   - Allocate two to six new shadow DEDB area data sets if an active DEDB area is defined as multiple area data sets. The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image copy data set

is required. If you need seven area data sets for multiple area data sets, run the DEDB Create utility to create the seventh area data set after the DEDB Alter utility finishes.

- You can change from single area data set to multiple area data sets at DEDB Alter processing:
  – Allocate two to six new shadow DEDB area data sets if it is to change to multiple area data sets.
  – The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image copy data set is required.
- You can change from multiple area data sets to single area data set at DEDB Alter processing:
  – Allocate one new shadow DEDB area data set if it is to change to single area data set.

The VSAM control interval size must match the SIZE= parameter on the AREA macro statement or the SIZE PRIMARY parameter on the ALTER TABLESPACE statement for the active DEDB area to be altered in the new DEDB DBD definitions.

If a DEDB database has a sequential dependent (SDEP) segment defined, the shadow area SDEP portion must be greater than or equal to the active area SDEP portion, and the shadow area CI size must be greater than or equal to the active area CI size.

8. Register the shadow data sets that were allocated in step 7 on page 774 for an active DEDB area to be altered to DBRC by using the DBRC INIT.ADS command with the SHADOW option. One must be registered as a SHADOW IC dataset with DBRC. The shadow area data sets are unavailable until the DEDB Alter utility is run to replace the active area data sets with the shadow area data sets:

```
INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW
```

The ADDN(*name*) for the data sets' ddname and ADSN(*name*) for the data sets' dsname must be unique for each shadow area data set, and they must be different from those of its associated active area data set.

9. For the active DEDB area to be altered, format the data sets with the updated DEDB area definitions. Use the DEDB Initialization utility (DBFUMIN0) with the SHADOW control statement to format the data sets in the staging ACBLIB. After the DEDB Initialization utility runs, these shadow DEDB area data sets remain inaccessible to IMS online system but are marked available in DBRC under the 'SHADOW ADS LIST:'

10. Register one of the shadow area data sets, which was allocated in step 7 on page 774, as a shadow image copy data set for an active DEDB area to be altered to DBRC. Use the DBRC INIT.ADS command with SHADOW and IC options to register the shadow image copy data set. The shadow image copy data set is unavailable until the DEDB Alter utility is run to replace the active DEDB area data sets with the shadow area data sets. The image copy is created by the DEDB Alter utility during execution.

```
INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW IC
```

The ADDN(name) for the shadow image copy data set's ddname and ADSN(name) for the shadow image copy data set's dsname must be different from those of its associated area data set.

11. Format one shadow image copy data set for the active DEDB area to be altered with the updated DEDB area definitions for the DEDB database associated with the active DEDB area to be altered in the staging ACBLIB using the DEDB Initialization utility (DBFUMIN0).

After the DEDB Initialization utility runs, the shadow image copy data set remains inaccessible to IMS online systems.

You can format shadow area data sets and shadow image copy data sets for a DEDB area to be altered in one DEDB Initialization utility execution.

After the DEDB Area Initialization utility completes, the shadow area and shadow image copy data sets are marked as `SHADOW AVAIL` in the RECON.

12. Run the DEDB Alter utility using the ALTERAREA function to alter an active DEDB area while the DEDB area remains accessible online.

If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

After successful completion of the ALTERAREA of the DEDB Alter utility, the shadow area data sets are promoted to the active area data sets. The old active area data set is demoted to the shadow area data set, and is no longer accessible to IMS online systems.

The shadow image copy data sets are promoted to user image copy data sets.

You can delete the old active DEDB area data set in DBRC at a later time when you no longer have a need for the old active DEDB area data set using the DELETE.ADS command.

```
DELETE.ADS ADDN(name) AREA(name) DBD(name)
```

The ADDN(name) is the ddname of the old active area data set.

If you are adding fields to existing free space that is in a segment of an active DEDB area, the fields are added while the DEDB database remains accessible online.

Related reference:

⬛➡ ALTER TABLE (Application Programming APIs)

⬛➡ DEDB Alter utility (DBFUDA00) (Database Utilities)

⬛➡ DEDB Initialization utility (DBFUMIN0) (Database Utilities)

⬛➡ AREA statement (System Utilities)

⬛➡ FIELD statements (System Utilities)

⬛➡ SEGM statements (System Utilities)

⬛➡ ACB Generation and Catalog Populate utility (DFS3UACB) (System Utilities)

⬛➡ INIT.ADS command (Commands)

Adding a DEDB area online with the DEDB Alter utility

You can add one or more DEDB areas at the end of a DEDB database while the DEDB area remains accessible online. With the ADDAREA function of the DEDB Alter utility (DBFUDA00), you can also replace the DEDB randomizer for the DEDB database to be altered and add a Segment Edit/Compression exit routine for a segment.

You can also use the ADDAREA function to add new fields to existing free space that is in a segment of an existing active DEDB area.

A maximum of 100 areas can be added to the end of a DEDB database per DEDB Alter utility execution.

The DEDB Alter utility's ADDAREA function supports DEDB databases with and without SDEP segments defined.

1. If you are replacing the randomizer or adding a Segment Edit/Compression exit routine, assemble and link edit the randomizer and the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

   If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

2. Add one or more DEDB areas at the end of the online DEDB database that you are altering.
   - If you use the DBDGEN utility, add one or more AREA statements. Each AREA statement can have different SIZE, UOW, and ROOT parameters in the DEDB DBD definitions.
   - If you use DDL, use one or more CREATE TABLESPACE statements. Each CREATE TABLESPACE statement can have different SIZE PRIMARY, UOW, and ROOT parameters in the DEDB DBD definitions.

3. If you are adding one or more fields to existing free space that is in a segment of an existing active DEDB area, use one of the following methods to add the fields:
   - If you use the DBDGEN utility, add one or more FIELD statements under the SEGM statement of the segment that you are altering.
   - If you use DDL, use the ALTER TABLE statement with the INTERNALNAME parameter.

4. If you are replacing the randomizer, change the RMNAME parameter in the DEDB DBD definitions.
   - If you use the DBDGEN utility, change the RMNAME parameter on the DBD statement.
   - If you use DDL, change the RMNAME parameter on the ALTER DATABASE statement.

5. If you are adding a Segment Edit/Compression exit routine, perform one of the following options:
   - If you use the DBDGEN utility, add the COMPRTN parameter on a SEGM statement.
   - If you use DDL, add the EDITPROC parameter on the CREATE TABLE statement.

6. If you use the IMS generation utilities, run the DBDGEN utility with one or more DEDB AREA statements added at the end of the DEDB DBD definitions.

7. If you use the IMS generation utilities, run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

8. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:

- By using the staging ACBLIB that you created in step 6 as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.
    - Use DDL.
9. Allocate one or more DEDB area data sets for each DEDB area to be added:
    - Allocate one new DEDB area data set if the DEDB area to be added is defined as a single area data set (SADS).
    - Allocate one to seven new DEDB area data sets if the DEDB area to be added is defined as multiple area data sets (MADS).

    The VSAM control interval size must match the SIZE= parameter on the AREA statement or the SIZE PRIMARY parameter on the CREATE TABLESPACE statement for the DEDB area to be added in the new DEDB DBD definitions.
10. Register each shadow area for each DEDB area to be added to DBRC by using the DBRC INIT.DBDS command with the SHADOW option.

    `INIT.DBDS AREA(name) DBD(name) SHADOW`
11. Register one or more DEDB area data sets for each DEDB area to be added to DBRC by using the DBRC INIT.ADS command. The DEDB area data sets are unavailable until you run the DEDB Alter utility to add one or more DEDB areas at the end of a DEDB database online.

    `INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name)`

    The ADDN(name) for the area data set's ddname and the ADSN(name) for the area data set's dsname must be unique for each area data set.
12. Format one or more DEDB area data sets for each DEDB area to be added at the end of the DEDB database in the staging ACBLIB, by using the DEDB Initialization utility (DBFUMIN0) with the ACTIVE control statement. After the DEDB Initialization utility runs, these DEDB area data sets for the shadow area remain inaccessible to IMS online systems. The DEDB area data sets are marked as `AVAIL` in the RECON.
13. Run the DEDB Alter utility by using the ADDAREA function to add one or more DEDB areas while the DEDB database remains accessible online. The DEDB Alter utility adds one or more DEDB areas at the end of a DEDB database online when the utility is run.

    If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

If the parameter on the ADDAREA control statement is invalid, or if one ore more of the DEDB areas to be added are not at the end of the DEDB database, the DEDB Alter utility terminates.

If the total number of DEDB areas after the ADDAREA function exceeds 2048 DEDB areas, the DEDB Alter utility terminates.

After the DEDB Alter utility completes successfully, the SHADOW status for the area is reset in the RECON. The area is now active and accessible to IMS online systems.

If you are replacing the randomizer, the new randomizer replaces the old randomizer for all DEDB areas of the DEDB database.

If you are adding a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine is invoked for all DEDB areas of the DEDB database.

If you are adding fields to existing free space that is in a segment of an existing active DEDB area, the fields are added while the DEDB database remains accessible online.

Related reference:

⇨ CREATE TABLE (Application Programming APIs)

⇨ CREATE TABLESPACE (Application Programming APIs)

⇨ ALTER TABLE (Application Programming APIs)

⇨ DEDB Alter utility (DBFUDA00) (Database Utilities)

⇨ INIT.ADS command (Commands)

⇨ INIT.DBDS command (Commands)

⇨ AREA statement (System Utilities)

⇨ DBD statements (System Utilities)

⇨ FIELD statements (System Utilities)

⇨ SEGM statements (System Utilities)

⇨ IMS Catalog Populate utility (DFS3PU00) (System Utilities)

Changing a randomizer when altering an active DEDB area with the DEDB Alter utility

Instead of using the REPLRAND function of the DEDB Alter utility (DBFUDA00), you can also change the randomizer name for an active DEDB with the ALTERAREA or ADDAREA functions. Only two-stage randomizers can be used with DEDB Alter. The randomizer must not alter the data location of existing areas in the DEDB. This function allows the renaming of a randomizer or preparation for adding new DEDB areas in the future.

Related reference:

⇨ DEDB Alter utility (DBFUDA00) (Database Utilities)

Changing a randomizer when altering an active DEDB area with the DEDB Alter utility and IMS generation utilities:

Instead of using the REPLRAND function of the DEDB Alter utility (DBFUDA00), you can also change the randomizer name for an active DEDB with the ALTERAREA or ADDAREA functions. Only two-stage randomizers can be used with DEDB Alter. The randomizer must not alter the data location of existing areas in the DEDB. This function allows the renaming of a randomizer or preparation for adding new DEDB areas in the future.

When changing a randomizer, the DEDB Alter utility's ALTERAREA, ADDAREA, and REPLRAND functions support DEDB databases with or without sequential dependent (SDEP) segments defined.

If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any

areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

1. Prepare for altering the DEDB randomizer online using ALTERAREA:

   a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

   b. If you are adding a Segment Edit/Compression exit routine, assemble and link-edit the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

   If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

   c. Specify the name of the new two-stage randomizer on the RMNAME parameter in the definition of the DEDB database that is associated with the DEDB area to be altered.

   d. If you are adding one or more fields to existing free space that is in a segment of an active DEDB area, add one or more FIELD statements under the SEGM statement of the segment that you are altering.

   e. If you are adding a Segment Edit/Compression exit routine, add the COMPRTN parameter on a SEGM statement in the DEDB DBD definitions.

   f. Alter the SIZE, UOW, and ROOT parameters on the AREA statement for the active DEDB area to be altered in the DEDB DBD definitions. If the DEDB database has SDEP defined, the SIZE parameter of the DEDB area to be altered in the DEDB DBD definitions must be greater than or equal to the SIZE parameter of the active DEDB area.

   g. If you are adding a Segment Edit/Compression exit routine, update the COMPRTN parameter in the DBDGEN utility.

   h. Run the DBDGEN utility with the updated SIZE, UOW, and ROOT parameters to alter the new DEDB DBD definitions for an active DEDB area, and the updated RMNAME.

   i. Run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

   j. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:

      • By using the staging ACBLIB that you created in step 1h as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.

      • Use DDL.

   k. Allocate one or more shadow DEDB area data sets for an active DEDB area to be altered.

      • Allocate one new shadow DEDB area data set if the active DEDB area is defined as a single area data set.

      • Allocate two to six new shadow DEDB area data sets if the active DEDB area is defined as multiple area data sets. The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image

copy data set is required. If you need seven area data sets for multiple area data sets, run the DEDB Create utility to create the seventh area data set after the DEDB Alter utility finishes.

You can change from single area data set to multiple area data sets at DEDB Alter processing:

- Allocate two to six new shadow DEDB area data sets if it is to change to multiple area data sets.
- The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image copy data set is required.

You can change from multiple area data sets to single area data set at DEDB Alter processing: allocate one new shadow DEDB area data set if it is to change to single area data set.

The VSAM control interval size must match the SIZE= parameter on the AREA statement for the active DEDB area to be altered in the new DEDB DBD definitions.

If a DEDB database has SDEP segments defined, the shadow area SDEP part must be greater than or equal to the active area SDEP part, and the shadow area CI size must be greater than or equal to the active area CI size.

l. Register one or more shadow DEDB area data sets for an active DEDB area to be altered to DBRC using the DBRC INIT.ADS command with the SHADOW option. The shadow DEDB area data sets are unavailable until the DEDB Alter utility is run to replace the active DEDB area data sets with the shadow DEDB area data sets.

   `INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW`

   The ADDN(name) for the shadow area data set's ddname and the ADSN(name) for the shadow area data set's dsname must be unique for each shadow area data set, and they must be different from those of the associated active area data set.

m. Format one or more shadow DEDB area data sets for the active DEDB area to be altered with the updated SIZE, UOW, and ROOT DEDB area definitions for the DEDB database associated with the active DEDB area to be altered in the staging ACBLIB using the DEDB initialization utility (DBFUMIN0). After the DEDB Initialization utility runs, the shadow DEDB area data sets remain inaccessible to IMS online systems.

n. Register one of the shadow area data sets, which was allocated in step 1k on page 780, as a shadow image copy data set for an active DEDB area to be altered to DBRC. Use the DBRC INIT.ADS command with SHADOW and IC options to register the shadow image copy data set. The shadow image copy data set is unavailable until the DEDB Alter utility is run to replace the active DEDB area data sets with the shadow DEDB area data sets.

   `INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW IC`

   The ADDN(name) for the shadow image copy data set's ddname and the ADSN(name) for the shadow image copy data set's dsname must be different from those of the associated active area data set.

o. Format one shadow image copy data set for the active DEDB area to be altered with the updated SIZE, UOW, and ROOT DEDB area definitions for the DEDB database associated with the active DEDB area to be altered in the staging ACBLIB using the DEDB Initialization utility (DBFUMIN0).

   After the DEDB Initialization utility runs, the shadow image copy data set remains inaccessible to IMS online systems.

You can format shadow area data sets and shadow image copy data sets for a DEDB area to be altered in one DEDB Initialization utility execution.

After the DEDB Area Initialization utility completes, the shadow area and shadow image copy data sets are marked as `SHADOW AVAIL` in the RECON.

2. Prepare to add DEDB areas at the end with the DEDB randomizer online using ADDAREA:

   a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIP concatenation.

   b. Alter the RMNAME parameter on the DBD statement with a different two-stage DEDB randomizer name for the DEDB database associated with the active DEDB area to be altered in the DEDB DBD definitions.

   c. Alter the SIZE, UOW, and ROOT parameters on the AREA statement for the active DEDB area to be altered in the DEDB DBD definitions.

   d. Run the DBDGEN utility with the updated SIZE, UOW, and ROOT parameters to alter the new DEDB DBD definitions for an active DEDB area and the updated RMNAME.

   e. Run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

   f. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:

      • By using the staging ACBLIB that you created in step 2e as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.

      • Use DDL.

   g. Allocate one or more shadow DEDB area data sets for an active DEDB area to be altered.

      • Allocate one new shadow DEDB area data set if an active DEDB area is defined as a single area data set.

      • Allocate 1-7 new shadow DEDB area data sets if an active DEDB area is defined as multiple area data sets.

      The VSAM control interval size must match the SIZE= parameter on the AREA statement for the active DEDB area to be added in the new DEDB DBD definitions.

   h. Register one or more shadow DEDB area data sets for an active DEDB area to be added to DBRC using the DBRC INIT.ADS command with the SHADOW option. The shadow DEDB area data sets are unavailable until the DEDB Alter utility is run to add one or more DEDB areas to the end of the DEDB database online

      `INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW`

      The ADDN(name) for the shadow area data set's ddname and the ADSN(name) for the shadow area data set's dsname must be unique for each shadow area data set and they must be different from those of the associated active area data set.

   i. Format one or more shadow DEDB area data sets for each DEDB area to be added at the end of the DEDB database in the staging ACBLIB by using the DEDB initialization utility (DBFUMIN0). After the DEDB Initialization utility runs, the DEDB shadow area data sets remain inaccessible to IMS online systems.

3. Perform the following steps to alter a DEDB randomizer of a DEDB database online:

a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

b. Alter the RMNAME parameter on the DBD statement with a different two-stage DEDB randomizer name for the DEDB database that is associated with the active DEDB area to be altered in the DEDB DBD definitions.

c. Run the DBDGEN utility with the new, altered DEDB AREA statement and the randomizer change in the DEDB DBD statement that are in the DEDB DBD definitions.

d. Run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

e. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:

- By using the staging ACBLIB that you created in step 3d as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.
- Use DDL.

4. Run the DEDB Alter utility using the ALTERAREA function to alter an active DEDB area with randomizer change while the DEDB area remains accessible online. The DEDB Alter utility alters one active DEDB area with randomizer change when the utility is run. The DEDB randomizer is altered as part of the ALTERAREA function of the DEDB Alter utility. The randomizer change does note impact the distribution of data across the DEDB areas that are not altered by the DEDB Alter utility for the DEDB database randomizer to be altered. The new randomizer is not shared with other DEDB databases unless it is an existing randomizer for other DEDB databases.

After the DEDB Alter utility has run successfully, the shadow area data set is promoted to the active area data set. The shadow Image Copy data set becomes the user Image Copy data set. The old active area data set is demoted to the shadow area data set. The old active area data set is not accessible to IMS online systems any longer.

You can delete the old active DEDB area data set in DBRC at a later time when you no longer have a need for the old active DEDB area data set using the DELETE.ADS command.

```
DELETE.ADS ADDN(name) AREA(name) DBD(name)
```

The ADDN(name) is the ddname of the old active area data set.

5. Run the DEDB Alter utility using the ADDAREA function to add one or more DEDB areas with randomizer change while the DEDB database remains accessible online. The DEDB Alter utility adds one or more DEDB areas at the end of a DEDB database online when the utility is run. The DEDB randomizer name can be changed by using the ADDAREA function of the DEDB Alter utility. The randomizer change does not affect the distribution of data across the DEDB areas that are not altered by the DEDB Alter utility. The new randomizer is shared with other DEDB databases.

After the DEDB Alter utility completes successfully, the SHADOW status for the area is reset in the RECON. The area is now active and accessible to IMS online systems.

6. If you are using the IMS Catalog, use one the following methods to update it after the DEDB Alter utility finishes:

- Run both the ACB Maintenance utility and the IMS Catalog Populate utility (DFS3PU00).

- Run the IMS Catalog ACBGEN utility (DFS3UACB), which builds the ACBs and populates the IMS Catalog in a single job step.

Related reference:

▢➜ ALTER TABLE (Application Programming APIs)

▢➜ ALTER TABLESPACE (Application Programming APIs)

▢➜ DEDB Alter utility (DBFUDA00) (Database Utilities)

▢➜ DEDB Initialization utility (DBFUMIN0) (Database Utilities)

▢➜ AREA statement (System Utilities)

▢➜ DBD statements (System Utilities)

▢➜ FIELD statements (System Utilities)

▢➜ SEGM statements (System Utilities)

▢➜ IMS Catalog Populate utility (DFS3PU00) (System Utilities)

▢➜ ACB Generation and Catalog Populate utility (DFS3UACB) (System Utilities)

▢➜ INIT.ADS command (Commands)

Changing a randomizer when altering an active DEDB area with the DEDB Alter utility and DDL:

Instead of using the REPLRAND function of the DEDB Alter utility (DBFUDA00), you can also change the randomizer name for an active DEDB with the ALTERAREA or ADDAREA functions. Only two-stage randomizers can be used with DEDB Alter. The randomizer must not alter the data location of existing areas in the DEDB. This function allows the renaming of a randomizer or preparation for adding new DEDB areas in the future.

When changing a randomizer, the DEDB Alter utility's ALTERAREA, ADDAREA, and REPLRAND functions support DEDB databases with or without sequential dependent (SDEP) segments defined.

If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

1. Prepare for altering the DEDB randomizer online using ALTERAREA:
    a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.
    b. If you are adding a Segment Edit/Compression exit routine, assemble and link-edit the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

        If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and

writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

c. Specify the name of the new two-stage randomizer on the RMNAME parameter of the ALTER DATABASE statement for the definition of the DEDB database that is associated with the DEDB area to be altered.

d. If you are adding one or more fields to existing free space that is in a segment of an active DEDB area, use the ALTER TABLE statement with the INTERNALNAME parameter.

e. If you are adding a Segment Edit/Compression exit routine, add the EDITPROC parameter on the ALTER TABLE statement for the DEDB DBD definitions.

f. Alter the SIZE PRIMARY, UOW, and ROOT parameters on the ALTER TABLESPACE statement for the active DEDB area to be altered in the DEDB DBD definitions.

If the DEDB database has SDEP defined, the SIZE PRIMARY parameter of the DEDB area to be altered in the DEDB DBD definitions must be greater than or equal to the SIZE PRIMARY parameter of the active DEDB area.

g. If you are adding a Segment Edit/Compression exit routine, update the EDITPROC parameter on the ALTER TABLE statement.

h. Allocate one or more shadow DEDB area data sets for an active DEDB area to be altered.

- Allocate one new shadow DEDB area data set if the active DEDB area is defined as a single area data set.
- Allocate two to six new shadow DEDB area data sets if the active DEDB area is defined as multiple area data sets. The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image copy data set is required. If you need seven area data sets for multiple area data sets, run the DEDB Create utility to create the seventh area data set after the DEDB Alter utility finishes.

You can change from single area data set to multiple area data sets at DEDB Alter processing:

- Allocate two to six new shadow DEDB area data sets if it is to change to multiple area data sets.
- The DEDB Alter utility supports six shadow DEDB area data sets, because one shadow image copy data set is required.

You can change from multiple area data sets to single area data set at DEDB Alter processing: allocate one new shadow DEDB area data set if it is to change to single area data set.

The VSAM control interval size must match the SIZE PRIMARY parameter on the ALTER TABLESPACE statement for the active DEDB area to be altered in the new DEDB DBD definitions.

If a DEDB database has SDEP segments defined, the shadow area SDEP part must be greater than or equal to the active area SDEP part, and the shadow area CI size must be greater than or equal to the active area CI size.

i. Register one or more shadow DEDB area data sets for an active DEDB area to be altered to DBRC using the DBRC INIT.ADS command with the SHADOW option. The shadow DEDB area data sets are unavailable until the DEDB Alter utility is run to replace the active DEDB area data sets with the shadow DEDB area data sets.

```
INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW
```

The ADDN(name) for the shadow area data set's ddname and the ADSN(name) for the shadow area data set's dsname must be unique for each shadow area data set, and they must be different from those of the associated active area data set.

j. Format one or more shadow DEDB area data sets for the active DEDB area to be altered with the updated SIZE PRIMARY, UOW, and ROOT DEDB area definitions for the DEDB database associated with the active DEDB area to be altered in the IMS directory staging data set using the DEDB Initialization utility (DBFUMIN0). After the DEDB Initialization utility runs, the shadow DEDB area data sets remain inaccessible to IMS online systems.

k. Register one of the shadow area data sets, which was allocated in step 1h on page 785, as a shadow image copy data set for an active DEDB area to be altered to DBRC. Use the DBRC INIT.ADS command with SHADOW and IC options to register the shadow image copy data set. The shadow image copy data set is unavailable until the DEDB Alter utility is run to replace the active DEDB area data sets with the shadow DEDB area data sets.

```
INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW IC
```

The ADDN(name) for the shadow image copy data set's ddname and the ADSN(name) for the shadow image copy data set's dsname must be different from those of the associated active area data set.

l. By using the DEDB Initialization utility (DBFUMIN0), format one shadow image copy data set for the active DEDB area to be altered with the updated SIZE PRIMARY, UOW, and ROOT DEDB area definitions that are in the IMS directory staging data set. This alters the DEDB database that is associated with the active DEDB area.

After the DEDB Initialization utility runs, the shadow image copy data set remains inaccessible to IMS online systems.

You can format shadow area data sets and shadow image copy data sets for a DEDB area to be altered in one DEDB Initialization utility execution.

After the DEDB Area Initialization utility completes, the shadow area and shadow image copy data sets are marked as SHADOW AVAIL in the RECON.

2. Prepare to add DEDB areas at the end with the DEDB randomizer online using ADDAREA:

a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIP concatenation.

b. Alter the RMNAME parameter on the ALTER DATABASE statement with a different two-stage DEDB randomizer name. This alters the DEDB database that is associated with the active DEDB area in the DEDB DBD definitions.

c. Alter the SIZE PRIMARY, UOW, and ROOT parameters on the ALTER TABLESPACE statement for the active DEDB area to be altered in the DEDB DBD definitions.

d. Allocate one or more shadow DEDB area data sets for an active DEDB area to be altered.

• Allocate one new shadow DEDB area data set if an active DEDB area is defined as a single area data set.

• Allocate 1-7 new shadow DEDB area data sets if an active DEDB area is defined as multiple area data sets.

The VSAM control interval size must match the SIZE PRIMARY parameter on the ALTER TABLESPACE statement for the active DEDB area to be added in the new DEDB DBD definitions.

e. Register one or more shadow DEDB area data sets for an active DEDB area to be added to DBRC using the DBRC INIT.ADS command with the SHADOW option. The shadow DEDB area data sets are unavailable until the DEDB Alter utility is run to add one or more DEDB areas to the end of the DEDB database online

```
INIT.ADS ADDN(name) ADSN(name) AREA(name) DBD(name) SHADOW
```

The ADDN(name) for the shadow area data set's ddname and the ADSN(name) for the shadow area data set's dsname must be unique for each shadow area data set and they must be different from those of the associated active area data set.

f. Format one or more shadow DEDB area data sets for each DEDB area to be added at the end of the DEDB database in the IMS directory staging data set by using the DEDB Initialization utility (DBFUMIN0). After the DEDB Initialization utility runs, the DEDB shadow area data sets remain inaccessible to IMS online systems.

3. Perform the following steps to alter a DEDB randomizer of a DEDB database online:

a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

b. Alter the RMNAME parameter on the ALTER DATABASE statement with a different two-stage DEDB randomizer name. This alters the DEDB database that is associated with the active DEDB area in the DEDB DBD definitions.

4. Run the DEDB Alter utility using the ALTERAREA function to alter an active DEDB area with randomizer change while the DEDB area remains accessible online. The DEDB Alter utility alters one active DEDB area with randomizer change when the utility is run. The DEDB randomizer is altered as part of the ALTERAREA function of the DEDB Alter utility. The randomizer change does note impact the distribution of data across the DEDB areas that are not altered by the DEDB Alter utility for the DEDB database randomizer to be altered. The new randomizer is not shared with other DEDB databases unless it is an existing randomizer for other DEDB databases.

After the DEDB Alter utility has run successfully, the shadow area data set is promoted to the active area data set. The shadow Image Copy data set becomes the user Image Copy data set. The old active area data set is demoted to the shadow area data set. The old active area data set is not accessible to IMS online systems any longer.

You can delete the old active DEDB area data set in DBRC at a later time when you no longer have a need for the old active DEDB area data set using the DELETE.ADS command.

```
DELETE.ADS ADDN(name) AREA(name) DBD(name)
```

The ADDN(name) is the ddname of the old active area data set.

5. Run the DEDB Alter utility using the ADDAREA function to add one or more DEDB areas with randomizer change while the DEDB database remains accessible online. The DEDB Alter utility adds one or more DEDB areas at the end of a DEDB database online when the utility is run. The DEDB randomizer name can be changed by using the ADDAREA function of the DEDB Alter utility. The randomizer change does not affect the distribution of data across the DEDB areas that are not altered by the DEDB Alter utility. The new randomizer is shared with other DEDB databases.

After the DEDB Alter utility completes successfully, the SHADOW status for the area is reset in the RECON. The area is now active and accessible to IMS online systems.

Related reference:

- DEDB Alter utility (DBFUDA00) (Database Utilities)

- DEDB Initialization utility (DBFUMIN0) (Database Utilities)

- ALTER DATABASE (Application Programming APIs)

- ALTER TABLE (Application Programming APIs)

- ALTER TABLESPACE (Application Programming APIs)

- INIT.ADS command (Commands)

Replacing an active DEDB database randomizer online with the DEDB Alter utility

You can replace an active DEDB database randomizer while the DEDB database remains accessible online by using the REPLRAND function of the DEDB Alter utility (DBFUDA00). The DEDB Alter utility's randomizer change supports DEDB databases with and without SDEP segments defined.

You can also use the REPLRAND function to add new fields to existing free space that is in a segment of an active DEDB area.

If the DEDB randomizer has been changed as part of the ALTERAREA function and you want to change back to the original DEDB randomizer name, run the DEDB Alter utility using the REPLRAND function. Doing so changes the new randomizer name to the original DEDB randomizer name after the DEDB Alter ALTERAREA function finishes.

1. Prepare for replacing the DEDB randomizer online

   a. Assemble and link-edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

   b. If you are adding a Segment Edit/Compression exit routine, assemble and link-edit the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

      If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.

   c. Alter the RMNAME parameter with a different two-stage DEDB randomizer name for the DEDB database associated with the active DEDB area to be altered in the DEDB DBD definitions.

      - If you use the DBDGEN utility, alter the RMNAME parameter on the DBD statement.

      - If you use DDL, alter the RMNAME parameter on the ALTER DATABASE statement.

   d. If you are adding one or more fields to existing free space that is in a segment of an active DEDB area, use one of the following methods to add the fields:

      - If you use the DBDGEN utility, add one or more FIELD statements under the SEGM statement of the segment that you are altering.

- If you use DDL, use the ALTER TABLE statement with the INTERNALNAME parameter.

  e. If you are adding a Segment Edit/Compression exit routine, perform one of the following options:
     - If you use the DBDGEN utility, add the COMPRTN parameter on the SEGM statement.
     - If you use DDL, add the EDITPROC parameter on the ALTER TABLE statement.

  f. If you are adding a Segment Edit/Compression exit routine, update the DEDB DBD definitions to include the routine.
     - If you use the generation utilities, update the COMPRTN parameter on the SEGM statement in the DBDGEN utility.
     - If you use DDL, update the EDITPROC parameter on the ALTER TABLE statement.

  g. If you use the IMS generation utilities, run the DBDGEN utility with the new and updated parameters on the DBD statement in the DEDB DBD definitions.

  h. If you use generation utilities, run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

  i. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:
     - By using the staging ACBLIB that you created in step 1h as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.
     - Use DDL.

2. Run the DEDB Alter utility using the REPLRAND function to replace a randomizer while the DEDB database remains accessible online.

   If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

3. If you are using the IMS Catalog, use one the following methods to update it after the DEDB Alter utility finishes:
   - Run both the ACB Maintenance utility and the IMS Catalog Populate utility (DFS3PU00).
   - Run the IMS Catalog ACBGEN utility (DFS3UACB), which builds the ACBs and populates the IMS Catalog in a single job step.

After the DEDB Alter utility completes successfully, the new randomizer replaces the old randomizer for the DEDB database while the DEDB database remains accessible online.

If you are adding a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine is invoked for all DEDB areas of the DEDB database.

If you are adding fields to existing free space that is in a segment of an active DEDB area, the fields are added while the DEDB database remains accessible online.

Related reference:

- ALTER DATABASE (Application Programming APIs)
- ALTER TABLE (Application Programming APIs)
- DEDB Alter utility (DBFUDA00) (Database Utilities)
- DBD statements (System Utilities)
- FIELD statements (System Utilities)
- SEGM statements (System Utilities)

Adding new fields to a segment of an active DEDB area with the DEDB Alter utility

You can add one or more fields to existing free space that is in a segment of an active DEDB area while the DEDB area remains accessible online. With the ALTERDB function of the DEDB Alter utility (DBFUDA00), you also can replace the DEDB randomizer for the DEDB database to be altered and add a Segment Edit/Compression exit routine for a segment.

The DEDB Alter utility's ALTERDB function supports DEDB databases with and without sequential dependent (SDEP) segments defined.

1. Prepare for adding new fields to an active DEDB area online:
   a. If you are replacing the randomizer or adding a Segment Edit/Compression exit routine, assemble and link edit the randomizer and the Segment Edit/Compression exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.

      If you are using the DEDB Alter utility to add a Segment Edit/Compression exit routine, the Segment Edit/Compression exit routine must be able to handle mixed compressed and non-compressed data in a DEDB database. If data is non-compressed, on read access, it does not expand the non-compressed data, and on write access, it compresses the data and writes out as compressed data. If data is compressed, on read access, it expands the compressed data, and on write access, it compresses the data and writes out as compressed data.
   b. Add one or more fields to existing free space that is in a segment of an active DEDB area.
      - If you use the DBDGEN utility, add one or more FIELD statements under the SEGM statement of the segment that you are altering.
      - If you use DDL, use the ALTER TABLE statement with the INTERNALNAME parameter.
   c. If you are replacing the randomizer, change the RMNAME parameter in the DEDB DBD definitions.
      - If you use the DBDGEN utility, change the RMNAME parameter on the DBD statement.
      - If you use DDL, change the RMNAME parameter on the ALTER DATABASE statement.
   d. If you are adding a Segment Edit/Compression exit routine, perform one of the following options:
      - If you use the DBDGEN utility, add the COMPRTN parameter on a SEGM statement.
      - If you use DDL, add the EDITPROC parameter on the ALTER TABLE statement.

e. If you use the IMS generation utilities, run the DBDGEN utility with the FIELD statements that you added to alter the new DEDB DBD definitions for the active DEDB area.

f. If you use the IMS generation utilities, run the ACBGEN utility for all PSBs that reference the changed DEDB DBD definitions to the staging ACBLIB.

If you are using the IMS catalog, use one of the following methods to update it after the DEDB Alter utility finishes:

- Run both the ACB Maintenance utility and the IMS Catalog Populate utility (DFS3PU00).
- Run the IMS catalog ACBGEN utility (DFS3UACB), which builds the ACBs and populates the IMS catalog in a single job step.

g. If ACBMGMT=CATALOG is specified in the DFSDFxxx member of the IMS PROCLIB data set, populate the new DEDB DBD definitions to the IMS catalog by using one of the following methods:

- By using the staging ACBLIB that you created in step 4 as input, run the IMS Catalog Populate utility (DFS3PU00) and use the MANAGEDACBS statement with the **STAGE** parameter.
- Use DDL.

2. Run the DEDB Alter utility by using the ALTERDB function to alter an active DEDB area while the DEDB area remains accessible online.

If you are adding a Segment Edit/Compression exit routine, during DEDB Alter utility execution the new Segment Edit/Compression exit is not invoked for any areas. The Segment Edit/Compression exit routine is not enabled during DEDB Alter processing. Instead, it is enabled for all DEDB areas after successful DEDB Alter utility completion.

Related reference:

➜ ALTER DATABASE (Application Programming APIs)

➜ ALTER TABLE (Application Programming APIs)

➜ DEDB Alter utility (DBFUDA00) (Database Utilities)

➜ FIELD statements (System Utilities)

➜ SEGM statements (System Utilities)

➜ IMS Catalog Populate utility (DFS3PU00) (System Utilities)

➜ ACB Generation and Catalog Populate utility (DFS3UACB) (System Utilities)

Changing databases dynamically in online systems

Dynamic resource definition allows you to add new databases to an online IMS system, update certain attributes of the database online, or remove databases from the online IMS system without requiring you to shut IMS down or perform any of the system definition process.

To use dynamic resource definition, the feature must be enabled in your IMS system.

The type-2 commands that support changing databases by using dynamic resource definition include:
- CREATE DB
- DELETE DB
- UPDATE DB

By using the QUERY DB command, you can display the database attributes that can be changed by the other commands.

Dynamic resource definition also allows you to define and store database resource descriptors, which can be used as a template to create new databases with common attributes. For example, when adding a database to an online system by using the CREATE DB command, instead of specifying each attribute individually, you can specify `CREATE DB NAME(db_name) LIKE(DESC(db_desc_name))`. The LIKE() parameter can also be used to create new databases based on existing databases or database descriptors.

To save changes made to databases online by using dynamic resource definition across cold starts, the database definitions must be either:
- Exported, either by the EXPORT DEFN command or automatic export, to the resource definition data set (RDDS) or the IMSRSC repository if IMS is enabled with the repository
- Added to the IMS MODBLKS data set by system definition and then imported during cold start. Across warm starts and emergency restarts, IMS recovers dynamic resource definition changes from the logs.

If IMS is enabled to use the IMSRSC repository, resource definitions in the IMS change list are read at the end of warm or emergency restart. IMS calls the Resource Manager (RM) to read the change list for the IMS. The IMS change list is maintained by RM in the repository.

If a change list exists, IMS imports the stored resource definitions from the IMSRSC repository for all the resources and descriptors in the change list. The IMS change list is processed after the IMS log is processed. The resources and descriptors in the change list are quiesced and are not available for use until the stored resource definitions are imported from the repository. The change list for the IMS system is deleted at the end of warm or emergency restart or at the end of IMS cold start.

Related concepts:

➥ Resource lists for the IMSRSC repository (System Definition)

➥ Overview of the IMSRSC repository (System Definition)

Related tasks:

"Introducing databases into online systems" on page 547

Changing database attributes dynamically in an online IMS system

You can change the attributes and status of an online database by using the UPDATE DB command.

For example, the UPDATE DB command can make a database available, take the database offline, stop scheduling, stop updates, lock, and unlock a database. You must stop access to databases before changing their attributes.

You can specify multiple database names on the UPDATE DB command. Each named database is processed individually and if the processing fails for an individual database, IMS returns a condition code for the individual database that explains the failure.

To view the current attributes and status of a database, issue the QUERY DB command.

By using the UPDATE DB command, you can:
* Start and stop databases, partitions, and DEDB areas
* Set the access type for databases
* Change the RESIDENT attribute of a database
* Lock or unlock a database
* In IMSplex environments, set the global status of a database

The changes to the definitional attributes of an online database are not saved across a cold start of IMS unless the database definitions are exported to the resource definition data set (RDDS) or the IMSRSC repository.

Database definitions can be exported to an RDDS or the repository by using the EXPORT DEFN command. In addition, IMS can be configured to export the definitions automatically to an RDDS or the repository during system checkpoints. You can enable automatic export by specifying AUTOEXPORT=AUTO, AUTOEXPORT=REPO, or AUTOEXPORT=RDDS in the DYNAMIC_RESOURCES section of the DFSDFxxx member in the IMS PROCLIB data set.

Related reference:

⇨ UPDATE DB command (Commands)

⇨ QUERY DB command (Commands)

Removing databases dynamically from an online IMS system

You can remove databases from an online IMS system by using the DELETE DB command.

Removing an MSDB database from an online system requires a different procedure than other database types.

Use the commands QUERY DB SHOW(WORK) and QUERY DB SHOW(PGM) to display the status of the DB and whether it is referenced by a PSB.

To remove a database other than an MSDB database from an online IMS system:
1. Stop access to the database by issuing either:
   * UPD DB NAME(name) STOP(ACCESS)
   * /DBR DB name
2. If the database uses logical relationships, you must remove the logical relationships before you can remove the database from the online system:
   a. Modify the DBDs to remove the relationships.
   b. Perform a DBDGEN and an ACBGEN with the modified DBD statements.
   c. Bring the new definitions online.
3. Remove references to the database from all PSBs in the program directory control block (PDIR) by either deleting the PDIR altogether or deleting references to the database from the PSBs and running the ACBLIB online change function.
4. Issue the DELETE DB command.
5. If IMS is enabled with either the IMSRSC repository or with system RDDS, do one of the following:
   * If IMS is enabled with the IMSRSC repository, issue a DELETE DEFN command to delete the database definition permanently from the IMSRSC repository.

- If IMS is enabled with system RDDS, issue a EXPORT DEFN command or trigger automatic export to create a new system RDDS without the deleted database definition.

  Note: An automatic export to the IMSRSC repository will not handle deleted runtime resource definitions. Issue an explicit DELETE DEFN command to delete the resource definition from the IMSRSC repository.

Removing an MSDB database dynamically from an online IMS system:

Because MSDB databases reside in the extended common storage area (ECSA), removing an MSDB database from an online system requires an IMS restart with MSDBLOAD specified.

To remove an MSDB database from an online IMS system:
1. Delete the MSDB database from the MSDBINIT data set by using the MSDB Maintenance utility (DBFDBMA0).
2. Remove the MSDB database (the BHDR control block) from the ACB library by using the online change function.
3. Remove references to the MSDB database from any PSBs.
4. Shut down IMS normally.
5. Warm start IMS with MSDBLOAD keyword. Because an MSDBINIT data set no longer exists and there is no entry for it in the ACBLIB, IMS recognizes the database, but not as an MSDB database.
6. Issue /DBR command against the MSDB database. The /DBR command is allowed against the database because IMS no longer recognizes the database as an MSDB.
7. Issue the command DELETE DB NAME(*MSDB_name*).

Online activation of new and changed database resources when IMS manages ACBs

When the IMS management of ACBs is enabled, for most types of database (DBD) and program view (PSB) resource changes, you activate the ACBs in the online IMS systems by issuing the IMPORT DEFN SOURCE(CATALOG) command.

For IMS systems that manage ACBs, the IMPORT DEFN SOURCE(CATALOG) command performs a function that is equivalent to the function that the online change process performs for IMS systems that use ACB libraries. In fact, internally the IMPORT DEFN SOURCE(CATALOG) uses a similar phased process to import the modified ACBs from the staging data set of the IMS catalog to the directory data set of the IMS catalog.

Depending on the type of changes you are making and whether you are adding new resources or modifying existing resources, additional steps might be required before or after you issue the IMPORT DEFN SOURCE(CATALOG). For example, if you are modifying the structure of an existing database, you might need to unload and reload the database before you activate ACB that contains the modified structure. Or if you are introducing a new database into an IMS system, you might need to allocate data sets or define the runtime attributes of the database by using either the CREATE DATABASE command or the DATABASE stage-1 system definition macro. In some cases, IMS might perform the some of these prerequisite steps for you.

If you are using DDL to create a new Fast Path DEDB database, you can configure IMS to create the area data sets for you automatically by specifying AUTOCREATE=YES in the catalog section of the DFSDF*xxx* PROCLIB member and specifying the attributes of the area data sets in the DDL section of the DFSDF*xxx* member.

If you are altering PSBs only, you can stage the activation of the PSBs across subsets of IMS systems that share the ACBs in an IMSplex by specifying OPTION(UPDATEPSB) FOR(IMSID(*imsid*)) on the IMPORT DEFN SOURCE(CATALOG) command.

Before issuing the IMPORT DEFN SOURCE(CATALOG), resources that will be affected by the change must be stopped.

Related concepts:

➙ Online changes in managed-ACB environments (System Administration)

Related reference:

➙ IMPORT DEFN SOURCE(CATALOG) command (Commands)

➙ CATALOG and CATALOGxxxx sections of the DFSDFxxx member (System Definition)

➙ DDL section of the DFSDFxxx member (System Definition)

Activating database changes by using the online change function

In IMS systems that use DBD, PSB and ACB libraries, you can use the online change function to apply database changes, including changes to full-function databases, full-function HALDB master databases, and Fast Path DEDB databases, to online IMS systems without stopping them.

Note: The online change function is not supported and the online change commands are rejected when the IMS management of ACBs is enabled.

In IMS systems that manage ACBs, database changes are made in the online IMS system by submitting the changes to the IMS catalog. Changes are submitted to the IMS catalog by using SQL DDL statements or by one of the population utilities for the IMS catalog. Some changes that are submitted by DDL are activated in the online system automatically. All other changes are activated by issuing the IMPORT DEFN SOURCE(CATALOG) command.

MSDBs do not support the online change function.

The online change function for DEDBs allows both database-level and area-level changes. A database-level change affects the structure of the DEDB and includes such changes as adding or deleting an area, adding a segment type, or changing the randomizing routines. An area-level change involves increasing or decreasing the size of an area (IOVF, DOVF, CI). An area-level change requires the user to stop only that area with the /DBRECOVERY command; a database-level change requires the user to stop all areas of the DEDB.

Unlike standard randomizers that distribute database records across the entire DEDB, two-stage randomizers distribute database records within an area. By using a two-stage randomizer, changes to an individual area's root addressable allocation are area-level changes, and only the areas that are affected need to be stopped.

To use online change with full-function databases, HALDB master databases, and DEDB databases, perform the following steps:

1. Allocate the data sets for online change as described in *IMS Version 15 Installation*.

2. Run a MODBLKS system definition if additions, changes, or deletions to the system definition DATABASE (and possibly APPLCTN) statements need to be made (see *IMS Version 15 System Administration* for more information).

3. Run the necessary DBDGEN, PSBGEN, and ACBGEN (see *IMS Version 15 System Utilities*).

   Note: All changes to ACBLIB members that resulted from the ACBGEN execution are available to the online system after the online change (provided that the changed PSB and DBD resources are defined in the online system).

4. Update the security definitions of the IMS system's security facilities to include any new databases. Security facilities can include RACF, another external security product, and exit routines. For more information on IMS security, see *IMS Version 15 System Administration*.

5. Allocate the database data sets for databases to be added.

6. Load your database.

7. For Fast Path, online change must be completed before the database can be loaded. Also, Fast Path can only load databases online; batch jobs cannot be used.

8. If dynamic allocation is used in a z/OS environment, run the dynamic allocation utility.

9. Use the Online Change Copy utility (DFSUOCU0) to copy your updated staging libraries to the inactive libraries (see *IMS Version 15 System Utilities* for information on running this utility).

10. Issue the operator commands to cause your inactive libraries to become your active libraries (see *IMS Version 15 Operations and Automation* for the commands that are used for online change).

If a database in a z/OS environment needs to be reorganized because of changes to the active ACBLIB data set, /DBR must be issued to deallocate the database before the /MODIFY COMMIT command that introduces the ACBGEN changes. The commands /DBR, /DBD, or /STA DATABASE ACCESS= must be completed to take the areas of the database to be changed or deleted offline before issuing the /MODIFY COMMIT command.

Related concepts:

Modifying ACB library members online

In IMS systems that use an ACB library, you can add or change individual members of the ACB library and bring these members online while IMS is running by using the ACB library member online change function.

Note: When the IMS management of ACBs is enabled, the member online change function is not supported and the member online change commands are rejected. Instead, IMS activates new and updated ACBs in the online system for you after the associated database (DBD) and program view (PSB) changes are entered in the IMS catalog. If the DBD and PSB changes are submitted by using DDL statements,

some of the changes are activated in the online system automatically. Any changes that are not activated automatically are activated by issuing the IMPORT DEFN SOURCE(CATALOG) command.

The ACB library member online change function quiesces only those resources that are affected by the online change, allowing for resources unaffected by the change to remain active concurrently during the online change process.

If any ACB library members that are being modified or deleted by the online change function have been loaded into online storage for use by IMS, IMS removes them from storage at the time of the online change. Non-resident ACB members are reloaded into storage the next time an application program that requires them is scheduled.

Related concepts:

▶ ACB library member online change (System Administration)

Related tasks:

▶ Changing or adding IMS.ACBLIB members online (System Administration)

The online change function, DEDBs, and Availability of IFP and MPP Regions

In IMS systems that use ACB libraries, you can make changes to DEDBs while maintaining the availability of IFP and MPP regions that access the DEDBs by using the online change function.Changes can be made to DEDBs using online change while maintaining the availability of IFP and MPP regions that access the DEDBs. If database level changes are made to the DEDB while an IFP/MPP is running, then the application will pseudo-abend and the PSB will be rescheduled on the next DL/I call to the DEDB.

Note: The online change function is not supported and the online change commands are rejected when the IMS management of ACBs is enabled.

If database level changes are made to the DEDB while an IFP/MPP is running, then the application will pseudoabend and the PSB will be rescheduled on the next DL/I call to the DEDB.

Two level changes can be made to DEDBs. The database level changes allow:
1. Add or Delete DEDBs.
2. Add or Delete segment types.
3. Add, Change, or Delete a segment and its fields.
4. Add, Change, or Delete segment compression routines.
5. Add, Change, or Delete data capture exit routines.
6. Change randomizers.
7. Add or Delete areas.
8. Change area RAP space allocation and the randomizer is not a 2-stage randomizer.

The area level changes allow:
1. Change area RAP space allocation and the randomizer is a 2-stage randomizer.
2. Change DOVF or IOVF space allocation.
3. Change SDEP space allocation.
4. Change CI size.

Area level changes and items 4 through 8 of the database level change require a BUILD DBD (not a BUILD PSB). In this case, with exception to items 4 and 5 when the defined PSB SENSEGs have reference to exit routines that are added or deleted, the PSB does not change. Changes can be made to DEDBs using online change while maintaining the availability of IFP and MPP regions that access the DEDBs only if there is no change to the scheduled PSB. The application will then pseudoabend with ABENDU0777 and the PSB will be rescheduled on the next DL/I call to DEDB. The message DFS2834I is issued. Other changes to the PSBs such as items 1 through 5 of the DEDB database changes, full-function database changes, or PSB changes using online change require that the IFP and MPP regions be brought down.

The following procedure describes the steps necessary to make database level changes to a DEDB with an IFP / MPP running:

1. Use a specific user-developed application program or OEM utility to unload the DEDB through existing system definitions.
2. DBDGEN, PSBGEN and ACBGEN to generate the application control blocks to implement the DEDB structural changes. The changed or new application control blocks must be built into the active IMS system's staging copy of ACBLIB, which is offline.
3. Run the online change utility, DFSUOCU0, to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.
4. Enter the normal /DBR command sequence to remove access to the DEDB from the active IMS system. If GLOBAL option is used on /DBR, Prohibit Further Authorization flag will be set in DBRC, which will cause U0458 abend when PSB is rescheduled. Recommendation for data sharing customers who normally use GLOBAL option is to add NOPFA keyword : /DBR DB NOFPA
5. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.
6. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes. The online IMS system will switch from using the active (A or B) copy of the ACBLIB to the inactive (A or B) copy.
7. Delete, define and initialize all of the DEDB AREA data sets with the new system definitions.
8. Enter the normal /START DATABASE and /START AREA commands to make the DEDB and its AREAs accessible to the active IMS system.
9. Use a specific user-developed application program or OEM utility to reload the DEDB through the change system definitions for the DEDB.
10. On the first access to the newly changed DEDB, the application will pseudoabend and the PSB will be rescheduled. Message DFS2834I will be displayed.

    The transaction will be tried again for both IFPs and MPPs when the PSB is rescheduled. If the application attempts to access the DEDB before commit processing has completed, an 'FH' status will be returned to the application. The DEDB is inaccessible because the randomizer for the DEDB is unloaded by the /DBR command.

If either database level or area level changes are made to DEDBs while a BMP or DBCTL thread is active, then commit processing fails and the message DFS3452 is issued.

Related information:

→ DFS3452 (Messages and Codes)

Online change and DEDB randomizer and exit routines

Randomizing routines determine the location of database records by AREA within the DEDB and by root anchor point (RAP) within the AREA. A change of the DEDB randomizer is a database level change.

A new randomizing routine affects the location (AREA and RAP) of every database record within the DEDB. The randomizer is defined for the DEDB in the DBD parameter: RMNAME=.

A randomizer change can involve introducing a brand new randomizer into the active IMS system or changing an existing randomizer in use by one or more DEDBs.

The name of the randomizer is specified in the DBD parameter: RMNAME=. If a new randomizer is introduced for an existing DEDB, a DBDGEN and ACBGEN of the database with the new randomizer name is required in addition to the following procedural steps.

To introduce a new DEDB randomizing routine using online change:

1. Use a specific customer-developed application program or original equipment manufacturer (OEM) utility to unload the DEDB with the current randomizer.
2. Assemble and link edit the new randomizer into the IMS SDFSRESL or one of the libraries in the STEPLIB concatenation.
3. Run a DBDGEN for the DEDB with the new randomizer designated in the DBD parameter: RMNAME=.
4. ACBGEN is also needed to build the application control blocks to implement the DEDB definition that includes the new randomizer. The changed or new application control blocks must be built into the active IMS system's staging copy of ACBLIB, which is offline.
5. Run the online change utility, DFSUOCU0, to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.
6. Remove access to the DEDB from the active IMS system and to unload the randomizer by entering the type-1 /DBR DB operator command sequence or the type-2 UPDATE DB STOP(ACCESS) operator command sequence. Do not use the OPTION(NORAND) parameter when you issue the UPDATE DB STOP(ACCESS) type-2 command, because the NORAND parameter prevents the randomizer from unloading.
7. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.
8. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes. The online IMS system will switch from using the active (A or B) copy of the ACBLIB to the inactive (A or B) copy.
9. Delete, define and initialize all of the DEDB AREA data sets with the new system definitions.
10. Make the DEDB and its areas accessible to the active IMS system by entering the type-1 commands /START DATABASE and /START AREA or the type-2 command UPDATE DB START(ACCESS) AREA(*).
11. Use a specific customer-developed application program or OEM utility to reload the DEDB with the new randomizing routine in effect.

Changing a DEDB randomizing routine using online change:

If a change is made to a randomizer already in use by one or more DEDBs, then all of the DEDBs using the subject randomizer must be included in the change process.

The changed randomizer will not be introduced if an existing version is already loaded for any DEDB in the active IMS system. You can determine that the existing version is no longer used by locating the keyword GONE in message DFS2838I. Also, you can determine that the randomizer module is brought from any library to the storage by locating the keyword LOADED in the message DFS2842I.

Changing DEDB randomizers requires the procedures described below. Because the name of the randomizer remains the same, DBDGEN, ACBGEN and the online change command sequence are not applicable.

1. Use a specific customer-developed application program or OEM utility to unload the DEDB with the existing randomizer. This should be done for all of the DEDBs that use the randomizer to be changed.
2. Remove access to the DEDBs from the active IMS system by entering the type-1 command `/DBR DATABASE` or the type-2 command `UPDATE DB STOP(ACCESS)`. The `/DBR DATABASE` command unloads the randomizer for the DEDBs designated as operands. The `UPDATE DB STOP(ACCESS)` command also unloads the randomizer for the DEDBs that are designated as operands, unless the OPTION(NORAND) parameter is specified. When all of the DEDBs that reference the randomizer are stopped, the randomizer is removed from the active IMS system. If a DEDB is not stopped and it references a randomizer that has been removed from the IMS system, then the next DL/I call results in a U1021 abend.
3. Assemble and link edit the changed randomizer into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.
4. Delete, define and initialize all of the DEDB AREA data sets to prepare for reloading the DEDB with the changed randomizer.
5. Resume access to the DEDBs that use the changed randomizer by issuing the type-1 command `/START DATABASE` or the type-2 command `UPDATE DB START(ACCESS)`. For DEDBs, the `/START DATABASE` command or the `UPDATE DB START(ACCESS)` command causes the randomizer to be loaded. To resume access to all of the areas at the same time, you can issue the type-2 command `UPDATE DB START(ACCESS) AREA(*)`.
6. Use a specific customer-developed application program or OEM utility to reload the DEDB with the changed randomizing routine in effect.

Deleting a DEDB randomizing routine using online change:

After all of the DEDBs that use the old randomizing routine are unloaded and had either the type-1 command /DBR or the type-2 command UPDATE DB STOP(ACCESS) run successfully against them, then the randomizing routine can be deleted.

To delete a randomizing routine from the active IMS system, follow the steps in "Online change and DEDB randomizer and exit routines" on page 799. Message DFS2838 is generated when the randomizing routine is deleted.

If you use the type-2 command `UPDATE DB STOP(ACCESS)`, do not include the OPTION(NORAND) parameter, because the NORAND parameter prevents the DEDB randomizing routine from unloading from storage.

Customers with data sharing IMS systems that do not share SDFSRESLs must be careful to delete the randomizing routine from both systems.

Online change and DEDB segment compression routines:

Segment compression routines are segment specific and are defined for the DEDB in the DBD SEGM parameter ("COMPRTN=").

Adding, changing or deleting segment compression routines is procedurally the same and involves the same restrictions as DEDB randomizing routines.

Online change and DEDB Data Capture exit routines:

Data Capture exit routines can be defined for the DEDB on the DBD statement, for a specific segment on the SEGM statement ("EXIT="), or for both.

Multiple exit routines can be specified on a single DBD or SEGM statement.

Adding a new DEDB Data Capture exit routine using online change:

You can add a new Data Capture exit routine to a DEDB by using the online change function.

To add a new Data Capture exit routine, follow the procedure below:
1. Assemble and link edit the new exit routine into the IMS.SDFSRESL or one of the libraries in the IMS.SDFSRESL STEPLIB concatenation.
2. Run a DBDGEN for the DEDB with the new exit routine designated in the DBD or SEGM parameter: "EXIT=".
3. ACBGEN is also needed to build the application control blocks to implement the DEDB definition that includes the new exit routine. The changed or new application control blocks must be built into the active IMS system's staging copy of ACBLIB, which is offline.
4. Run the Online Change Copy utility, DFSUOCU0, to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.
5. Enter the normal /DBR command sequence to remove access to the DEDB from the active IMS system.
6. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.
7. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes. The online IMS system will switch from using the active (A or B) copy of the ACBLIB to the inactive (A or B) copy.
8. Enter the normal `/START DATABASE` and `/START AREA` commands to make the DEDB and its areas accessible to the active IMS system.

Changing an existing DEDB Data Capture exit routine using online change:

You can change an existing DEDB Data Capture exit routine by using the online change function.

To change an existing Data Capture exit routine, follow these steps:
1. Allow the dependent regions that are accessing DEDBs with the particular Data Capture exit to end normally.

2. Assemble and link edit the changed exit routine into the IMS SDFSRESL or one of the libraries of the IMS SDFSRESL STEPLIB concatenation.
3. Start the dependent regions. Data Capture exits are loaded at dependent region initialization time, so the new version of the exit will take effect when the region is started. Data Capture exit routines that were linked as reentrant or reusable are unloaded at dependent region termination time. Otherwise, they are unloaded after every DL/I call.

Deleting a DEDB Data Capture exit routine using online change:

You can delete a DEDB Data Capture exit routine by using the online change function.

To delete a Data Capture exit routine, execute the following steps:
1. Run a DBDGEN for the DEDB with the old exit routine omitted from the DBD or SEGM statement.
2. Build the application control blocks by using the Application Control Block Maintenance utility to implement the DEDB definition that excludes the old exit routine. The changed or new application control blocks must be built into the active IMS system's staging copy of ACBLIB, which is offline.
3. Run the Online Change Copy utility, DFSUOCU0, to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.
4. Enter the normal /DBR command sequence to remove access to the DEDB from the active IMS system.
5. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.
6. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes. The online IMS system will switch from using the active (A or B) copy of the ACBLIB to the inactive (A or B) copy.
7. Enter the normal /START DATABASE and /START AREA commands to make the DEDB and its areas accessible to the active IMS system.

Changing DEDB root addressable space with two stage randomizing routine using online change:

You can change the DEDB root addressable space with two stage randomizing routine by using online change.

The UOW structure and root addressable allocation is specific to each area within each DEDB. However, a change to the number of root addressable CIs within one area can affect the number of root anchor points within the whole DEDB. If the DEDB uses a standard randomizing routine that randomly distributes database records across the entire database, changes to the root addressable allocation are *Database Level* changes and procedurally must be handled as such. This topic is not applicable to such changes.

If, however, a two-stage randomizing routine is used for the DEDB, a change to an individual area UOW root addressable definition is an *AREA Level* change. A two-stage randomizing routine does not attempt to evenly distribute database records across all areas based on the total number of root anchor points in the entire DEDB. A two-stage randomizing routine is designated in the DBDGEN by coding the randomizing routine name as follows:

```
RMNAME=(mmmmmmmm,2)
```

In prior releases of IMS, customers would get the following error message if a DEDB DBD had more than one operand in the RMNAME parameter:

```
8, DBD130 - RMNAME OPERAND IS OMITTED OR INVALID
```

The same message will appear for this release of IMS if anything but a two is specified as the second operand of RMNAME. Customers can still specify RMNAME=(mmmmmmmm) for standard randomizing routines.

Changing the DEDB AREA UOW structural definition using online change:

Changing the DEDB AREA UOW structural definition requires the following of several procedural steps.

To change the DEDB AREA UOW structural definition follow these steps:

1. Use a specific customer-developed application program or original equipment manufacturer (OEM) utility to unload the area through existing system definitions.
2. DBDGEN, PSBGEN and ACBGEN to generate the application control blocks to implement the DEDB structural changes. The "UOW=(x,y)" parameter on the AREA DBDGEN macro statement defines the amount of space allocated to overflow within a DEDB UOW. The "ROOT=(nnn,mmm)" parameter on the AREA DBDGEN macro statement defines the amount of space allocated to Independent Overflow. The changed or new application control blocks must be built into the active IMS system's staging copy of ACBLIB, which is offline.
3. Run the online change utility, DFSUOCU0, to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.
4. Enter the /DBR AREA command to remove access to the area from the active IMS system.
5. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.
6. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes.
7. Delete, define and initialize the area with the new system definitions.
8. Enter the /START AREA command to make the area accessible to the active IMS system.
9. Use a specific customer-developed application program or OEM utility to reload the DEDB through the changed system definitions for the DEDB.

Related tasks:

"Online change and DEDB dependent and independent overflow space allocation" on page 807

Making online changes at the DEDB and area level
You can make online changes to DEDB databases at either the database level or the area level.

Adding or deleting DEDB databases using online change:

You can add or delete a DEDB database by using the online change function.

The following figure shows the overall process for adding a database using online change.
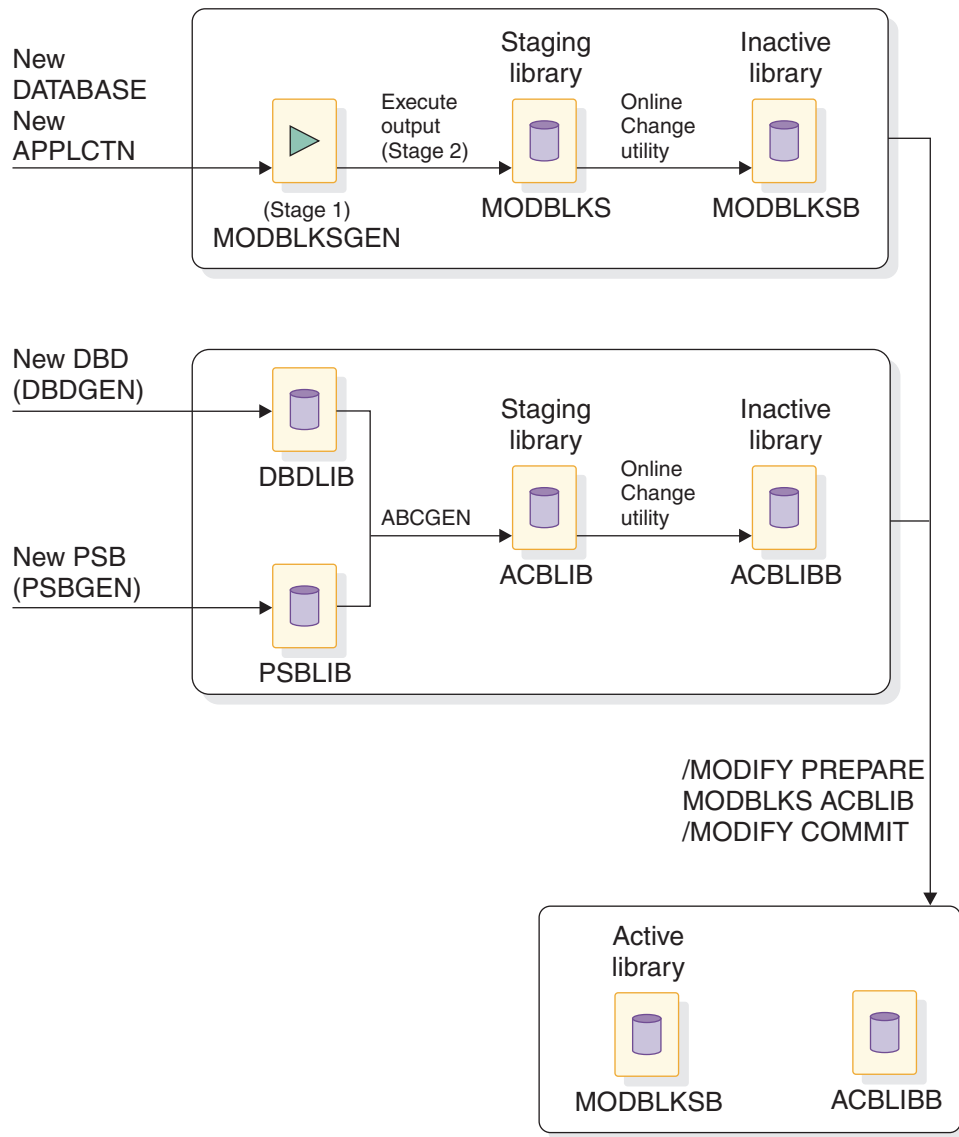
Figure 289. Adding a database using online change

Adding or deleting a DEDB and implementing the change by means of the IMS online change facility requires that you perform the following steps.

1. MODBLKS level system definition (Stage 1 and Stage 2) to add or delete the DEDB. The changes for the IMS.MODBLKS data set should be generated into the active IMS system's staging copy of the IMS.MODBLKS data set, which is offline.

2. DBDGEN, PSBGEN and ACBGEN to generate the application control blocks to add or delete the DEDB and PSBs that access it. The changed or new application control blocks must be generated into the active IMS system's staging copy of the IMS.ACBLIB data set, which is offline.

3. Run the online change utility, DFSUOCU0, to move the changed IMS.MODBLKS data set and IMS.ACBLIB data set from the staging libraries to the inactive (A or B) copies of these libraries that are online to the active IMS system.

4. Enter and follow the online change command sequence for PREPARE processing. If a DEDB is being added to an IMS system that does not have Fast Path installed, the DFS2833 error message will appear and the PREPARE process will be aborted.

   If a DEDB is added whose areas have CI sizes that exceed the system buffer size (BSIZ=), message DFS2832 appears and the PREPARE process aborts.

   Finally, if a DEDB is added to an IMS system that was initialized without any DEDBs, message DFS2837 appears and the PREPARE process aborts.

   Output threads are initialized during Fast Path initialization only if DEDBs are currently generated in the system. In order for the user to be able to add DEDBs with online change, IMS must be initialized with DEDBs to begin with.

5. If the DEDB is to be deleted, any BMP region or DBCTL thread scheduled for access to the DEDB must first be stopped. Full function transactions scheduled for access to the DEDB will be placed in a QSTOP state and as a result, MPP or IFP dependent regions need not be stopped to implement the online change to delete the DEDB.

6. If the DEDB is to be deleted, access to it from the active IMS system must be removed by means of a /DBR DB command. The commit will fail with a DFS3452 message if the DEDB has not had the /DBR command successfully run against it beforehand.

7. Execute the online change command sequence for COMMIT/ABORT processing.

8. If the DEDB is newly added, execute the following additional steps at any appropriate time prior to making the DEDB generally available for normal user access:

   a. Execute the normal procedures for defining the new DEDB and its areas and area data sets to DBRC and the RECON data sets.

   b. Define and initialize all of the area data sets belonging to the new DEDB.

   c. Execute the procedures to include the required Dynamic Allocation definitions that will enable the DEDB and its areas to be allocated to the active IMS system. Or register the DEDB and its areas to DBRC, and DBRC will dynamically allocate them during IMS initialization.

   d. Enter the /START DATABASE and /START AREA commands to make the DEDB and its areas accessible to the active IMS system.

   e. Run the necessary application load programs.

Adding or deleting DEDB segments using online change:

You can add or delete a DEDB segment by using the online change function.

Adding or deleting segment types or changing segment formats affects the structure of a DEDB and constitutes a Database Level change. The addition or deletion of segment types (including the DEDB Sequential Dependent Segment type) affects the hierarchical structure and the segment prefix layout to implement this structure. Similarly, the change of individual segment formats changes the structure of the entire database and space allocations within each AREA of the DEDB.

To make structural changes to an existing DEDB, execute the procedural steps described below.

1. Use a specific customer-developed application program or OEM utility to unload the DEDB through existing system definitions.

2. DBDGEN, PSBGEN and ACBGEN to generate the application control blocks to implement the DEDB structural changes. The changed or new application control blocks must be built into the active IMS system staging copy of ACBLIB, which is offline.

3. Run the Online Change Copy utility (DFSUOCU0) to move the changed ACBLIB from the staging ACBLIB to the inactive (A or B) copy of the ACBLIB that is online to the active IMS system.

4. Enter the normal /DBR command sequence to remove access to the DEDB from the active IMS system. This command may be issued any time prior to the /MODIFY COMMIT.

5. Enter and follow the online change command sequence for PREPARE processing for ACBLIB changes.

6. Enter and follow the online change command sequence for COMMIT/ABORT processing for ACBLIB changes.

7. Delete, define and initialize all of the AREA data sets belonging to the DEDB with the new system definitions.

8. Enter the normal /START DATABASE and /START AREA commands to make the DEDB and its areas accessible to the active IMS system.

9. Use a specific customer-developed application program or OEM utility to reload the DEDB through the changed system definitions for the DEDB.

Adding or deleting DEDB areas using online change:

Adding or deleting an area can affect the location of every database record throughout the DEDB. Changing the number of areas will alter the number of root anchor points (RAPs) within the DEDB.

DEDB randomizing routines attempt to randomly distribute database records throughout the entire DEDB based first on the area and then on the root anchor point (RAP) within the area.

Adding or deleting one or more areas to a DEDB constitutes a structural change such as adding a segment type. The steps described in "Adding or deleting DEDB segments using online change" on page 805 should be followed to change the number of areas defined in the DEDB. If areas are newly added, the required DBRC definitions for areas and area data sets must be processed and dynamic allocation blocks must be prepared before the new areas can be accessed by the active IMS system.

Changing DEDB root addressable space allocation using online change:

There are different implications depending on whether you randomly distribute DEDB records or use a standard randomizing routine to evenly distribute DEDB records. In either case, you can distribute DEDB records across an entire DEDB or just a single DEDB area.

Random distribution of database records across all DEDB areas

Changes to the DEDB unit of work (UOW) structure that affect the number of DEDB Control Intervals defined to the Root Addressable portion impact the number of root anchor points within the entire DEDB. This type of change potentially affects the location of every database record within the DEDB.

Standard DEDB randomizing routines

Standard DEDB randomizing routines attempt to evenly distribute database records across all areas and within the selected AREA. Such randomizing routines determine the record location based on the total number of root anchor points in the entire DEDB.

A change to the UOW structure that changes the number of CIs defined to the root addressable area constitutes Database Level change when a standard DEDB randomizing routine is used. This type of change should be treated the same as a DEDB structural change in terms of online change procedures.

Online change and DEDB dependent and independent overflow space allocation:

Fast Path has provides limited support for extending DEDB AREA Independent Overflow space allocation. Additionally, DEDB online change will allow changes to the overflow space allocation both within each UOW (Dependent Overflow) and outside the root addressable portion (Independent Overflow) of the AREA.

Both Dependent and Independent Overflow changes are considered to be Area-level changes. However, such changes must not alter the number of CIs defined to the root addressable portion. Changing the number of root addressable CIs will change the number of root anchor points and could affect the DEDB randomizing routine in locating database records.

Changing DEDB AREA overflow allocation requires the same procedural steps as those defined for changing the root addressable area.

Related tasks:

"Changing the DEDB AREA UOW structural definition using online change" on page 803

Changing DEDB CI size using online change:

DEDB online change can be used to change DEDB AREA control interval size. However, CI size changes must not alter the number of CIs allocated to the root addressable portion of an AREA because this could affect the DEDB randomizing routine in locating database records across the DEDB.

The SIZE= parameter on the AREA statement of DBDGEN defines the CI size of the data set that constitutes the AREA.

Extending DEDB independent overflow online

You can extend the independent overflow (IOVF) portion of a DEDB area while IMS is online.

The first time the area is opened after this procedure is completed, a message is issued to verify that Fast Path recognizes and accepts the change to the area and normal open processing completes. You can also modify the IOVF portion of a DEDB using DEDB online change.

You cannot decrease the size of the IOVF with this procedure. However, the size of the sequential dependent part might increase or decrease depending on the total amount of space allocated to the area. The steps in this procedure also reorganize the area.

To increase the size of the IOVF portion of a DEDB online:

1. Run the DBDGEN utility to obtain an updated DBD. Update the *number2* and *overflow2* operands on the ROOT= keyword of the AREA statement. See AREA statement (System Utilities).

   All other control statements must remain identical to those on the existing DBD. Changing other control statements might damage data and create unpredictable results.

2. Run the ACBGEN utility using the updated DBD. You should run PSB=ALL to create a new and complete ACBLIB with the new ROOT= parameters. The output should be a different data set from the one currently used by the control region. The new ACBLIB is identical to the old ACBLIB, except for the ROOT= changes. You can use the staging ACBLIB, but do not switch with the online change function.

3. Ensure that the area is in good condition. The area must not have any in-doubts, and must not be in a recovery-needed condition. Also, at least one copy of the area (one area data set) must have no error queue elements (EQEs). Use the `/DIS AREA` command to display EQEs and the condition. Use the `/DIS CCTL INDOUBT` command to display all in-doubt threads. Eliminate potential defects before continuing to the next step so that data is not lost or damaged.

4. Process SDEPs using the SDEP scan and delete utilities. This step is required because the IOVF extension procedure requires an unload and load of the area. Some unload and load utilities are unable to process SDEPs. Unload/load utilities that do process SDEPs might reload them in root order rather than time order, which can interfere with subsequent SDEP scan and delete operations.

5. If multiple copies of the area (MADS) exist, stop all copies of the area except one using the `/STOP ADS` command. Ensure that the remaining copy does not have any EQEs and is not in a recovery-needed condition. Multiple ADSs must be stopped to ensure that DBRC has accurate information when the area is brought online after the IOVF is extended.

6. Issue a `/DBR` or `/STO AREA` command against the area.

7. Take an image copy of the area.

8. If the area is registered with DBRC, set the recovery-needed flag on for the area. This flag is required by the DEDB Initialization utility and can be set using a `CHANGE.DBDS RECOV` command.

9. Unload the area.

10. Execute the IDCAMS utility to delete and redefine the data set. The amount of space you allocate for the area in the Define procedure should reflect the increased size of the IOVF. The number of SDEP CIs in the area might change because this number represents the difference between the total amount of space allocated to the area and the amount used by the other parts. These other parts are the root addressable part, the IOVF, the reorganization UOW, and two control CIs.

    Related Reading: See *z/OS DFSMS Access Method Services for Catalogs* for a description of the IDCAMS Delete and Define functions.

11. Execute the Fast Path initialization utility against the new area using the new ACBLIB.

12. Issue the `/START AREA` command to bring the area online.

13. Reload the area.

Recommendation: Reload the area in batch. If you reload the area using a BMP, the BMP might fail with message DFS3709A and reason code 5. If this failure occurs, issue the `CHANGE.DBDS` command to set ICOFF and restart the BMP.

14. Take an image copy of the area after the reload.

When the area is next accessed, message DFS3703I is issued. This message alerts you that discrepancies were found during open processing. However, open processing continues because the discrepancies indicate to IMS that you used an accepted procedure to increase the size of the IOVF. DFS3703I is not issued during subsequent opens of the area as long as IMS remains online. DFS3703I is also issued by any sharing subsystem the first time the area is opened on that subsystem after the IOVF is extended.

During emergency restart or extended recovery facility (XRF) takeover, the updated area information is picked up from the log. Therefore, DFS3703I is not issued.

Use the new ACBLIB for any subsequent normal restarts of the online system. Ensure that the new ACBLIB reflects *only* the changes made to the ROOT= keyword. Any other changes you make might cause damage to the area. If you do not use the new ACBLIB, open logic allows the discrepancy between information from the old ACBLIB and information from the area data set, but issues message DFS3703I each time the discrepancy is encountered.

Note: You cannot use the online change function to update the ACBLIB with the altered ROOT= parameter.

Related concepts:

"DEDB area design guidelines" on page 460

➡ DBRC administration (System Administration)

Related reference:

➡ Application Control Blocks Maintenance utility (System Utilities)

➡ Database Description (DBD) Generation utility (System Utilities)

➡ DEDB Sequential Dependent Scan utility (DBFUMSC0) (Database Utilities)

➡ Data Entry Database Sequential Dependent Scan utility exit routine (DBFUMSE1) (Exit Routines)

Modifying HALDB databases

You can modify existing HALDB databases just as you can modify any other full-function database; however, you must perform some modification tasks differently for HALDB and other modification tasks are unique to HALDB.

You should also familiarize yourself with the many unique HALDB concepts and characteristics before performing any of the modification tasks described in this topic.

Related concepts:

"Converting HDAM and HIDAM databases to HALDB" on page 848

"Implementing HALDB design" on page 541

"Options for offline reorganization of HALDB databases" on page 665

Overview of modifying HALDB databases

Many of the changes that you can make to HALDB databases are the same as those that you can make to other IMS full-function database types. However, some changes are unique to HALDB databases, such as adjusting the distribution of records across the partitions in a HALDB database.

You specify most changes to HALDB databases in either the database definition (DBD) statements that define the HALDB master database or in the DBRC RECON data set where the partition definitions are stored. Where you specify changes depends on the type of change you are making. For example, to change the hierarchical structure of records you specify the changes in the DBD using the DBDGEN process; to modify the range of records stored in a partition you specify the change in the DBRC RECON data set using either the Partition Definition utility or batch DBRC commands.

The scope of HALDB database modifications

When you modify HALDB databases, you need to be aware of the scope of your changes. Some of the changes that you can make affect only one partition or a subset of partitions. Other changes to HALDB databases affect the entire database and all of its partitions.

The following table provides a list of typical HALDB database changes, where you specify the change, and the scope of the change.

Table 84. Examples of HALDB database changes, where to specify them, and their scope

What you are modifying	Where you specify the change	The scope of the change	Notes
Field definition, adding or deleting	DBD statements	Entire HALDB database and its partitions	The HALDB database must be unloaded only if you are changing field offsets or the length of a segment
Data capture exit routine	DBD statements	Entire HALDB database and its partitions	Does not require an unload of the HALDB database
Segment type, adding or deleting	DBD statements	Entire HALDB database and its partitions	The HALDB database must be unloaded
Pointer options	DBD statements	Entire HALDB database and its partitions	The HALDB database must be unloaded
Segment Edit/Compression exit routine	DBD statements	Entire HALDB database and its partitions	The HALDB database must be unloaded
Selection criteria of a HALDB partition selection exit routine	DBD statements and the RECON data set	Only the partitions that are impacted by the change in the distribution of records by the partition selection exit routine	You must unload only the partitions that are impacted by the change in record distribution

Table 84. Examples of HALDB database changes, where to specify them, and their scope (continued)

What you are modifying	Where you specify the change	The scope of the change	Notes
Data set name prefix	The RECON data set	Only the specified partition	Only the affected partition is unauthorized, access to all other partitions is unaffected
Randomizing module or randomizing parameters	DBD statements, the RECON data set, or both	If made in the DBD, all partitions in the database. If made in the RECON data set, only the specified partition	For changes to the specifications in the RECON data set, only the affected partition is unauthorized, access to all other partitions is unaffected
High keys of partitions	The RECON data set	All partitions affected by the change in record distribution introduced by the new or changed high key	You must unload only the partitions affected by the change in record distribution

Changes that affect all of the partitions in a HALDB database:

Certain characteristics of HALDB partitions are shared by all of the partitions in a HALDB database.

Before you can change these characteristics, you must take all of the partitions in the HALDB database offline by issuing either the type-1 command /DBRECOVERY DB *HALDB_master_name* or the type-2 command UPDATE DB NAME(*HALDB_master_name*) STOP(ACCESS).

You can change the following characteristics of HALDB partitions only after you take offline all of the partitions the HALDB database:
- DBD definition
- A HALDB partition selection exit routine
- Share level
- Nonrecoverable attribute status
- Maximum size of OSAM data sets for PHDAM and PHIDAM databases

Note: In addition to taking the partitions in the HALDB database offline, some of the changes listed above require you to unload all of the partitions, make the applicable change, reload the partitions, and then restore access to the partitions.

For example, suppose a HALDB database has an existing HALDB partition selection exit routine that needs to be replaced with a HALDB partition selection exit routine that selects partitions based on a new algorithm. This change requires the entire HALDB database to be offline because partition selection exit routines can affect the placement of records in every partition in the database. After taking the database offline, you must unload the database, make your changes to the exit routine, reload the database, and then restore access to the database.

Changes you can apply to a single partition:

Certain characteristics of HALDB databases are specific to each partition in the
database. To change these characteristics, you do not need to take the entire
HALDB database offline; you need to take offline only the partition in which you
are making changes.

Before making changes to a partition, issue either the type-1 command
/DBRECOVERY DB *partition_name* or the type-2 command UPDATE DB
NAME(*partition_name*) STOP(ACCESS) against the partition.

For example, you can change the following characteristics of a HALDB partition by
taking only the single partition offline:
- Data set name prefix
- Randomizing module name
- Number of root anchor points (RAPs)
- Bytes parameter
- OSAM block size
- VSAM CI size

Note: In addition to taking the partition offline, the changes listed above require
you to unload the partition, make the applicable change, reload the partition, and
then restore access to the partition.

Record distribution and partition boundaries in HALDB databases

You can adjust the distribution of records in your HALDB database by redefining
the range of records that a partition holds.

You redefine the range of records for a partition by changing the boundaries
between the partitions. Because the number of records that a partition contains can
change over time, you would typically adjust the distribution of records in a
HALDB database when one or more partitions in the HALDB database become too
large or too small.

How you change partition boundaries depends on how your HALDB database
performs partition selection. HALDB databases perform partition selection by
using either partition high keys, which are based on the root keys of the records,
or a partition selection exit routine, which is based on a user-defined partition
selection string.

If your HALDB database selects partitions based on high keys, you change the
range of records in a partition by changing its high key.

If your HALDB database selects partitions using a partition selection exit routine,
you change the range of records in a partition by modifying the exit routine itself.
These modifications are typically installation-specific and therefore are not
documented in this topic.

Changes to partition boundaries can affect one or more partitions. If a change to
one partition causes records to be moved to or from any other partitions, the
change also affects those other partitions. If you use high keys for partition
selection, IMS automatically sets the initialization-required flag for the partitions
that are affected by a boundary change. If you use a HALDB partition selection
exit routine, you must flag the partitions that are affected by a boundary change as

requiring initialization. In either case, before you change the partition boundaries, you must issue the /DBR command against all of the partitions that will be affected by the change.

Adding, disabling, and deleting partitions from a HALDB database can also affect the distribution of records.

Related concepts:

"Disabling and enabling HALDB partitions" on page 824

Related tasks:

"Adding partitions to an existing HALDB database" on page 820

"Deleting partitions from an existing HALDB database" on page 827

Record distribution and high key partitioning:

If you use key range partitioning, you can change the distribution of records across HALDB partitions by changing the high key of one or more partitions. The high key of a partition specifies the highest root key of a record that a partition can contain.

The high key of a partition also defines the boundary between the partition that owns the high key and the partition with the next higher high key, if such a partition exists. If you change a high key that defines a boundary between two partitions, both partitions are affected by the change. For example, lowering a high key of a partition usually reduces the number of records in the partition that owns the high key and increases the number of records in the partition with the next higher high key.

Before you adjust the distribution of records in your HALDB database:

* Analyze the current distribution of records across the partitions in the HALDB database.

  For an accurate report of the records in an existing HALDB database, you can use the IBM IMS HALDB Toolkit for z/OS, a separately licensed software tool. For more information about the tool, see the information about IMS Tools in the IBM Knowledge Center at www.ibm.com/support/knowledgecenter..

* Enable the HALDB reorganization number verification function, which prevents the reorganization numbers of partitions from being regressed by the movement of records between partitions.

You can change the high key of a partition by using either the Partition Definition utility or by using DBRC batch commands. These changes generally require that you unload the affected partitions before changing the partition definitions, and then initialize and reload the partitions when you are done making changes. An affected partition is any partition whose range of keys has increased or decreased, even if you did not directly change the definition of the partition.

You typically change the high key of a partition when the amount of data in a HALDB database is not balanced from one partition to the next. Changing a high key of a partition changes the boundary of the partition and causes some records to be moved from one partition to another.

The following figure shows the change of the high key for partition B from 400 to 500, which results in the movement of records with keys from 401 to 500 from partition C to partition B.
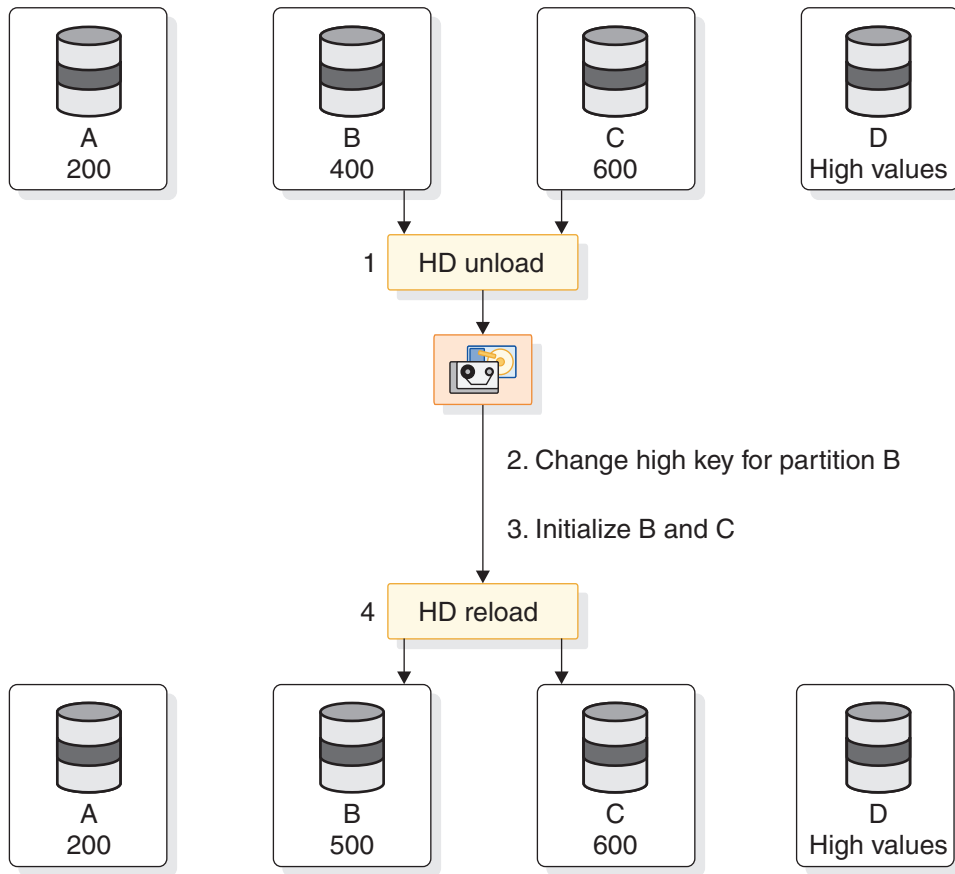
Figure 290. Changing a high key with key range partitioning

The change to the high key of partition B in the previous figure is therefore a change that affects both partition B and partition C. When the definition of partition B is changed, IMS flags both partition B and partition C as requiring initialization. Partitions A and D are not affected by the change.

Note that the online IMS system is not aware of the changes to the partition definitions of partitions B and C in the previous figure until one of the following events occurs:

- A /START DB *HALDB_Master* OPEN command is issued.
- An UPDATE DB NAME(*HALDB_Master*) START(ACCESS) OPTION(OPEN) command is issued.
- A DL/I call causes an authorization call to DBRC for partition B or C. The first DL/I call goes through HALDB partition selection to select and authorize either partition B or C.

Related concepts:

"HALDB partition reorganization numbers" on page 183

Related tasks:

"Changing the high key of a partition" on page 819

Record distribution and the partition selection exit routine:

If you use a partition selection exit routine, the distribution of records across partitions is managed by the exit routine, not by IMS.

To adjust the distribution of records across partitions, you must modify the exit routine.

IMS cannot detect changes in the exit routine to the distribution of records. Nor does IMS know which partitions are affected by those changes. Consequently, you must understand which partitions are affected by any changes you make and initialize them accordingly.

When a change causes records to be moved from one partition to another, you must run the HD Reorganization Unload utility (DFSURGU0), set the *partition-initialization-required* (PINIT) flags, initialize, and run the HD Reorganization Reload utility (DFSURGL0) for every partition that is affected by the movement of records.

You should also ensure that any partition selection exit routine that you are using contains code that processes the structure initialization and structure modification calls that IMS issues when updating online partition structures after partitions are added or modified. The sample Partition Selection exit routine (DFSPSE00) provided with IMS includes code that supports these calls.

Related tasks:

"Adding or changing a HALDB partition selection exit routine" on page 836

Related reference:

➡ HALDB Partition Selection exit routine (DFSPSE00) (Exit Routines)

Partition definition control blocks and partition definitions in the RECON data set

The online IMS system creates an internal partition definition control block for each HALDB partition that you define.

Any time you change HALDB partition definitions in the RECON data set, IMS must rebuild the partition definition control blocks of all the partitions that are affected by the change before the partitions can be used.

To detect when IMS needs to rebuild a partition definition control block, IMS tracks changes to HALDB databases using a *change version number*. Each time you change the partition definitions recorded in the RECON data set, DBRC increases the change version number for the HALDB master database by one. When the online IMS system detects an increase in the change version number for a HALDB database, IMS rebuilds the partition definition control block of each partition affected by the change.

Online change is not used to change HALDB partition definitions or rebuild partition definition control blocks. IMS recognizes any increases in HALDB change version numbers and dynamically reflects the new definitions in the online IMS system.

If you are using XRF, the alternate IMS system sees the dynamic change and automatically updates the definitions in the alternate system.

Related concepts:

"HALDB partition names and numbers" on page 182

Events that trigger a rebuild of partition definition control blocks:

IMS checks the HALDB change version numbers to detect changes to partition definitions.

IMS rebuilds the internal partition definition control blocks, if necessary, in the following events:

- When a partitions are authorized for use, which detects changes in existing partitions or in partitions that are not already authorized. This situation occurs commonly when a partition is taken offline, changes are made to its definition in the RECON data set, and the partition is made available again. The first use of the updated partition triggers the rebuild of the partition definition control blocks.
- When an invalid key is detected by partition selection or by a partition selection exit routine, such as when a new partition is added beyond the high key of the last partition and all existing partitions are already authorized. In this case, IMS partition selection or the partition selection exit routine detects the new partition and IMS rebuilds the partition definition control blocks automatically.
- When a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command is issued. For example, if a new partition is added beyond the high key of the last partition and all existing partitions are already authorized, these commands initiate a rebuild of the partition definition control blocks.

Related concepts:
"Additional considerations for the partition definition control blocks"

Additional considerations for the partition definition control blocks:

When making changes to HALDB partition definitions, until the online partition definition control blocks are rebuilt, a number of circumstances could occur that you should be aware of.

Consider the following points:

- If you use a HALDB partition selection exit routine, you must issue either the /DBRECOVERY command or the UPDATE STOP(ACCESS) command and then, after making the changes to the partition definition, the /START command or the UPDATE START (ACCESS) command. When the HALDB partition selection exit routine selects a HALDB partition, IMS is not aware of HALDB partition boundaries until the internal partition definition control blocks are finished being rebuilt.
- If you are using a HALDB partition selection exit routine and IMS notifies you of a structure modification, you might need the exit routine to select partitions correctly based on the current partition structure.
- A /START DB *partition_name* OPEN command might fail after the definition of a partition structure is changed because the corresponding partition definition control block needs to be rebuilt. To invoke a rebuild of the control block, an application program that uses the partition must be run or the type-1 command /START DB *HALDB_Master* OPEN must be issued.
- Newly added partitions are not known by the online IMS system until a rebuild of the corresponding partition definition control blocks has been invoked and the new control blocks have been created.
- If you are using a partition selection exit routine, be sure that it contains code that processes the structure initialization and structure modification calls that IMS issues when updating partition definition control blocks after partitions are

added or modified. The sample Partition Selection exit routine (DFSPSE00) delivered with IMS includes code that supports these calls.

Related concepts:

"Events that trigger a rebuild of partition definition control blocks" on page 816

How IMS assigns partition ID numbers

IMS assigns a partition ID number to each partition when the partition is defined.

The partition ID of each new partition is generated by incrementing the partition ID number of the last partition defined to IMS. Therefore, each new partition in a HALDB database is assigned a new higher number. The last partition ID that is assigned is stored in the RECON data set. Previously assigned partition ID numbers are not reused.

If you delete the definition of a partition, the partition ID of that partition is permanently lost. If you try to restore the partition by redefining it, the redefined partition will have a new higher partition ID.

The assignment of new partition IDs when deleting and redefining partitions has important implications for your ability to back out of such changes. Partition IDs are physically stored in partition data sets. Consequently, data sets that are created for one partition cannot be used with other partitions or with the same partition if it is deleted and redefined. Similarly, image copies cannot be used if the partition ID in the image copies does not match the partition ID of the partition you are applying them to.

Recommendation: Disable partitions before you delete them. Disabled partitions are removed from active use, but can be re-enabled. Re-enabled partitions retain their original partition ID. Disabled partitions can be deleted later, after their removal from the active HALDB database has been thoroughly tested.

The following three figures illustrate the assignment of partition IDs in a HALDB database that has three partitions.

The following figure shows a HALDB database with three partitions: A, B, and C. The three partitions have partition IDs of 001, 002, and 003.

HALDB database
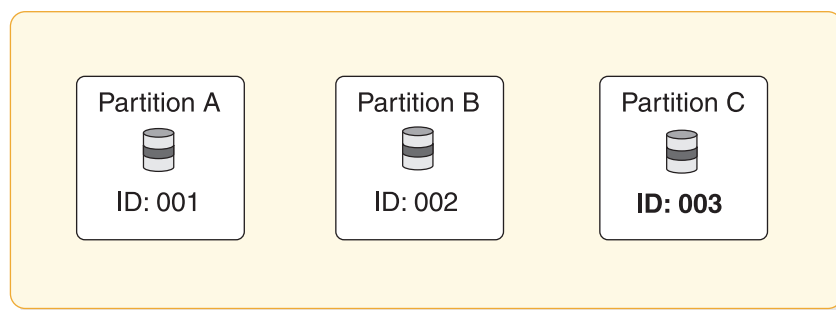
Figure 291. Partition IDs prior to deleting and redefining a partition

The following figure shows the same HALDB database after partition C is deleted. Deleting partition C requires unloading both partitions B and C, deleting partition C, initializing partition B, and then reloading the records that were in both

partition B and C into partition B. This changes the database from three to two partitions.

HALDB database
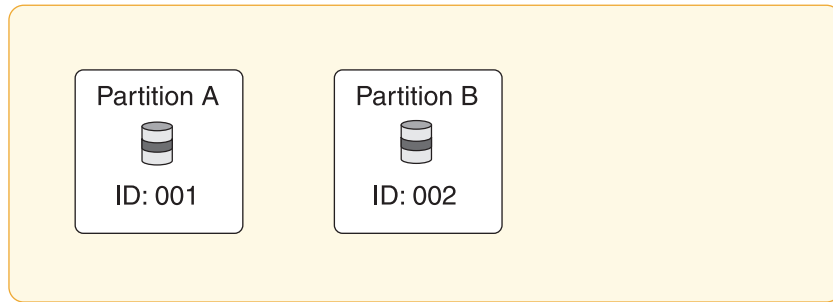
Figure 292. Partition IDs after deleting, but before redefining a partition

The following figure illustrates why you cannot recover partition C after redefining it: because the original partition ID of partition C was permanently lost when the partition was deleted. When partition C is redefined, IMS assigns a partition ID of 004, which was obtained by incrementing the previously assigned partition ID number. Because partition C now has partition ID 004, image copies of the partition C data sets taken before the partition was deleted cannot be used. These image copies contain partition ID 003.

HALDB database
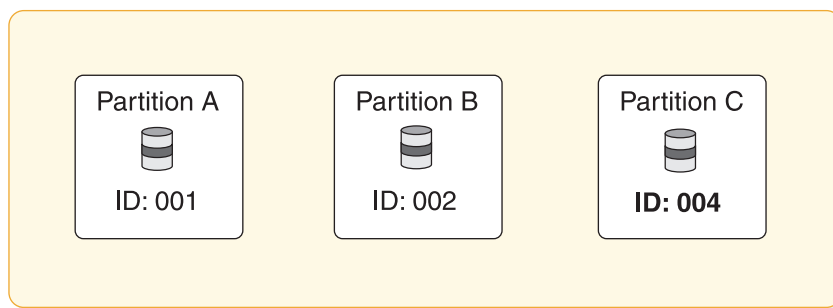
Figure 293. Partition IDs after deleting and redefining a partition

Related concepts:
"Disabling and enabling HALDB partitions" on page 824
Related tasks:
"Deleting partitions from an existing HALDB database" on page 827
"Restoring deleted HALDB partitions" on page 831

Automatic update of HALDB secondary index and logical relationship pointers

When an application program updates a HALDB database that has logical relationships or secondary indexes, IMS automatically corrects the pointers for you by using the HALDB self-healing pointer process.

You do not need to update the pointers used in the logical relationships nor do you need to rebuild your secondary indexes.

For example, when you make changes to an indexed database that do not require changes to secondary index definitions, you do not need to unload, reload, or rebuild the secondary indexes. The HALDB self-healing pointer process adjusts the pointers individually when the pointers are used. The reload process for the indexed database updates the ILDSs. Nothing else needs to be done to ensure that the secondary index pointers can be used to find the moved target segments.

Also, you can change the partitions of a logically related database without making any changes to other logically related databases. When you add, delete, or modify partitions in a database, you do not need to unload and reload any logically related database. The self-healing pointer process of HALDB adjusts pointers individually when they are used.

Related concepts:

"The HALDB self-healing pointer process" on page 691

Online change and HALDB databases

DBD changes for HALDB databases can be implemented in online systems with online change.

The database cannot be in use during the online change process. You should stop access to the entire HALDB database by issuing either a /DBRECOVERY DB *HALDB_master_name* command or an UPDATE DB NAME(*HALDB_master_name*) STOP(ACCESS) command.

After the DBD change is committed, you can resume access to the database by issuing either a /START DB *HALDB_master_name* command or a UPDATE DB NAME(*HALDB_master_name*) START(ACCESS) command. DBD changes affect every partition, so the master database name must be used for these commands.

Restriction: If you delete a database name from IMS using online change, you cannot reuse that name for another HALDB partition without first performing a cold start. Similarly, if you delete a partition name from the RECON data set, you cannot use online change to reuse that name for another database without first performing a cold start.

Related concepts:

"Online change and HALDB alter" on page 761

Related tasks:

"Activating database changes by using the online change function" on page 795

Changing the high key of a partition

You can change the high key of a partition to either increase or decrease the number of records the partition contains.

Recommendation: Prior to changing the high keys of partitions in your HALDB database, enable the HALDB reorganization number verification function. The HALDB reorganization number verification function protects the integrity of data when records are moved by a change in a high key or by database reorganizations.

To change the high key of a partition:

1. For all partitions that will be affected by the change to the high key, issue either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.
2. Unload all partitions that will be affected by the change to the high key.
3. Change the high key definition for the partition by using either the HALDB Partition Definition utility (%DFSHALDB) or DBRC commands.
4. Initialize all partitions that are affected by the change by running either of the following utilities:
   - HALDB Partition Data Set Initialization utility (DFSUPNT0)
   - Database Prereorganization utility (DFSURPR0)
5. Reload all partitions that are affected by the change by using the output of step 2. The image copy needed flag is set for the data sets in the reloaded partitions.
6. Create image copies of the data sets in the affected partitions. DBRC does not allow updates to the data after the image copy required flag is set and before the image copies have been recorded in the RECON data sets.
7. Make the partitions available again by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

An online IMS system is not aware of the changes to partition definitions in the RECON data set until one of the following events occurs:
- A /START DB *HALDB_Master* OPEN command is issued.
- An UPDATE DB NAME(*HALDB_Master*) START(ACCESS) OPTION(OPEN) command is issued.
- A DL/I call causes an authorization call to DBRC for a partition affected by the change. The first DL/I call goes through HALDB partition selection again to select and authorize the correct partition.

Related concepts:

"Record distribution and high key partitioning" on page 813

"HALDB partition reorganization numbers" on page 183

Adding partitions to an existing HALDB database

You can add partitions to your existing HALDB databases. Typically, you do this when a partition grows too large. Adding a partition usually causes records in an existing partition to be moved to the new partition.

To add a new partition:
- Determine whether your HALDB database performs partition selection using high keys or a partition selection exit routine. If your HALDB database is using a partition selection exit routine you must take extra steps when adding a partition.
- Identify all existing partitions that contain records that will be redistributed to the new partition. You must unload, initialize, and reload these partitions to add the new partition.
- After you add the partition, ensure that the online IMS system recognizes the new partition and updates its online database structures. IMS rebuilds its online structures when any of the following events occur:
  - A /START DB *HALDB_Master* OPEN command is issued.
  - An UPDATE DB NAME(*HALDB_Master*) START(ACCESS) OPTION(OPEN) command is issued.

- A DL/I call causes an authorization call to DBRC for a partition affected by the change. The first DL/I call goes through HALDB partition selection again to properly select and authorize the correct partition.

In the following figure, partition D is added to a database that uses high keys for partition selection. The high key of partition D is 300. Previously defined partitions A, B, and C had high keys of 200, 400, and high values. The addition of the new partition requires the movement of records with keys above 200 and up to 300 from partition B to partition D. This means that partition B is affected by the change. When partition D is defined with a high key of 300, IMS sets the PINIT flag for partitions B and D. Partition initialization will initialize these two partitions. Partitions A and C are not affected by the change.
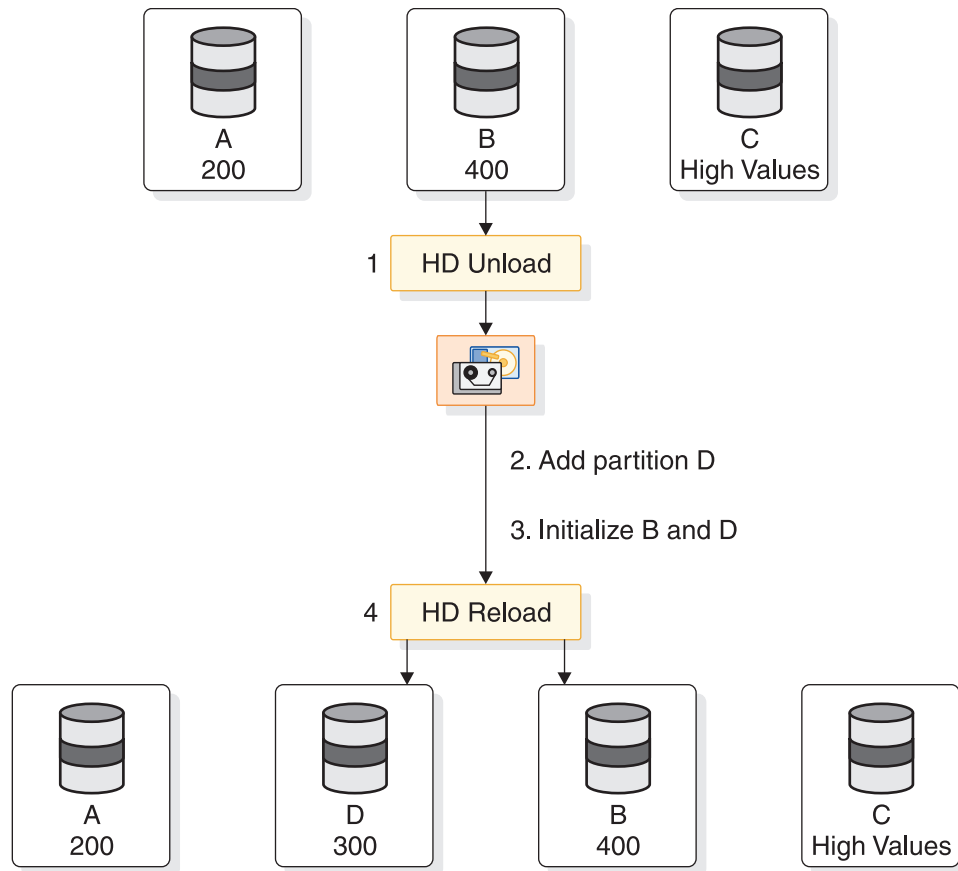
Figure 294. Adding a partition with key range partitioning

Related concepts:

"Record distribution and partition boundaries in HALDB databases" on page 812

Related tasks:

"Restoring deleted HALDB partitions" on page 831

Adding a partition to a HALDB database that uses high key partition selection

Unless you use a partition selection exit routine, your HALDB database uses high keys for partition selection. IMS automatically sets the partition initialization (PINIT) flag for new partitions you add, as well as for any existing partitions affected by the redistribution of records to the new partition.

To add a partition to a HALDB database that uses high keys for partition selection:

1. For all partitions that contain records that will be redistributed, issue either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.

2. Unload all partitions that contain records that will be redistributed to the new partition.

3. Define the new partition with the Partition Definition utility or DBRC commands.

4. Allocate the data sets for the new partition.

5. Initialize the new partition and the existing partitions affected by the redistribution of records to the new partition.

6. Reload all partitions using the output of step 1. The image copy needed flag is set for the reloaded data sets.

7. Make the partitions available again by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

8. If the online IMS system does not immediately recognize the new partition, issue either a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command to rebuild all of the online partition structures of the HALDB database specified in the command.

Adding a partition that defines a new highest high key

You might add a partition that has a higher high key than any existing partition, if the keys in your HALDB database are based on time or on an ascending value. As the key values increase, you can add new partitions to hold new records.

To add a partition that will define a new highest high key for your HALDB database:

1. If records from any other partition will be moved into the new partition, stop access to the partitions from which records will be moved by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.

2. If records from any other partition will be moved into the new partition, unload the partitions from which records will be moved.

3. Define the new partition, specifying an appropriate high key, such as X'FF'.

4. If necessary, modify the high key of the partition that previously had the highest high key to avoid having two partitions with a high key of X'FF'.

5. Initialize the new partition and any partitions from which records were moved.

6. If necessary, load the new partition and any partitions from which records were moved. The image copy needed flag is set for the data sets in the new partition.

7. Make the partitions available again by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

8. If the online IMS system does not immediately recognize the new partition, issue either a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command to rebuild all of the online partition structures of the HALDB database specified in the command.

The following figure illustrates the addition of a partition that defines a new highest high key for a HALDB database. The key for the database is based on time. The high order part of the key contains the year. There is a partition for each

year. A new partition is required for each new year. There are no records with keys above 2007zzz. Before they are added to the database, partition D with a high key of 2008zzz is added. Because no records are moved from an existing partition, they are not affected.

1. Add partition D

2. Initialize partition D
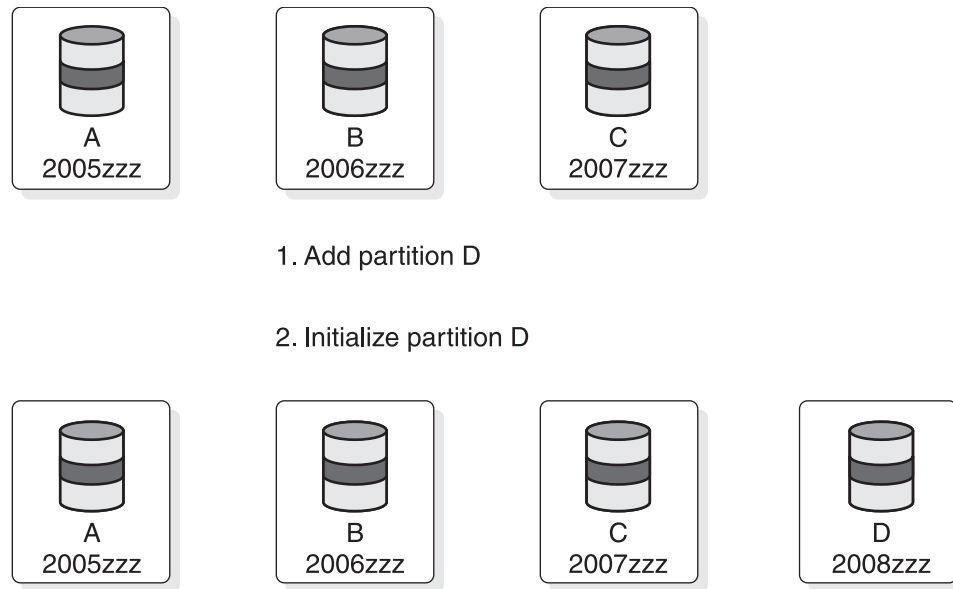
Figure 295. Adding a higher key partition with key range partitioning

Adding a partition to a HALDB database that uses a partition selection exit routine

When you add a partition to a HALDB database that uses a partition selection exit routine, you must manually set the PINIT flag in any partition from which records will be moved to the new partition.

You might also need to modify the exit routine to recognize the new partition, as well as change the partition selection criteria used by the exit routine.

When you use a partition selection exit routine, IMS cannot know which existing partitions are affected by the addition of the partition. Consequently, IMS does not set the PINIT flag for existing partitions that are affected by the redistribution of records to the new partition. You must set the PINIT flag for the existing partitions.

You should also be sure that any partition selection exit routine that you use contains code that processes the structure initialization and structure modification calls that IMS issues when it updates online partition structures after partitions are added or modified. The sample Partition Selection exit routine (DFSPSE00) delivered with IMS includes code that supports these calls.

To add a new partition to a HALDB database that uses a partition selection exit routine:

1. Stop access to the database as appropriate:
   - If you need to modify the partition selection exit routine, stop access to the HALDB master database by issuing either the `/DBRECOVERY DB`
     `HALDB_master_name` command or the `UPDATE DB NAME(HALDB_master_name)`
     `STOP(ACCESS)` command. Stopping access to the HALDB master unloads the existing partition selection exit routine from storage.

- If you do not need to modify the partition selection exit routine, stop access only to those partitions from which records will be moved to the new partition by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.

2. If necessary, modify the partition selection exit routine to use the new partition. The exit routine should be able to process the

3. Unload the partition or partitions from which records will be moved.

4. Define the new partition. IMS sets the PINIT flag for the new partition.

5. Set the PINIT flag in the appropriate existing partitions.

6. Initialize the partitions that have the PINIT flag set.

7. Load the new partition and any partition from which records were moved.

8. Take an image copy of the reloaded data sets. The image copy needed flag is set for the data sets in the partitions that are initialized or loaded by HD Reload.

9. Resume access to the database as appropriate:
   - If you stopped the entire database, resume access by issuing either the /START DB *HALDB_master_name* command or the UPDATE DB NAME(*HALDB_master_name*) START(ACCESS) command. Stopping access to the HALDB master unloads the existing partition selection exit routine from storage.
   - If you stopped only selected partitions, resume access by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

10. If you did not stop access to the entire database and the online IMS system does not immediately recognize the new partition, issue either a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command to rebuild all of the online partition structures of the HALDB database specified in the command.

For more information about the sample HALDB Partition Selection exit routine (DFSPSE00), see *IMS Version 15 Exit Routines*.

Disabling and enabling HALDB partitions

You can temporarily remove partitions from an active HALDB database by *disabling* the partition.

Disabled partitions appear to be deleted from the HALDB database and are unavailable for use by IMS and most utilities; however all of the records related to the partition, such as the partition ID number and image copy records, are retained by DBRC in the RECON data set.

You would normally disable a partition that you plan to delete. Disabling the partition allows you to test the proposed deletion without actually deleting the partition and the recovery information associated with it. After you disable a partition, you can delete it or restore the partition to the HALDB database by enabling and recovering the partition.

Related concepts:

"Record distribution and partition boundaries in HALDB databases" on page 812

"How IMS assigns partition ID numbers" on page 817

About disabled partitions

In a RECON listing, the flag PARTITION DISABLED=YES identifies a partition as disabled. The HALDB Partition Definition utility identifies a disabled partition by showing its status as DISABLED.

In most situations, a disabled partition is not known to IMS and is treated as if it is not registered in the RECON data set. However, you can still view all information about a disabled partition by using either the DBRC LIST command or the HALDB Partition Definition utility (%DFSHALDB). A disabled partition still counts towards the maximum number of partitions that can be defined for a HALDB database.

Disabled partitions are not removed from the DBRC groups CAGROUP, DBDSGRP, DBGROUP and RECOVGRP; however, disabled partitions are generally not processed as part of a group when the group is used in a DBRC command. This is also true of implied groups used in a command. An exception to this is that changes are accumulated for disabled partitions that are members of a change accumulation group when you issue the GENJCL.CA command and execute the IMS Database Change Accumulation utility.

The GENJCL.IC and GENJCL.RECOV commands fail for disabled partitions. If the commands specify a group of any kind, disabled partitions are simply skipped, JCL is not generated and no message is issued.

The following code is an example of disabling a partition by using the DBRC CHANGE.PART command:

```
//CHGPART JOB
...
...
//SYSIN DD *
CHANGE.PART DBD(DB3) PART(PART3) DISABLE
/*
```

Disabling HALDB partitions

Disabling a partition removes it from an active HALDB database while retaining all of the partition records in the RECON data set. Disabled partitions can be enabled later.

To disable a HALDB partition:

1. Take an image copy of the partition that you intend to disable and any partitions into which records will be moved from the disabled partition.
2. Stop access to the partition you are disabling and the other affected partitions by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.
3. Unload the partition you are disabling and the other affected partitions.
4. Disable the partition by using one of the following methods:
   - The HALDB Partition Definition utility
   - The DBRC command CHANGE.PART DBD(*HALDB_master_name*) PART(*partition_name*) DISABLE
5. If the HALDB database uses high-key partitioning, ensure that the high key values defined to DBRC for each partition are still appropriate.
6. Initialize the remaining partitions affected by the change.
7. Reload the remaining partitions affected by the change.

8. Take an image copy of the reloaded data sets. The image copy needed flag is set for the data sets in the partitions that are initialized or loaded by the HD Reorganization Reload utility (DFSURGL0).
9. Issue the `/START DB` *partition_name* command or the `UPDATE DB NAME(`*partition_name*`) START(ACCESS)` command to make the affected partitions available again.

About enabling partitions

Enabling a disabled partition makes it available for use by IMS again.

The procedure for enabling a partition for use in a HALDB database is essentially the same as the procedure for adding a new partition to a HALDB database. You can enable a disabled partition by issuing the DBRC command `CHANGE.PART` with the ENABLE parameter.

When you enable a HALDB partition, you can either enable the partition in conjunction with the unload and reload processes or you can restore the HALDB database to its state prior to the partition being disabled by using the recovery process.

If updates have been made to the database after the partition was disabled, you must use a time stamp recovery to recover the HALDB database. A time stamp recovery typically requires you to recover the entire HALDB database and all of its partitions, as well as any related HALDB databases.

When you enable a partition, DBRC flags the partition as needing recovery. If the HALDB database uses a partition selection exit routine, DBRC flags all of the partitions in the database as needing partition initialization. If the HALDB database uses high key partitioning, DBRC flags the partition with the next higher high key value as needing partition initialization.

Enabling HALDB partitions

Enabling a disabled partition makes it available for use by IMS again.

To enable a HALDB partition in conjunction with the unload and reload processes:
1. For all partitions that contain records that will be redistributed after the partition is enabled, issue either the `/DBRECOVERY DB` *partition_name* command or the `UPDATE DB NAME(`*partition_name*`) STOP(ACCESS)` command.
2. Unload the offline partitions.
3. Enable the partition by using one of the following methods:
   - The HALDB Partition Definition utility
   - The DBRC command `CHANGE.PART DBD(`*HALDB_master_name*`) PART(`*partition_name*`) ENABLE`

   DBRC marks all of the database data sets in the enabled partition as needing recovery.
4. If the HALDB database uses high-key partitioning, ensure that the high key values defined to DBRC for each partition are appropriate.
5. Allocate the data sets for the enabled partition and any other affected partitions.
6. Initialize the offline partitions.
7. Reload the partitions. The image copy needed flag is set for the data sets.
8. Take an image copy of the reloaded partitions.

9. Make the partitions available again by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

10. If the online IMS system does not immediately recognize the enabled partition, issue either a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command to rebuild all of the online partition structures of the HALDB database.

Recovering HALDB databases when enabling partitions

When a partition is enabled, it is flagged in the RECON data set as needing recovery. When recovering an enabled partition, you must also recover any partitions that were affected when the partition was disabled.

If updates were made to the HALDB database after the partition was disabled, you will likely have to recover the entire HALDB database, as well as any other HALDB databases that are logically related to the HALDB database that contains the partition being enabled.

To enable a partition as part of recovery:

1. Enable the partition by using one of the following methods:
   - The HALDB Partition Definition utility
   - The DBRC command CHANGE.PART DBD(*HALDB_master_name*) PART(*partition_name*) ENABLE

   DBRC marks all of the database data sets in the enabled partition as needing recovery.

2. By using the image copies taken before the partition was disabled, recover all of the partitions that changed when the partition was disabled or that were changed by applications after the partition was disabled.

   If updates were made to the databases after the partition was disabled, you must perform a time stamp recovery.

3. Make the partitions available again by issuing either the /START DB *partition_name* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) command.

4. If the online IMS system does not immediately recognize the enabled partition, issue either a /START DB *HALDB_Master* OPEN or UPDATE DB NAME(*HALDB_Master*) OPTION(OPEN) command to rebuild all of the online partition structures of the HALDB database.

Related concepts:

"Recovery of databases" on page 606

Deleting partitions from an existing HALDB database

You can delete partitions from an existing HALDB database. You do this typically when a partition has little data. Deleting a partition usually involves moving its records to another partition.

Prior to deleting a partition, you should:

1. Determine whether your HALDB database performs partition selection using high keys or a partition selection exit routine. If your HALDB database is using a partition selection exit routine you must take extra steps when you delete a partition.

2. Identify all other partitions that will receive records previously stored in the deleted partition. You must unload, initialize, and reload these partitions in addition to the deleted partition.

3. Determine if the HALDB database uses a secondary index (PSINDEX). You might need to rebuild the PSINDEX after you delete a partition.

4. Make an image copy of the partition.

5. Export the partition definition of the partition you are going to delete by using the HALDB Partition Definition utility (%DFSHALDB).

6. Disable the partitions that you intend to delete. Disabling a partition effectively removes the partition from the HALDB database without deleting any information about the partition from the RECON data set. The partition definition, including the partition ID number, are retained in the RECON data set. If you need to restore the partition, you can easily enable it again and any prior image copies are still usable. After you are certain you will not need to restore the partition, you can then delete the partition record from the RECON data set.

The following figure shows the deletion of a partition from a HALDB database that performs partition selection using high keys. The HALDB database has partitions A, B, C, and D. Partition B, with a high key of 400, is deleted. The records in partition B move to partition C, meaning that partition C is affected by the deletion of partition B. When the definition of partition B is deleted, IMS sets the PINIT flag for partition C. Partition initialization initializes partition C. Partitions A and D are not affected by the change.
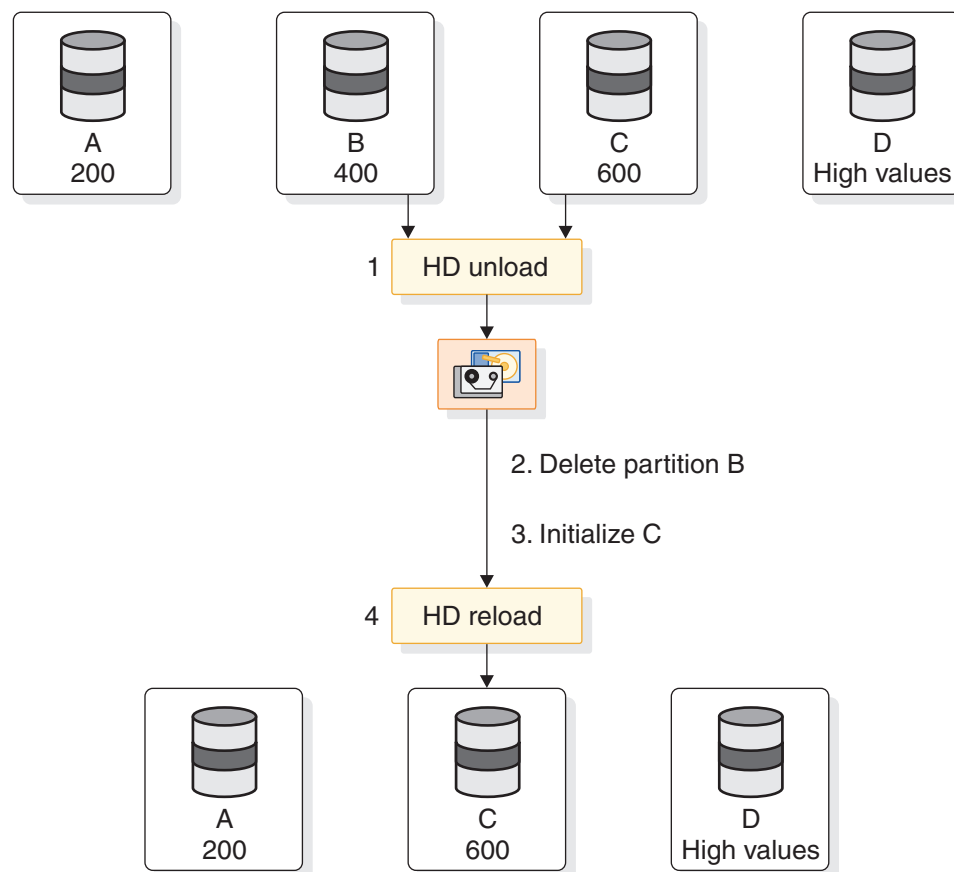
Figure 296. Deleting a partition with key range partitioning

Deleting a partition from a HALDB database that uses high-key partitioning

Deleting a partition permanently removes a partition and any records that it might contain from both the HALDB database. The records associated with the deleted partition are also permanently removed from the RECON data set.

To delete a partition from a HALDB database that selects partitions using high keys:

1. Take an image copy of the partition you intend to delete and any partitions in to which records from the deleted partition will be moved.

2. Stop access to the partition you are deleting and the other affected partitions by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.

3. Unload both the partition you are deleting and the other partitions affected by the deletion.

4. Delete the definition of the partition from the RECON data set by using one of the following methods:
   - The HALDB Partition Definition utility
   - The DBRC command DELETE.PART DBD(*HALDB_master_name*) PART(*partition_name*)

5. Ensure that the high key values defined in the RECON data set for each partition are still appropriate.

6. Initialize the remaining partitions that will contain the records of the deleted partition.

7. Reload the remaining partitions using the output of step 1.

8. Take an image copy of the reloaded partitions. The image copy needed flag is set for the data sets of the reloaded partition.

9. Issue the /START DB *partition_name* command or the UPDATE DB NAME(*partition_name*) START(ACCESS) command to make the affected partitions available again.

After you complete the deletion process, IMS issues message DFS0415W with reason code 90 for the deleted partition when IMS rebuilds the online partition structures for the HALDB database.

Deleting the partition with the lowest key and all of its records

In some cases, it might be useful to delete a partition that has the lowest key range along with all of the records the partition contains.

Such a case might arise if the keys in a HALDB database are based on time or an ascending value. As the key values increase, you can delete a partition that has become empty or whose records are no longer of interest.

For example, in the following figure the key for the database is based on time. The high order part of the key contains the year and there is a partition for each year. In this example, partition A with a high key of 2005zzz is no longer needed. Either all of its records have been deleted or they are no longer needed. In fact, this is an efficient way to delete all the records in partition A. Because no records are moved either to or from any other partitions, no other partitions are affected. No

partitions need to be unloaded, reloaded, or initialized. There is only one step in the process: to delete partition A.

1. Delete partition A
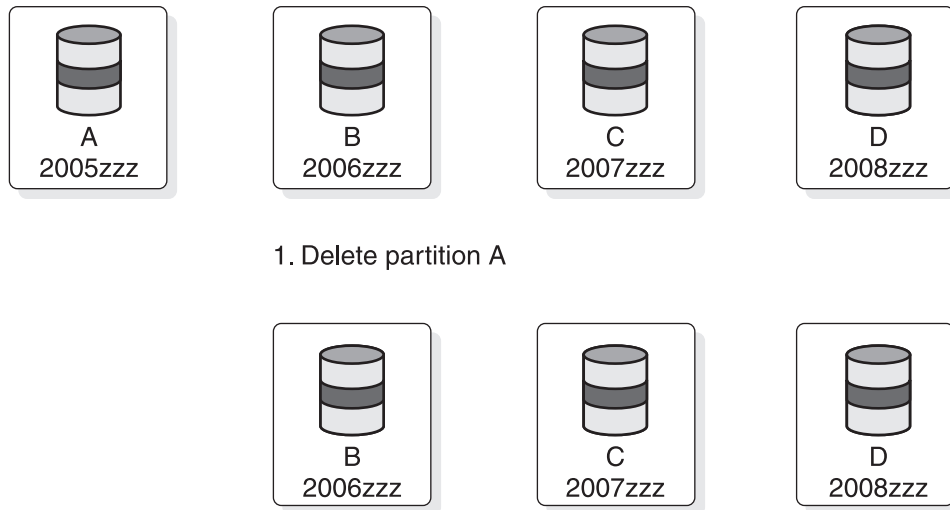
Figure 297. Deleting the partition with the lowest key in a HALDB that uses key range partitioning

Considerations for deleting a partition from a HALDB database that uses a secondary index

If there a HALDB database uses a secondary index, you must be careful when you delete a partition.

If the secondary index has entries that point to the deleted partition, the secondary index becomes invalid. You must rebuild it with a tool such as the IMS Index Builder. If the secondary index does not have entries that point to the deleted partition, the secondary index is not affected and does not need to be rebuilt. If there are no records in the deleted partition, the secondary index does not need to be rebuilt. The secondary index is unaffected by the deletion of the partition.

Deleting a partition from a HALDB database that uses a partition selection exit routine

When you use a partition selection exit routine, IMS cannot know which partitions are affected by the deletion of a partition.

Consequently, IMS does not set the PINIT flag for the partitions affected by the redistribution of records from the deleted partition. You must set the PINIT flag for these partitions manually.

You might also need to modify the exit routine to recognize the new partition, as well as change partition strings in some partitions.

To delete a partition from a HALDB database that uses a partition selection exit routine:

1. Take offline both the partition you are going to delete and any partition into which the records from the deleted partition will be moved by issuing the `/DBRECOVERY DB` *partition_name* command or the `UPDATE DB NAME(`*partition_name*`) STOP(ACCESS)` command.

2. Unload both the partition you are going to delete and any partition into which the records from the deleted partition will be moved by running the HD Reorganization Unload utility (DFSURGU0).

3. Delete the definition of the partition you are deleting from the RECON data set.

4. If necessary, change the partition selection exit routine.

5. Change the partition strings for partitions, as necessary.

6. Set the PINIT flag in the partitions that will receive records from the deleted partition.

7. Initialize the partitions that have the PINIT flag set.

8. Load the partitions by running the HD Reorganization Reload utility (DFSURGL0).

9. Take an image copy of the reloaded data sets. The image copy needed flag is set for the data sets in the partitions that are initialized or loaded by the DFSURGL0 utility.

10. Make the affected partitions available again by issuing the /START DB *partition_name* command or the UPDATE DB NAME(*partition_name*) START(ACCESS) command.

Restoring deleted HALDB partitions

The way in which you restore a deleted HALDB partition depends on whether or not you took certain precautions before you deleted the partition.

If you made image copies and used the HALDB Partition Definition utility (%DFSHALDB) to export the partition definitions, including their partition IDs, before you deleted the partition, you can restore a deleted partition by importing the partition definition and applying the image copies.

To import only the deleted partition by using the HALDB Partition Definition utility, specify Try all partitions as the processing option in the Import a Database panel when you import the partition definition. Only the definition of the deleted partition is imported. Any definitions for partitions that were not deleted, are not imported. The restored partition has the same partition ID that it had before it was deleted, which allows you to use the image copies of the partition data sets that you made before you deleted the partition.

If you did not make image copies and export the partition definitions before you deleted the partition, instead of restoring the deleted partition, you can define a new partition to hold the records previously held by the deleted partition. However, in this case, the new partition has a different partition ID than the deleted partition and you cannot use any existing image copies of the deleted partition.

Related concepts:

"How IMS assigns partition ID numbers" on page 817

Related tasks:

"Adding partitions to an existing HALDB database" on page 820

Changing the name of a HALDB partition

You can change the name of HALDB partition by deleting the partition and redefining it with a new name.

However, deleting and redefining a partition has a number of consequences that you should be aware of:

- The partition ID number changes.
- The image copy records for the partition are deleted from the RECON data sets.
- The ALLOC records for the partition are deleted from the RECON data sets.

Note: Using the online change function, a HALDB partition name that has been deleted from the RECON data set cannot be reused by the online change function to name a database without also performing a cold start of IMS. The opposite is also true: a database name that has been deleted from IMS cannot be reused to name a HALDB partition without also performing a cold start.

To change the name of a HALDB partition:
- Take the partition offline by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.
- Unload the partition.
- Delete the partition definition from the RECON data set.
- Redefine the partition in the RECON data set with the new name.
- Initialize the partition.
- Reload the partition.
-
    Make the partition available by issuing the /START DB *partition_name* command or the UPDATE DB NAME(*partition_name*) START(ACCESS) command.

After you complete the name change process, when IMS rebuilds the online partition structures for the HALDB database it issues message DFS0415W with reason code 90 for the deleted partition.

Related concepts:
"How IMS assigns partition ID numbers" on page 817
Related tasks:
"Changing databases dynamically in online systems" on page 791

Modifying the number of root anchor points in a PHDAM partition

If a PHDAM partition has many roots that randomize to the same root anchor point (RAP), lock contention problems that negatively impact performance can occur. To remedy this problem, you can increase the number of RAPs in the partition.

The modify the number of RAPs in a PHDAM partition:
1. Take the partition offline by issuing either the /DBRECOVERY DB *partition_name* command or the UPDATE DB NAME(*partition_name*) STOP(ACCESS) command.
2. Unload the data from the partition.
3. Change the number of RAPs in the partition by using either the HALDB Partition Definition utility (%DFSHALDB) or DBRC commands.
4. Initialize the partition.
5. Reload the partition.
6. Take an image copy of the data sets for the partition.
7. Make the partition available by issuing either the /START DB *partition_name* command or the UPDATE DB NAME(*partition_name*) START(ACCESS) command.

Modifications to HALDB record segments

Changes to segments and segment hierarchies in a HALDB database are made in the DBD statements of the DBDGEN process.

Most changes to segments and segment hierarchies in a HALDB database require you to first unload all of the partitions in the HALDB database and reload the partitions after the changes are complete.

The following list includes typical changes you can make to HALDB segments and segment hierarchies and whether or not they require a complete unload of the HALDB database:

- Adding a field definition that does not increase the size of a segment does not require an unload of the database.
- Deleting a field definition that does not decrease the size of a segment and that is not used with a secondary index does not require an unload of the database.
- Adding or deleting a segment type requires an unload of the database.
- Changing the definition of a segment size requires an unload of the database.
- Changing the definition of a sequence field requires an unload of the database.
- Changing the location of a segment in the hierarchy requires an unload of the database.
- Changing pointer options requires an unload of the database.

Related concepts:

"Modifying record segments" on page 721

Modifying HALDB partition data sets

In a HALDB database, each partition has its own data sets. Changes to the database data sets of one partition do not affect the database data sets in any other partitions.

Moreover, only the partition in which you are making the change needs to be unavailable for processing. All other partitions in the HALDB database can remain available.

Before making any of the following changes to HALDB partition data sets, you must stop access to the partition by issuing a /DBRECOVERY DB *partition_name* command or a UPDATE DB NAME(*partition_name*) STOP(ACCESS) command against the partition. All other partitions in the HALDB database can remain available.

Changing HALDB data set name prefixes

The data set name prefix for each partition is stored in the RECON data set. You can change it using either the HALDB Partition Definition utility (%DFSHALDB) or the DSNPREFX(*string*) parameter of the DBRC batch command CHANGE.PART.

If you change a data set name prefix, the partition initialization needed (PINIT) flag is set for the partition.

Changing the free space parameters for a partition data set

The free space specifications for a HALDB partition data set are stored in the RECON data set.

You can change them using either the HALDB Partition Definition utility (%DFSHALDB) or by using the free space parameters FBFF(*value*) or FSPF(*value*) of the DBRC batch command CHANGE.PART.

You cannot change the free space parameters of a partition while the partition is being reorganized by the integrated HALDB Online Reorganization function, as indicated by OLREORG CURSOR ACTIVE = YES in the record in the RECON data set, or while the partition is owned by an IMS system for the integrated HALDB Online Reorganization process.

Changing the OSAM block size for a HALDB database data set

The OSAM block size for a HALDB database data set is stored in the RECON data set.

You can change it using either the HALDB Partition Definition utility (%DFSHALDB) or the BLOCKSZE(*nnnnn*) parameter of the DBRC batch command CHANGE.PART. After you change the OSAM block size of a data set, the partition initialization needed (PINIT) flag is set for the partition.

Changing the VSAM CI size for a HALDB database data set

You can change VSAM CI sizes for a HALDB database data set only by using z/OS DFSMS commands. After you change the VSAM CI size, the partition initialization needed (PINIT) flag is set for the partition.

Because VSAM CI sizes are not stored in the DBRC RECON data set, you cannot change them using the HALDB Partition Definition utility (%DFSHALDB) or the DBRC CHANGE commands.

For information about the z/OS DFSMS commands used for VSAM data sets, see *z/OS DFSMS Access Method Services for Catalogs*.

The maximum size of OSAM data sets and HALDB databases

You set the maximum size of the OSAM PHDAM and PHIDAM data sets for a HALDB database in the RECON data set.

The maximum size of OSAM PHDAM and PHIDAM data sets for a HALDB database can be 4 or 8 gigabytes. When a HALDB database supports 8-GB OSAM data sets, the HALDB online reorganization (OLR) function cannot be used to reorganize the database. Instead, use offline utilities for database reorganization or use 4-GB OSAM data sets, which supports OLR.

Defining the maximum size of OSAM data sets for a new HALDB database

When you register a new HALDB master database with DBRC, you can specify the maximum size of OSAM PHDAM or PHIDAM data sets.

A HALDB database can support up to 4 or 8 gigabytes of data in OSAM PHDAM or PHIDAM database data sets.

1. Issue the DBRC batch command INIT.DB DBD(*master_HALDB_name*) TYPHALDB.
2. Unless you want the default of 4-GB OSAM data sets and the ability to use the HALDB online reorganization (OLR) function, include these keywords on the command:
   - For 8-GB OSAM data sets, specify OLRNOCAP and OSAM8G.

- For 4-GB OSAM data sets that cannot be reorganized online, specify OLRNOCAP and NOOSAM8G.

The HALDB database is registered with DBRC, and the master database record indicates whether the HALDB database supports 4-GB or 8-GB OSAM data sets.

Changing the maximum OSAM data set size for a HALDB from 4 GB to 8 GB

Use the CHANGE.DB command to increase the maximum size of OSAM PHDAM and PHIDAM data sets for a HALDB database from 4 to 8 GB.

To increase the maximum size of HALDB OSAM data sets to 8 GB:

1. If the database is open in an online IMS system, stop access to the database by issuing either the /DBRECOVERY DB *haldb_master* command or the UPDATE DB NAME(*haldb_master*) STOP(ACCESS) command.
2. If HALDB online reorganization has been run against the database, ensure that all of the database partitions use A through J data sets.
3. Issue the CHANGE.DB command with the OSAM8G keyword. If the database has been reorganized online, you must also specify the OLRNOCAP keyword.
4. Make the partitions available by issuing either the /START DB *haldb_master* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) OPTION(OPEN) command.

The master database record in the RECON data set is updated to indicate that the HALDB supports 8-GB OSAM data sets.

Changing the maximum OSAM data set size for a HALDB from 8 GB to 4 GB

Use the CHANGE.DB command to decrease the maximum size of OSAM PHDAM and PHIDAM data sets for a HALDB database.

When you reduce the data capacity of the OSAM data sets from 8 GB to 4 GB, you must consider how to manage the data in the database, especially if the database contains more than 4 GB of data.

To decrease the maximum size of the OSAM data sets to 4 GB:

1. If the database is already open in an online IMS system, stop access to the database by issuing either the /DBRECOVERY DB *haldb_master* command or the UPDATE DB NAME(*haldb_master*) STOP(ACCESS) command.
2. Unload the data from all of the partitions in the database.
3. Issue the CHANGE.DB command with the NOOSAM8G keyword.
4. Split the partitions that contain any data sets that have more than 4 GB of data. Either define new partitions or rearrange the high key definitions of the existing partitions by using either the HALDB Partition Definition utility (%DFSHALDB) or DBRC commands.
5. Initialize all of the partitions that are affected by the change by running either the HALDB Partition Data Set Initialization utility (DFSUPNT0) or the Database Prereorganization utility (DFSURPR0).
6. Reload all of the partitions that are affected by the change by using the output from unloading the data from the database.
7. Create image copies of the data sets in the affected partitions. DBRC does not allow updates to the data after the image copy required flag is set and before the image copies are recorded in the RECON data sets.

8. Make the partitions available by issuing either the /START DB *haldb_master* command or the UPDATE DB NAME (*partition_name*) START(ACCESS) OPTION(OPEN) command.

The master database record in the RECON data set is updated to indicate that the HALDB supports 4-GB OSAM data sets.

Exit routine modifications and HALDB databases

For most exit routines that you can use with HALDB databases, the methods for modifying them are no different from modifying exit routines for a non-partitioned database.

You make most specifications for HALDB exit routines in the database definition statements and not in the RECON data set. Exceptions to this rule are modifications to a partition selection exit routine or the PHDAM randomizing modules, which you can specify in both the DBD and in the RECON data set. Specifications in the RECON data set override any specifications made in the database definition.

With the exception of PHDAM randomizing routines, any changes you make to database exit routines apply to the whole HALDB database and all of its partitions.

After you make changes to some exit routines, such as PHDAM Randomizing routines or partition selection exit routine, consider reorganizing your database.

The exit routines that are commonly used with HALDB databases include:
- HALDB Partition Selection exit routine
- Data Capture exit routine
- Data Conversion User exit routine
- Segment Edit/Compression exit routines
- Secondary Index Database Maintenance exit routine
- HDAM and PHDAM Randomizing routines

Related concepts:

"Data Capture exit routines" on page 393

"Segment Edit/Compression exit routine" on page 390

Related reference:

↪ Database Manager exit routines (Exit Routines)

Adding or changing a HALDB partition selection exit routine

If your installation uses a HALDB partition selection exit routine, you might need to modify the exit routine to adjust the distribution of records across the partitions in your HALDB database.

Typically, you do this when the amount of data in each partition is not balanced across all of the partitions in the HALDB database. You can change the distribution of records by switching exit routines or modifying the existing exit routine while the entire HALDB database is offline.

To add or change a HALDB partition selection exit routine:
1. Take the entire HALDB database offline by issuing either the type-1 command /DBRECOVERY DB or the type-2 command UPDATE DB NAME(*dbname*) STOP(ACCESS).

2. Unload all of the partitions that will be affected by the new or changed partition selection exit routine.
3. Change the partition definitions, such as the partition strings or randomizing parameters, as necessary. Changing the definition of a partition sets the PINIT flag for the partition, which might save you time in the next step.
4. Set or unset the PINIT flags of each partition as necessary by using either the HALDB Partition Definition utility (%DFSHALDB) or the batch DBRC command `CHANGE.DB`:
    - If you are adding a new partition selection exit routine and are, therefore, introducing a new distribution of records across all partitions, you do not need to change any of the PINIT flags. The PINIT flag is correctly set in all of the partitions.
    - If you are changing only the name of an existing partition selection exit routine and the distribution of records across all partitions remains the same, turn off the PINIT flags in all partitions. Even though only the name of the exit routine has changed, IMS views the exit routine as a new partition selection exit routine and, therefore, sets the PINIT flag of all partitions.
    - If you are changing the name of an existing partition selection exit routine and you have also changed how the partition selection exit routine distributes records across partitions, turn off the PINIT flags in any partitions that are not affected by the change in the distribution of records.
    - If you are changing only the way an existing partition selection exit routine distributes records across partitions, but you have not changed the name of the partition selection exit routine, you must set the PINIT flag in all partitions that are affected by the change in the distribution of records.
5. Initialize the partitions that are affected by any changes in the distribution of records.
6. Reload the partitions affected by any changes in the distribution of records using HD Reorganization Reload utility (DFSURGL0).
7. Take an image copy of the reloaded partitions. The image copy needed flag is set for the data sets in the partitions that are initialized or loaded by the DFSURGL0 utility.
8. Restore access to the HALDB database by issuing either the type-1 command `/START DB` or the type-2 command `UPDATE DB NAME(dbname) START(ACCESS)`.

For more information:
- About the sample HALDB Partition Selection exit routine (DFSPSE00), see *IMS Version 15 Exit Routines*.
- About the HD Reorganization Reload utility, see *IMS Version 15 Database Utilities*.

Related concepts:

"Record distribution and the partition selection exit routine" on page 814

Changing the randomizing module or the randomization parameters of a PHDAM partition

You can store the specifications for PHDAM randomizing modules in two places: the DBD for the HALDB database and the record for each partition in the RECON data set.

The specifications in the DBD apply to all of the partitions in the HALDB database. The specifications in a partition record in the RECON data set apply only to that partition and override the specifications in the DBD.

To change a randomizing module or randomization parameters of a PHDAM partition:

1. Back up the current randomization specifications by issuing the DBRC command `LIST.DB DBD(`*`partition_name`*`)` against the partition and saving the output.
2. Stop access to the partition by issuing either the type-1 command `/DBRECOVERY` or the type-2 command `UPDATE DB NAME(`*`partition_name`*`) STOP(ACCESS)` against the partition. All other partitions in the database can remain available.
3. Unload the partition.
4. If you are modifying the randomization parameters, modify them by using either the Partition Definition utility or the DBRC command `CHANGE.PART`.
5. If you are replacing the randomizing module, specify the name of the new randomizing module by using either the Partition Definition utility or the DBRC command `CHANGE.PART DBD(`*`name`*`) PART(`*`name`*`) RANDOMZR(`*`name`*`)`.
6. Initialize the partition by using either the HALDB Partition Data Set Initialization utility (DFSUPNT0) or the Database Prereorganization utility (DFSURPR0). The partition initialization needed (PINIT) flag is set for the partition after you change the parameters.
7. Reload the partition.
8. Restart the partition by issuing either the type-1 command `/START DB` or the type-2 command `UPDATE DB NAME(`*`partition_name`*`) START(ACCESS)`.

Adding a secondary index to a HALDB database

You can add a secondary index to an existing HALDB database.

IMS does not provide a utility to create secondary indexes. The easiest way to add a secondary index is by using a tool, such as the IBM IMS Index Builder. The IMS Index Builder reads an existing HALDB database and creates one or more secondary indexes for it.

When you add a secondary index, you do not add any entries to the ILDSs because the pointers in the newly created secondary index are accurate. Later, when the partitions are reorganized, IMS adds entries for target segments to the ILDSs.

Adding a secondary index requires new definitions in the indexed database DBD, but does not require changes in the database data sets.

To add a secondary index to a HALDB manually:

1. Create an unload file for the indexed database. You can use HD Reorganization Unload utility (DFSURGU0) or an application program that you write.
2. Add the secondary index definitions to the indexed database DBD.
3. Create the DBD for the secondary index.
4. Define the partitions for the secondary index.
5. Allocate the data sets for the secondary index.
6. Initialize the secondary index partitions.
7. Load the indexed database. You must provide the program to do this. The program reads the file that is created in step 1. When the indexed database is loaded, secondary index entries are created. The creation of the secondary index entries is a random process that can significantly increase the load time.

If you use the DFSURGU0 utility in step 1, the output file contains a header record, one record for each segment, and a trailer record. The segment record includes the segment name and the segment data. Your load program for step 7 might map the records in this file by using the DSECT in the IMS DFSURGUP macro in SDFSMAC.

Related concepts:

"Modifying a HALDB partitioned secondary index"

Modifying a HALDB partitioned secondary index

Changes to HALDB partitioned secondary indexes (PSINDEXes) are like changes to other HALDB databases: you can add, delete, or modify the partitions of a PSINDEX.

Affected partitions must be unloaded, initialized, and reloaded. The indexed database is unaffected by changes to secondary index partitions.

As with non-partitioned full-function databases, if changes to the secondary index require changes to the definition of the indexed database, you might have to unload and reload the indexed database.

An example of a change to a secondary index is a change in a subsequence field. If you add or modify a subsequence field, you must also change the DBD of the indexed database. If the field in the indexed database already exists or does not require other changes to the DBD, you do not have to unload and reload the indexed database. Of course, you need to recreate the secondary index using the new definitions.

Reorganizations of an indexed database do not cause its secondary indexes to be recreated: you must use other means for this, such as the IBM IMS Index Builder tool. Alternatively, you can use a technique similar to one used to add a secondary index.

Initialize partitions in a PSINDEX as you would partitions in a PHDAM or PHIDAM database by using the HALDB Partition Data Set Initialization utility (DFSUPNT0). DFSUPNT0 automatically generates recovery points for the PSINDEX. Recovery points are not created if you delete and redefine your PSINDEX partitions and then turn off their PINIT flags.

When you make changes to an indexed database that do not require changes to secondary index definitions, you do not need to make any changes to the secondary indexes. They do not need to be unloaded, reloaded, or rebuilt. The self-healing pointer scheme of HALDB provides this capability. The reload process for the indexed database updates the ILDSs. This is all that is required to ensure that the secondary index pointers can be used to find the moved target segments.

Related tasks:

"Adding a secondary index to a HALDB database" on page 838

Chapter 32. Converting database types

If the characteristics of your applications have changed over a period of time, performance might be improved by changing to another DL/I access method.

DL/I access methods (or types of databases) are typically chosen based on such variables as:

- The type of processing you needed to do (sequential, direct, or both)
- The volatility of your data

Assuming that you have decided to change access methods, this topic tells you:

- Given your existing DL/I access method, what things you need to change to convert to a different DL/I access method
- How to do the conversion

The reorganization utilities described in related topics can be used to change DL/I access methods among the HISAM, HDAM, and HIDAM access methods. One exception to this is that HDAM cannot be changed to HISAM or HIDAM unless HDAM database physical records are in root key sequence. This exception exists because HISAM and HIDAM databases must be loaded with database records in root key sequence. When the HD Reorganization Unload utility unloads an HDAM database, it unloads it using GN calls. GN calls against an HDAM database unload the database records in the physical sequence in which they were stored by the randomizing module. This will not be root key sequence unless you used a sequential randomizing module (one that put the database records into the database in physical root key sequence).

Related concepts:

Chapter 31, "Modifying databases," on page 721

"Types of pointers you can specify" on page 148

"Determining the size of CIs and blocks" on page 439

"Number of buffers" on page 442

"Choosing HDAM or PHDAM options" on page 434

"Determining which randomizing module to use (HDAM and PHDAM only)" on page 433

Related tasks:

"Specifying free space (HDAM, PHDAM, HIDAM, and PHIDAM only)" on page 431

"Choosing a logical record length for HD databases" on page 438

"Estimating the minimum size of the database" on page 559

Converting a database from HISAM to HIDAM

Converting a database from HISAM to HIDAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need the following before changing your DL/I access method from HISAM to HIDAM:

- Determine whether you are going to set aside free space in the HIDAM database. (Free space is space into which database records are not loaded when the database is initially loaded.)

  Unlike HISAM, in a HIDAM database you can set aside periodic blocks or CIs of free space or a percentage of free space in each block or CI (in the ESDS or OSAM data set). This free space can then be used for inserting database records or segments into the database after initial load.

- Determine what type of pointers you are going to use in the database. Unlike HISAM, HIDAM uses direct-address pointers to point from one segment in the database to the next.

- Reassess your choice of logical record size. A logical record in HISAM can only contain segments from the same database record. In HIDAM, a logical record can contain segments from more than one database record.

- Reassess your choice of CI or block size. In HISAM, your choice of CI or block size should have been some multiple of the average size of a database record. In HIDAM, the size should be chosen because of the characteristics of the device and the type of processing you plan to do.

- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.

- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space.

Once you have determined what changes you need to make, you are ready to change your DL/I access method from HISAM to HIDAM. To do this:

1. Unload your database using the existing DBD and the HD Reorganization Unload utility.
2. Code a new DBD that reflects the changes you need to make. You must also code a DBD for the HIDAM index.
3. If you need to make change that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload the database using the new DBD and the HD Reorganization Reload utility. Remember to make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you need to run additional utilities immediately before and after reloading your database.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Converting a database from HISAM to HDAM

Converting a database from HISAM to HDAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need to do the following before changing your DL/I access method from HISAM to HDAM:

- Determine what type of pointers you are going to use in the database. Unlike HISAM, HDAM uses direct-address pointers to point from one segment in the database to the next.
- Determine which randomizing module you are going to use. Unlike HISAM, HDAM uses a randomizing module. The randomizing module generates information that determines where a database record will be stored.
- Determine which HDAM options you are going to use. Unlike HISAM, an HDAM database is divided into two parts: a root addressable area and an overflow area. The root addressable area contains all root segments and is the primary storage area for dependent segments in a database record. The overflow area is for storage of dependent segments that do not fit in the root addressable area. The HDAM options here are the ones that pertain to choices you make about the root addressable area. These are:
  - The maximum number of bytes of a database record to be put in the root addressable area when segments in the database record are inserted consecutively (without intervening processing operations).
  - The number of blocks or CIs in the root addressable area.
  - The number of RAPs (root anchor points) in a block or CI in the root addressable area. (A RAP is a field that points to a root segment.)
- Reassess your choice of logical record sizes. A logical record in HISAM can only contain segments from the same database record. In HDAM, a logical record can contain segments from more than one database record. In addition, HDAM logical records contain RAPs and two space management fields (FSEs and FSEAPs).
- Reassess your choice of CI or block size. In HISAM, your choice of CI or block size should have been some multiple of the average size of a database record. In HDAM, the size should be chosen because of the characteristics of the device and the type of processing you plan to do.
- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space.

Once you have determined what changes you need to make, you are ready to change your DL/I access method from HISAM to HDAM. To do this:

1. Unload your database, using the existing DBD and the HD Reorganization Unload utility.
2. Code a new DBD that reflects the changes you need to make.
3. If you need to make changes that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes. HDAM only requires one data set, whereas HISAM requires two.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload the database using the new DBD and the HD Reorganization Reload utility. Make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you need to run additional utilities before reloading your database.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Converting a database from HIDAM to HISAM

Converting a database from HIDAM to HISAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need to do the following before changing your DL/I access method from HIDAM to HISAM:

- Reassess your choice of logical record size. A logical record in HISAM can only contain segments from the same database record. In HIDAM, a logical record can contain segments from more than one database record.
- Reassess your choice of CI or block size. In HIDAM, your choice of CI or block size should be based on the characteristics of the device and the type of processing you plan to do. In HISAM, the size should be some multiple of the average size of a database record.
- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space.

Once you have determined what changes you need to make, you are ready to change your DL/I access method from HIDAM to HISAM. To do this:

1. Unload your database using the existing DBD and the HD Reorganization Unload utility.
2. Code a new DBD that reflects the changes you need to make. You will not be specifying direct-address pointers or free space in the DBD, because HISAM, unlike HIDAM, does not allow use of these. Also, HISAM has only one DBD whereas HIDAM had two.
3. If you need to make changes that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload the database using the new DBD and the HD Reorganization Reload utility. Remember to make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you must run additional utilities right before and after reloading your database.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Converting a database from HIDAM to HDAM

Converting a database from HIDAM to HDAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need to do the following before changing your DL/I access method from HIDAM to HDAM:

- Reassess your choice of direct-address pointers. Although both HIDAM and HDAM use direct-address pointers, you might need to change the type of direct-address pointer used:

- Because of the changing needs of your applications.
  - Because pointers are partly chosen based on the type of database you are using. For example, if you used physical twin backward pointers on root segments in your HIDAM database to get fast sequential processing of roots, they will not have any use in an HDAM database.
- Determine which randomizing module you are going to use. Unlike HIDAM, HDAM uses a randomizing module. The randomizing module generates information that determines where a database record is to be stored.
- Determine which HDAM options you are going to use. Unlike HIDAM, an HDAM database does not have a separate index database. Instead the database is divided into two parts: a root addressable area and an overflow area. The root addressable area contains all root segments and is the primary storage area for dependent segments in a database record. The overflow area is for storage of dependent segments that do not fit in the root addressable area. The HDAM options here are the ones that pertain to choices you make about the root addressable area. These are:
  - The maximum number of bytes of a database record to be put in the root addressable area when segments in the database record are inserted consecutively (without intervening processing operations).
  - The number of blocks or CIs in the root addressable area.
  - The number of RAPs in a block or CI in the root addressable area.
- Reassess your choice of logical record size.
- Reassess your choice of CI or block size.
- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space.

After you have determined what changes you need to make, you are ready to change your DL/I access method from HIDAM to HDAM. To do this:

1. Unload your database using the existing DBD and the HD Reorganization Unload utility.
2. Code a new DBD that reflects the changes you need to make. You probably will not be specifying free space, but you will be specifying HDAM options. Note also that you'll need only one DBD for HDAM, whereas HIDAM required two DBDs.
3. If you need to make changes that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes. HDAM only requires one data set, whereas HIDAM requires two.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload the database using the new DBD and the HD Reorganization Reload utility. Remember to make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you will need to run additional utilities right before and after reloading your database.

Related concepts:

"Types of pointers you can specify" on page 148

Converting a database from HDAM to HISAM

Converting a database from HDAM to HISAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need to do the following before changing your DL/I access method from HDAM to HISAM:

- Reassess your choice of logical record size. A logical record in HISAM can only contain segments from the same database record. In HISAM, a logical record can contain segments from more than one database record.
- Reassess your choice of CI or block size. In HDAM, your choice of CI or block size should be based on the characteristics of the device and the type of processing you plan to do. In HISAM, the size should be some multiple of the average size of a database record.
- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.
- Recalculate database space. You need to recalculate database space because the changes you are making will result in different requirements for database space.

After you have determined what changes you need to make, you are ready to change your DL/I access method from HDAM to HISAM. Remember you must write your own unload and reload programs unless database records in the HDAM database are in physical root key sequence. In writing your own load program, if your HDAM database uses logical relationships, you must preserve information in the delete byte (for example, a segment that is logically deleted in the database might not be physically deleted).

To change from HDAM to HISAM:

1. Unload your database using the existing DBD and one of the following:
   - Your unload program
   - The HD Reorganization Unload utility if database records are in physical root key sequence
2. Code a new DBD that reflects the changes you need to make. You will not be specifying direct-address pointers or HDAM options.
3. If you need to make changes that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes. HDAM only requires one data set, whereas HISAM requires two.
4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Reload the database using the new DBD and:
   - Your load program, or
   - The HD Reorganization Reload utility if database records are in physical root key sequences

   Remember to make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you need to run additional utilities right before and after reloading your database.

Related tasks:
"Offline reorganization by using the reorganization utilities" on page 646

Converting a database from HDAM to HIDAM

Converting a database from HDAM to HIDAM can be performed in a few steps; however, you need to perform a number of preliminary steps also.

You need to make the following changes before changing your DL/I access method from HDAM to HIDAM:

- Determine whether you are going to set aside free space in the HIDAM database. (Free space is space into which database records are *not* loaded when the database is initially loaded.) In a HIDAM database, you can set aside periodic blocks or CIs of free space or a percentage of free space in each block or CI (in the ESDS or OSAM data set). This free space can then be used for inserting database records or segments into the database after initial load. In an HDAM database, you generally get the free space you need by careful choice of HDAM options.
- Reassess your choice of direct-address pointers. Although both HIDAM and HDAM use direct-address pointers, you might need to change the type of direct-address pointer used:
  - Because of the changing needs of your applications.
  - Because pointers are partly chosen based on the type of database you are using. For example, you can chose to use physical twin forward and backward pointers on root segments in your HIDAM database to get fast sequential processing of roots.
- Reassess your choice of logical record size.
- Reassess your choice of CI or block size.
- Reassess your choice of database buffer sizes and the number of buffers you have allocated. If you have changed your CI or block size, you need to allocate buffers for the new size.
- Recalculate database space. You need to recalculate database space because the changes you are making will result in different requirements for database space.

Once you have determined what changes you need to make, you are ready to change your DL/I access method from HDAM to HIDAM. Remember you must write your own unload and reload programs unless database records in the HDAM database are in physical root key sequence. In writing your own load program, if your HDAM database uses logical relationships, you must preserve information in the delete byte (for example, a segment that is logically deleted in the database might not be physically deleted).

To change from HDAM to HIDAM:

1. Unload your database using the existing DBD and one of the following:
   - Your unload program
   - The HD Reorganization Unload utility if database records are in physical root key sequence
2. Code a new DBD that reflects the changes you need to make. You must also code a DBD for the HIDAM index. You will not be specifying HDAM options but you probably will be specifying free space.

3. If you need to make changes that are not specified in the DBD (such as changing database buffer sizes or the amount of space allocated for the database), make these changes. HDAM only requires one data set, whereas HIDAM requires two.

4. For non-VSAM data sets, delete the old database space and define new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.

5. Reload the database using the new DBD and one of the following:
   - Your load program
   - The HD Reorganization Reload utility if database records are in physical root key sequence.

   Remember to make an image copy of your database as soon as it is reloaded.

   If you are using logical relationships or secondary indexes, you need to run additional utilities before reloading your database.

Related tasks:

"Offline reorganization by using the reorganization utilities" on page 646

Converting HDAM and HIDAM databases to HALDB

You can convert an HDAM or HIDAM database to a HALDB database by using the IMS base utilities.

For a logical view of HDAM and HIDAM databases before and after changing to PHDAM and PHIDAM, respectively, see the following figure.
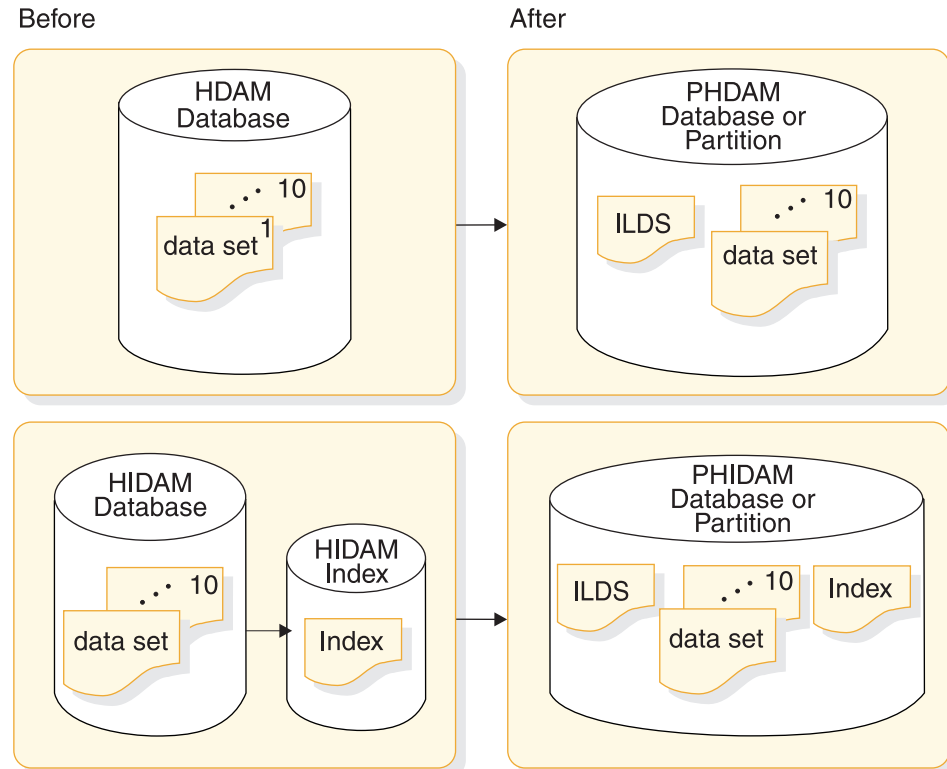
Figure 298. HDAM and HIDAM databases before and after changing to PHDAM and PHIDAM

Restriction: HALDB support for logical relationships does not include support for virtual pairing. When converting a logically related database to HALDB, you can convert the logical relationships as follows:

- Unidirectional relationships in the database being converted can remain unidirectional in the HALDB database.
- Physically-paired logical relationships in the database being converted can remain physically-paired in the HALDB database.
- Virtually-paired logical relationships in the database being converted can be converted to physically-paired in the HALDB database.

In addition to the basic steps for converting a database to HALDB, this section also contains the steps for converting a database with a secondary index to HALDB, and the steps for converting databases with logical relationships to HALDB. Each of these tasks, as well other concepts and issues related to the conversion process, are covered in the following topics.

Related tasks:

"Modifying HALDB databases" on page 809

Parallel unload for migration to HALDB

If you are migrating a large database to HALDB, you can improve unload performance by running the HD Unload Utility (DFSURGU0) in multiple parallel jobs, with each job unloading a specified key range of database records.

Unloading a key range of database records is supported only when the MIGRATE=YES option is used.

To further improve migration performance, unload a key range for a single partition in one step followed by a migration reload of the corresponding HALDB partition.

The migration reload must use DBRC=Y and requires that the RECON data sets that contain the new HALDB partition definitions be allocated either with JCL or, for dynamic allocation, with DBRC RECON MDA members.

Related reference:

▶ HD Reorganization Unload utility (DFSURGU0) (Database Utilities)

Backing up existing database information

Before beginning the conversion process, back up the critical database elements.

Because you will be modifying the DBD statements for your existing database, deleting the database information from the RECON data sets, and unloading and reloading the database itself, you should back up each of these elements in case you need to restore the non-HALDB database structure.

To begin the conversion process, complete the following steps to back up the critical database elements:

- Create image copies of your database data sets by using one of the IMS image copy utilities or other means.
- Before making any changes to the DBD, back up the DBD source for the existing database.
- Before deleting information from the RECON data sets, back up the RECON data sets. There are two methods to backing up the RECON data sets:

- To save only the information specific to the database you are converting to HALDB, issue the DBRC LIST.DB command. If you need to revert your database to its non-HALDB state after the conversion, you will only have to re-enter into the RECON data sets the non-HALDB database information; however, you will likely have to manually re-enter the information into the RECON data sets.
- To back up the entire RECON data set, issue the DBRC BACKUP.RECON command. If you need to revert your database to its non-HALDB state after the conversion, you will have to restore the backup copy of the RECON data set and then recover all changes to the RECON data sets that have occurred since the backup copy was created.

Regardless of which method of recovery you intend to use, consider issuing both the LIST.DB command and the BACKUP.RECON command to ensure that all options are available to you in the event you need to revert your database to its non-HALDB state.

Related tasks:

"Converting logically related HDAM or HIDAM databases to HALDB" on page 869

"Converting HDAM or HIDAM databases with secondary indexes to HALDB" on page 856

"Converting simple HDAM or HIDAM databases to HALDB PHDAM or PHIDAM"

Converting simple HDAM or HIDAM databases to HALDB PHDAM or PHIDAM

Databases that do not have secondary indexes or logical relationships are referred to as *simple databases.* If your database uses secondary indexes or logical relationships, you must take extra steps to convert them.

Prior to converting your database to HALDB, backup the database, supporting data sets, and the associated DBD statements.

To convert a simple HDAM or HIDAM database to a HALDB database, complete the steps that are detailed in the topics listed below.

Related tasks:

"Backing up existing database information" on page 849

"Converting logically related HDAM or HIDAM databases to HALDB" on page 869

"Converting HDAM or HIDAM databases with secondary indexes to HALDB" on page 856

Unloading the existing database

Unload the existing database by using the IMS HD Reorganization Unload utility (DFSURGU0).

Specify the MIGRATE=YES control statement. The MIGRATE=YES control statement identifies this unload as being part of the HALDB conversion process. The IMS HD Reorganization Unload utility must use the DBD that defines the non-HALDB database.

Recommendations:

- If you currently use OSAM database data sets, use OSAM sequential buffering when unloading the database. OSAM sequential buffering typically improves the performance of the unload.
- If the database that you are migrating is large and you are using the MIGRATE control statement, the DFSURGU0 utility can unload records in ranges of keys. Multiple key ranges can be unloaded in parallel jobs to improve unload performance.

Related concepts:

"OSAM sequential buffering" on page 445

Related reference:

➡ HD Reorganization Unload utility (DFSURGU0) (Database Utilities)

Deleting database information from the RECON data sets

Delete the database information from the RECON data sets by using the DBRC DELETE.DB command.

You must delete the database information prior to registering the new HALDB database with DBRC in the next step.

Defining HALDB database DBD statements

Because the DBD for your new HALDB is based on the DBD of your current non-HALDB database, you must make several changes to your existing DBD to convert your database to HALDB.

To modify the DBD, you need to modify the following statements:
- The DBD statement of the database being converted
- The DBD statement of a HIDAM primary index
- The DATASET statement
- The SEGM statement
- The LCHILD statement
- The FIELD statement
- The XDFLD statement

Changing the DBD statement

You might need to make the following changes to the DBD statement for HALDB:
- Specify PHDAM, PHIDAM, or PSINDEX for the ACCESS parameter, depending on the type of HALDB database to which you are converting.
- If you are going to use a partition selection exit routine, specify the module name of the exit routine by using the PSNAME parameter in the DBD statement for the HALDB database. If you are not going to use the exit routine, do not specify the PSNAME parameter.

  You can also introduce or change a partition selection exit routine during the partition definition process. Any specification that you make for a partition selection exit routine during the partition definition process overrides any specification you have made in the DBD statement.
- If you are converting to a PHDAM database, specify the RMNAME parameter. For HALDB, the RMNAME parameter defines a default randomizer value for all partitions in the HALDB database. You can specify different randomizer values for individual partitions when you define each partition.

Tip: When you convert your database to a HALDB database, keep the same database name, if possible. If you change the name, you will need to change all of the PSBs that reference the database. All of the conversion processes that are described in this information use the same database name before and after the conversion process.

Omitting the DBD statements for HIDAM indexes

If your database is a HIDAM database, delete the DBD statements for the primary index. PHIDAM primary indexes do not require a DBD. When a HIDAM database is migrated to PHIDAM, the DBD for the HIDAM index is discarded. IMS obtains the information that is required to generate the PHIDAM primary index from the PHIDAM DBD.

Deleting the DATASET statement

HALDB databases do not use the DATASET statement. If you are creating the DBD for the HALDB database by modifying the DBD of the existing database, delete the DATASET statement. You can specify the elements that were previously defined in the DATASET statement by using the following methods:
- Create ddnames when you define partitions.
- Define data set groups by using the DSGROUP parameter on SEGM statements.
- Specify free space and OSAM block sizes when you define partitions.
- Omit the SCAN parameter for HALDB databases. HALDB searches only the current cylinder for available storage space during segment insertion operations.

Changing the SEGM statement

You might need to make the following changes to the SEGM statement for HALDB:
- Change the pointer specifications of the PTR parameter.

  HALDB does not use hierarchical pointers. You must change any specifications of the HIER, H, HIERBWD, or HB keywords with the PTR parameter to TWIN, T, TWINBWD, or TB.

  PHIDAM does not support the use of twin forward only (TWIN or T) pointers for root segments. If you have HIDAM roots that use twin forward only pointers, change the keyword for the PTR parameter on the SEGM statement to NOTWIN, NT, TWINBWD, or TB.

  HALDB does not use symbolic pointers. The changes that are required for logical relationships are explained in "Defining DBD statements for logically related HALDB databases" on page 871.
- If you want to use multiple data set groups, specify the DSGROUP parameter in the SEGM statement.

  To define multiple data set groups, you must specify the DSGROUP parameter on the SEGM statement for any segment that is not in the first data set group. The valid values for DSGROUP are the letters A through J. A is the first data set group, B is the second, and J is the tenth.
- If you are using a /SX field as a subsequence field, you must increase the value of the BYTES parameter for a secondary index segment by 4 bytes. For more information, see "Defining DBD statements for the PSINDEX" on page 860.

Changing the LCHILD statement

The LCHILD statement does not define the primary index for PHIDAM databases. When converting a HIDAM DBD to PHIDAM, delete the LCHILD statement.

When converting a secondary index DBD to a HALDB PSINDEX DBD, you might need to make the following changes:

- If PTR=SYMB is specified in a secondary index, you must either change it to PTR=SNGL or omit the PTR= keyword. PTR=SNGL is the default and is the only valid specification for PSINDEX LCHILD statements.
- If PTR=SYMB is specified for a HDAM or HIDAM indexed database, you must change it to PTR=INDX.
- If your database has a secondary index, you must specify the size of the key of the root segment in the target database by using the RKSIZE parameter on the LCHILD statement in the secondary index DBD.

When you use symbolic pointing to convert a logical relationship, you must omit the PTR=SYMB specification on the LCHILD statement for the logical relationship.

Changing the FIELD statement

The changes for LCHILD statements that are used to define logical relationships that use virtual pairing are explained in "Defining DBD statements for logically related HALDB databases" on page 871.

In the secondary index DBD, the value of the BYTES parameter of the FIELD statement for the sequence field of a secondary index segment must be increased by 4 bytes if a /SX field is used as a subsequence field. The /SX field is defined in the indexed database DBD. For more information, see "Converting HDAM or HIDAM databases with secondary indexes to HALDB" on page 856.

Changing the XDFLD statement

HALDB does not support shared secondary indexes. If the CONST parameter is specified in an XDFLD statement of an indexed database, you must delete it. The CONST parameter specifies the character that is associated with a shared secondary index. Each HALDB secondary index must be stored in its own secondary index database. Separate PSINDEX databases must be defined for each shared secondary index that is being converted to HALDB.

Related tasks:

"Defining DBD statements for logically related HALDB databases" on page 871

"Defining the DBD statements for a HALDB database indexed by a PSINDEX" on page 860

Registering the HALDB master database with DBRC
A HALDB master database must be registered with DBRC.

To register the HALDB master database with DBRC use either the Partition Definition utility or the DBRC batch command INIT.DB.

Defining the partitions to DBRC
HALDB partitions are defined in the DBRC RECON data set.

When defining partitions, you must have update authority for the RECON data sets.

To define the partitions to DBRC, use either the Partition Definition utility or the DBRC batch command `INIT.PART`.

Allocating database data sets

You need to allocate the database data sets that are used by the databases.

The data set names are created from the data set name prefix that you specify when you define a partition and the data set name algorithm. You must use these names when you allocate your data sets.

Allocate the database data sets for each partition, including the indirect list data set (ILDS).

The information that is generated by the HALDB Migration Aid utility (DFSMAID0) can assist you in determining how large to make the data sets. The utility reports the number of bytes used by the existing segment data in each partition. You will need to add additional bytes to allow space for segments to be added, for the free space you want to include, and for bitmaps.

When migrating to HALDB, you might want to increase your free space parameters. Non-HALDB databases sometimes have suboptimal free space values due to data set size limitations. Additional free space allows you to reorganize your databases or partitions less frequently. In some cases, additional free space can eliminate the need for reorganizations altogether.

Although databases that do not have logical relationships or secondary indexes do not use the ILDS, in some cases HALDB requires that an ILDS exist in each partition anyway. An online IMS system does not allocate an ILDS if the database does not require one; however, batch jobs do allocate an ILDS, so you must define an ILDS. IMS issues message IEC161I 152-061 when an empty ILDS is opened, but this message does not indicate a problem.

When no secondary indexes or logical relationships are defined, the HD Reorganization Reload utility (DFSURGL0) does not update the ILDS.

If you do not use the ILDS, you can allocate a very small amount of space for the data set, such as one track.

All ILDSs have fixed-length 50-byte records. Keys are 9 bytes at zero offset. The following example shows IDCAMS DEFINE statements to allocate an ILDS.

```
DEFINE CLUSTER(                        -
    NAME(JOUKO3.HALDB.DB.PEOPLE.L00001) -
    INDEXED                            -
    CYL(1 1)                           -
    RECORDSIZE(50 50)                  -
    SHAREOPTIONS(3 3)                  -
    SPEED                              -
    KEY(9,0)                           -
    FREESPACE(10,10)                   -
    CONTROLINTERVALSIZE(8192)          -
    VOLUMES(TOTIMN)                    -
)
```

You must specify REUSE on the DEFINE statement for all HALDB VSAM data sets other than ILDSs.

The output of the DBDGEN utility is useful when you allocate PHIDAM primary indexes. The output of the DBDGEN utility for the PHIDAM database lists the

required parameters for the IDCAMS definition, as shown following the label RECOMMENDED VSAM DEFINE CLUSTER PARAMETERS. The required parameters are the INDEXED parameter, the values of the RECORDSIZE parameter, the REUSE parameter, and the values of the KEY parameter. The following example shows the IDCAMS DEFINE statement to allocate a PHIDAM primary index data set.

```
DEFINE CLUSTER(                        -
   NAME(JOUKO3.HALDB.DB.RZL.X00001) -
   INDEXED                            -
   CYL(10 5)                          -
   RECORDSIZE(20 20)                  -
   SHAREOPTIONS(3 3)                  -
   REUSE                              -
   KEY(14,5)                          -
   FREESPACE(25,10)                   -
   CONTROLINTERVALSIZE(8192)          -
   VOLUMES(TOTIMN)                    -
)
```

Related concepts:

"Naming conventions for HALDB partitions, ddnames, and data sets" on page 26

Related tasks:

"Allocating logically related database data sets" on page 874

"Allocating the indexed database data sets" on page 866

Initializing the partitions

Before you can use HALDB partitions, you must initialize them.

To initialize partitions, you can use either the IMS Database Prereorganization utility (DFSURPR0) or the IMS HALDB Database Data Set Initialization utility (DFSUPNT0). Alternatively, you can use the IBM IMS High Performance Load tool to initialize partitions.

Related concepts:

"HALDB partition initialization" on page 185

Loading the database as a HALDB database

Reload the database as a HALDB database by using the HD Reorganization Reload utility (DFSURGL0).

The input to the HD Reorganization Reload utility is the output from the HD Reorganization Unload utility (DFSURGU0). You can ensure that your ILDS includes free space by specifying the ILDSINGLE control statement when running the HD Reorganization Reload utility.

Image copying the database data sets

The reload process sets the *image copy needed* flag for each database data set other than ILDSs and PHIDAM primary indexes.

Create image copies of the flagged data sets.

Cleaning up DFSMDA members and HIDAM primary index DBDs

HALDB databases do not use DFSMDA members for dynamic allocation and do not require a separate DBD for the primary index of a PHIDAM database.

HALDB databases use information stored in the DBRC RECON data set to enable dynamic allocation of the database data sets and do not use DFSMDA members.

After the conversion process is complete and you are certain you will not need to revert the database to its non-HALDB state, you can delete the DFSMDA members.

IMS obtains the information that is required to generate the PHIDAM primary index from the PHIDAM DBD. If you converted a HIDAM database to a HALDB PHIDAM database, after the conversion process is complete and you are certain that you will not need to revert the database to its non-HALDB state, you can discard the DBD for the primary index of the old HIDAM database.

Related tasks:

"Cleaning up DFSMDA members and HIDAM primary index DBDs" on page 876

"Cleaning up DFSMDA members and HIDAM primary index DBDs after converting to PSINDEX" on page 869

Converting HDAM or HIDAM databases with secondary indexes to HALDB

Partitioned secondary indexes (PSINDEXes) are the only type of secondary index that HALDB databases support. When you convert HDAM or HIDAM databases that have secondary indexes to HALDB PHDAM or PHIDAM databases, you must also convert the secondary index databases to HALDB PSINDEX databases.

Create image copies of your database data sets and back up your RECON data sets and your existing DBD definitions.

To convert a target HDAM or HIDAM database and its secondary index to HALDB, the steps are essentially the same as those documented in greater detail in "Converting simple HDAM or HIDAM databases to HALDB PHDAM or PHIDAM" on page 850, except for the changes that are related to converting the secondary index and populating the ILDS.

You must define partition boundaries for the PSINDEX database. You can run the HALDB Migration Aid utility (DFSMAID0) against your existing secondary index to help you determine a partitioning scheme for your PSINDEX.

If you are using only the IMS base utilities for the conversion process, you can take either of the two following approaches when you convert your database and its secondary indexes:

- To create secondary index output files as the HD Reorganization Unload utility (DFSURGU0) unloads the target database, specify the MIGRATX=YES control statement when you run the HD Reorganization Unload utility. The MIGRATX=YES control statement instructs the utility to create secondary index output files from the target segments of the indexed database without reading the secondary indexes. The MIGRATX=YES control statement method does not preserve any user data in the secondary indexes.
- To convert the secondary index as a standalone database, specify the MIGRATE=YES control statement when you run the HD Reorganization Unload utility for both the indexed database and the each secondary index. This method requires that the HD Reorganization Unload utility read and unload each secondary index separately. When reading the secondary index, the utility must also read the indexed database.

  This information does not describe the MIGRATE=YES control statement method for converting secondary indexes to HALDB.

Recommendation: Use the MIGRATX=YES method.

Related tasks:

Unloading the existing database

Unload the existing target database by using the HD Reorganization Unload utility with the MIGRATX=YES control statement. The MIGRATX=YES control statement creates unload files for the secondary indexes in addition to the unload file for the indexed database.

When you unload the database by using the MIGRATX=YES control statement, the HD Reorganization Unload utility does not read the secondary indexes. DD statements for the secondary index data sets are not required. The information that is needed to create secondary index entries is generated from the source segments. The unload files for secondary indexes must be sorted before they are used to load the secondary indexes. The output of the HD Reorganization Unload utility includes sort control statements for sorting the unload files of the secondary indexes.

The MIGRATX=YES control statement technique does not preserve any user data in the secondary indexes, which is rarely a problem. Most installations do not maintain user data in secondary indexes because user data is lost every time a reorganization of a non-HALDB database requires the rebuilding of its secondary indexes.

The following figure illustrates a use of the HD Reorganization Unload utility with the MIGRATX=YES control statement to migrate a database and its two secondary indexes.
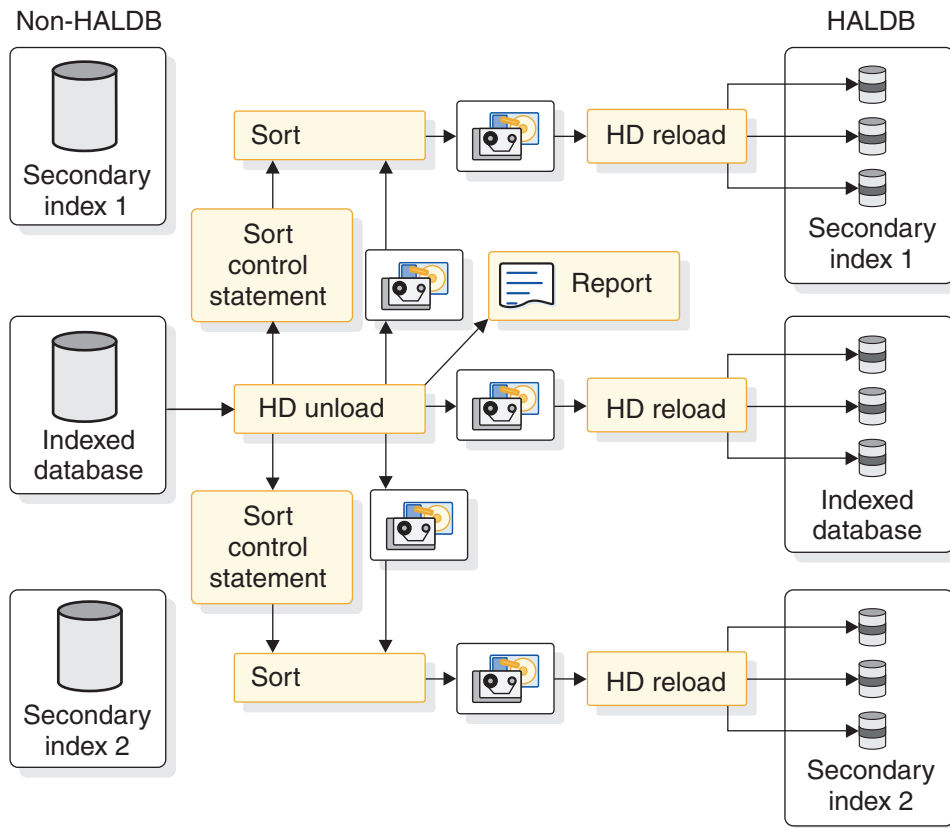
Figure 299. Using the MIGRATX=YES control statement to migrate a database with two secondary indexes

If you use OSAM database data sets, you can improve the performance of the unload process by using OSAM sequential buffering.

The SYSPRINT listing of the HD Reorganization Unload utility includes a Work File Statistics report. The HD Reorganization Unload utility assigns a name, such as DFSWORK1 or DFSWORK2, to the output file for each secondary index. The Work File Statistics report lists the output file name, the sort file name, and other data associated with each secondary index on a single row in the report. This report maps each output file to each secondary index. You must use the report to understand this relationship. The utility assigns the name DFSWRK01 to the first secondary index relationship it encounters in during the unload process. The report also lists the number of segments in the secondary index files and the sort field sizes and offsets. The following example shows a sample report in which the work file DFSWRK01 contains the records for secondary index STUNUM, and the work file DFSWRK02 contains the records for secondary index STUCRT.

```
W O R K   F I L E   S T A T I S T I C S

SINAME     WFNAME     SFNAME     RCDTOTAL       OFFSET    LENGTH

STUNUM     DFSWRK01   DFSSRT01   000087012      0124      0006

STUCRT     DFSWRK02   DFSSRT02   000010702      0124      0010
```

Sorting the output of the HD Reorganization Unload utility

The method for sorting the output of the HD Reorganization Unload utility differs depending on whether you are using /SX fields to create unique keys in your secondary index.

Sorting output when you are not using a /SX field for uniqueness

If you are not using /SX fields, sort the secondary index output data sets by using the sort control statements that are generated by the HD Reorganization Unload utility in the data sets that are defined by the DFSSRTnn DD statement.

You must sort each secondary index output file. The following example shows sample JCL for the sorting of the STUNUM secondary index. The SORTIN DD specifies the DFSWRK01 output from the HD Reorganization Unload utility. The SYSIN DD specifies the DFSSRT01 output from the HD Reorganization Unload utility. The output of this sort is input to the HD Reorganization Reload utility (DFSURGL0) for the STUNUM secondary index.

```
//*********************************************************************
//*   SORT THE STUNUM SEC. INDEX RECORDS FROM MIGRATX=YES
//*********************************************************************
//*
//SORT01   EXEC PGM=SORT,REGION=2048K,PARM='CORE=MAX'
//SORTLIB  DD DSN=SYS1.SORTLIB,DISP=SHR
//SYSOUT   DD SYSOUT=*
//SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,(50,5),,CONTIG)
//SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,(50,5),,CONTIG)
//SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,(50,5),,CONTIG)
//SYSIN    DD DSN=JOUKO3.STUDENT.MIGR.SRT01,DISP=OLD
//SORTIN   DD DSN=JOUKO3.STUDENT.MIGR.WRK01,DISP=OLD
//SORTOUT  DD DSN=JOUKO3.STUDENT.MIGR.WRK01.SORTED,DISP=(NEW,CATLG),
//           UNIT=3390,VOL=SER=TOTIMN,
//           SPACE=(CYL,(50,10),RLSE)
```

Sorting output when you are using a /SX field for uniqueness

If you are using /SX fields for uniqueness, perform the following steps to place the secondary index unload records in the correct subsequence order:

1. Split the file into three separate files that contain the header record, the unload records, and the trailer record. The following example shows sort JCL and control statements for this step.

   Note: The following example is used when MIGRATX=YES is specified. When MIGRATE=YES is specified, the OUTFIL INCLUDE statement for the trailer record includes X'0090' instead of X'0290'.

   ```
   //SORTIN    DD  DSN=UNLOAD.OUTPUT,DISP=SHR
   //HEADER    DD  DSN=HEADER.FILE,DISP=(NEW,PASS)
   //TRAILER   DD  DSN=TRAILER.FILE,DISP=(NEW,PASS)
   //ULCOPY    DD  DSN=UNLOAD.COPY,DISP=(NEW,PASS)
   //SYSIN     DD  *
   OPTION     COPY
   OUTFIL INCLUDE=(5,2,CH,EQ,X'0080'),FNAMES=HEADER
   OUTFIL INCLUDE=(5,2,CH,EQ,X'0290'),FNAMES=TRAILER
   OUTFIL SAVE,FNAMES=ULCOPY
   RECORD TYPE=V
   ```

2. Sort the file that contains the unload records in the /SX subsequence order. You must calculate the offset and the size of the sort field. The offset is 63 bytes plus the size of the root key of the indexed database. The sort field size is the size of the secondary index search field plus 8 bytes. The following example shows sort JCL and control statements for this step. In this example, the offset is 73 bytes and the sort field size is 18 bytes.

   ```
   //SORTIN    DD DSN=UNLOAD.COPY,DISP=SHR
   //SORTOUT   DD DSN=UNLOAD.SORTED1,DISP=(,CATLG),
   //             UNIT=SYSDA,VOL=SER=000000,SPACE=(CYL,(1,5))
   ```

```
//SYSIN    DD  *
SORT FIELDS=(73,18,CH,A),FILSZ=E1000
RECORD TYPE=V
END
```

3. Merge the records into one file. The following example shows sort JCL and control statements for this step.

```
//SORTIN DD DSN=UNLOAD.HEADER,DISP=(OLD,DELETE),
//          UNIT=SYSDA,VOL=SER=000000,SPACE=(CYL,(1,5))
//       DD DSN=UNLOAD.SORTED1,DISP=(OLD,DELETE),
//          UNIT=SYSDA,VOL=SER=000000,SPACE=(CYL,(1,5))
//       DD DSN=UNLOAD.TRAILER,DISP=(OLD,DELETE),
//          UNIT=SYSDA,VOL=SER=000000,SPACE=(CYL,(1,5))
//SORTOUT DD DSN=UNLOAD.SORTED2,DISP=(NEW,KEEP),
//          UNIT=SYSDA,VOL=SER=000000,SPACE=(CYL,(1,5))
//SYSOUT   DD  SYSOUT=A
//SYSIN    DD  *
OPTION    COPY
END
```

Deleting database information from the RECON data sets

You must delete the old database information from the RECON data set prior to registering the new HALDB database with DBRC in the next step.

Delete the database information from the RECON data sets by using the DBRC DELETE.DB command.

Defining the DBD statements for a HALDB database indexed by a PSINDEX

Attributes of the indexed HALDB master database are defined by using the DBD statements.

To define the DBD statements for the indexed HALDB database, modify the following DBD statements:

- The DBD statement
- Remove the DATASET statement
- The SEGM statement
- The LCHILD statement
- The FIELD statement
- If your database uses shared secondary indexes, delete the CONST parameter from the XDFLD statement of the indexed database before you convert the secondary index to HALDB.

Related tasks:

"Defining HALDB database DBD statements" on page 851

Defining DBD statements for the PSINDEX

Attributes of the PSINDEX master database are defined by using the DBD statements.

Define the following DBD statements for the PSINDEX:

- DBD statement

  Specify ACCESS=PSINDEX to identify this database as a HALDB partitioned secondary index.

- SEGM statement
- LCHILD statement

  You must specify the RKSIZE parameter for a PSINDEX.

If you used symbolic pointers in your non-HALDB secondary index, you must change the PTR=SYMB specification in the indexed database DBD to PTR=INDX and either omit the PTR parameter in the secondary index DBD or specify PTR=SNGL in it.

- XDFLD statement
- FIELD statement

  If you use non-unique keys in your existing secondary index, you can use the /SX field to create unique keys in your PSINDEX. HALDB PSINDEXes require unique keys.

You must change the DBDs for your secondary index databases before they are loaded as HALDB secondary indexes. You might also need to change the DBDs of some of your indexed databases.

Removing symbolic pointers

If you use symbolic pointers, you must remove them, because HALDB secondary indexes do not support symbolic pointers.

Symbolic pointers are defined with a PTR=SYMB parameter on the LCHILD statements in the non-HALDB secondary index and the indexed database DBDs. You must change the PTR=SYMB specification in the indexed database DBD to PTR=INDX and either omit the PTR parameter in the secondary index DBD or specify PTR=SNGL in it.

The examples in the following series of figures illustrate the changes you need to make to remove symbolic pointers from the DBD statements. The DBD statements involved are:

- The non-HALDB secondary index DBD with symbolic pointing defined.
- The indexed HDAM DBD.
- The secondary index DBD after it has been converted to HALDB.
- The indexed database DBD after it has been converted to HALDB.

The following figure shows the non-HALDB secondary index DBD with symbolic pointing defined.

Figure 300. Example: DBD for secondary index that uses symbolic pointing

```
DBD    NAME=CONTRSI,ACCESS=INDEX
DATASET DD1=CONTSI,DEVICE=3390,SIZE=8192
SEGM  NAME=CONTR,BYTES=26,PARENT=0
FIELD NAME=(CONTRNUM,SEQ,U),BYTES=8,START=1,TYPE=C
LCHILD NAME=(CONTRACT,ENGAGEM),INDEX=CONTRIDX,PTR=SYMB
DBDGEN
FINISH
END
```

The following figure shows the indexed HDAM DBD.

Figure 301. Example: DBD for indexed database with symbolic pointing

```
DBD    NAME=ENGAGEM,ACCESS=HDAM,RMNAME=(DFSHDC40,1,500,824)
DATASET DD1=ENGAHDAM,BLOCK=1648,SCAN=0
SEGM  NAME=CLIENT,BYTES=100,PTR=TWIN
FIELD NAME=(CLNUM,SEQ,U),BYTES=10,START=1,TYPE=C
```

```
SEGM  NAME=CONTRACT,PARENT=CLIENT,BYTES=60,PTR=TWIN
FIELD NAME=(CONTRNO,SEQ,U),BYTES=8,START=1,TYPE=C
LCHILD NAME=(CONTR,CONTRSI),PTR=SYMB
XDFLD NAME=CONTRIDX,SRCH=CONTRNO
DBDGEN
FINISH
END
```

The following figure shows the secondary index DBD after it has been converted to HALDB. PTR=SYMB is deleted from the LCHILD statement. RKSIZE is added to it. Symbolic pointers are stored in the data area of index segments. Because it is not in the HALDB secondary index, the BYTES parameter on the SEGM segment is reduced by the size of the symbolic pointer.

Figure 302. Example: DBD for HALDB secondary index

```
DBD   NAME=CONTRSI,ACCESS=PSINDEX
SEGM  NAME=CONTR,BYTES=8,PARENT=0
FIELD NAME=(CONTRNUM,SEQ,U),BYTES=8,START=1,TYPE=C
LCHILD NAME=(CONTRACT,ENGAGEM),INDEX=CONTRIDX,RKSIZE=10
DBDGEN
FINISH
END
```

The following figure shows the indexed database DBD after it has been converted to HALDB. PTR=INDX is specified on the LCHILD segment for the secondary index relationship.

Figure 303. Example: DBD for HALDB indexed database

```
DBD   NAME=ENGAGEM,ACCESS=PHDAM,RMNAME=(DFSHDC40,1,500,824)
SEGM  NAME=CLIENT,BYTES=100,PTR=TWIN
FIELD NAME=(CLNUM,SEQ,U),BYTES=10,START=1,TYPE=C
SEGM  NAME=CONTRACT,PARENT=CLIENT,BYTES=60,PTR=TWIN
FIELD NAME=(CONTRNO,SEQ,U),BYTES=8,START=1,TYPE=C
LCHILD NAME=(CONTR,CONTRSI),PTR=INDX
XDFLD NAME=CONTRIDX,SRCH=CONTRNO
DBDGEN
FINISH
END
```

If you have applications that process a secondary index by using symbolic pointers as a database, you will need to make other changes to your DBDs.

Converting non-unique secondary index keys to unique keys

HALDB secondary indexes must have unique keys.

If your secondary indexes use non-unique keys, complete the following steps to convert them to unique keys:

1. Add a subsequence field by adding the /SX system-related field to the XDFLD statement in the indexed database DBD. This field adds 8 bytes to the length of the sequence field in the index. In the examples, the BYTES parameter in the FIELD statement is increased by 8 bytes for this conversion.
2. On the FIELD statement, change the value of the third subparameter from M to U on the NAME parameter. The U parameter indicates that only unique values are allowed in the sequence field of the segment.

The following example shows a non-HALDB secondary index DBD with non-unique keys.

Figure 304. Example: non-HALDB secondary index DBD with non-unique keys

```
DBD     NAME=XSI3,ACCESS=INDEX
DATASET DD1=XSI301,OVFLW=XSI302
SEGM    NAME=XSNAM,BYTES=6,PARENT=0
FIELD   NAME=(XSNAME,SEQ,M),START=1,BYTES=6
LCHILD  NAME=(PERF,XPER01),INDEX=NAMX1,POINTER=SNGL
DBDGEN
FINISH
END
```

The following figure shows the DBD for the secondary index after it has been converted to HALDB.

Figure 305. Example: HALDB secondary index DBD with unique keys

```
DBD     NAME=XSI3,ACCESS=PSINDEX
SEGM    NAME=XSNAM,BYTES=14,PARENT=0
FIELD   NAME=(XSNAME,SEQ,U),START=1,BYTES=14
LCHILD  NAME=(PERF,XPER01),INDEX=NAMX1,POINTER=SNGL,RKSIZE=12
DBDGEN
FINISH
END
```

Modifying the PSINDEX DBD for the larger HALDB /SX field

If you already use a /SX field to create unique keys for the secondary index you are converting to HALDB, increase the BYTES parameter on both the SEGM statement and the FIELD statement by 4 bytes. In non-HALDB secondary indexes, a /SX field contains a 4-byte RBA. In a HALDB secondary index, a /SX field contains an 8-byte ILK.

You do not need to change the XDFLD statement or the /SX FIELD statement in the indexed DBD. The BYTES parameter on the /SX FIELD statement is not required and is ignored if it is specified.

The following example shows the DBD for an indexed HDAM database with a /SX field defined.

Figure 306. Example: HDAM DBD with /SX field

```
DBD   NAME=VEHICLE,ACCESS=(HDAM,OSAM),                    X
      RMNAME=(DFSHDC40,2,500,)
DATASET DD1=VEHICLE1,BLOCK=1648,SCAN=0
SEGM  NAME=AUTO,BYTES=54,PTR=TB
FIELD NAME=(ID,SEQ,U),BYTES=10,START=1
FIELD NAME=MAKE,BYTES=20,START=11
FIELD NAME=MODEL,BYTES=20,START=31
FIELD NAME=YEAR,BYTES=4,START=51
FIELD NAME=/SX1
LCHILD NAME=(MAKEMOD,VEHSI),PTR=INDX
XDFLD NAME=MMIDX,SRCH=(MAKE,MODEL),SUBSEQ=/SX1
DBDGEN
FINISH
END
```

The following example shows the DBD for a non-HALDB secondary index that uses the /SX field that is defined in the preceding figure.

Figure 307. Example: non-HALDB secondary index DBD using a /SX field

```
DBD    NAME=VEHSI,ACCESS=INDEX
DATASET DD1=VEHSI1,DEVICE=3390,SIZE=8192
SEGM  NAME=MAKEMOD,BYTES=44,PARENT=0
FIELD NAME=(NAMES,SEQ,U),BYTES=44,START=1
LCHILD NAME=(AUTO,VEHICLE),INDEX=MMIDX
DBDGEN
FINISH
END
```

The following example shows the DBD from Figure 306 on page 863 after it has been converted to HALDB. The only changes are those made for all HDAM DBDs when they are converted to PHDAM. That is, the ACCESS parameter on the DBD STATEMENT is changed to PHDAM and the DATASET statement is deleted.

Figure 308. Example: PHDAM DBD with /SX field

```
DBD    NAME=VEHICLE,ACCESS=(PHDAM,OSAM),                          X
       RMNAME=(DFSHDC40,2,500,)
SEGM  NAME=AUTO,BYTES=54,PTR=TB
FIELD NAME=(ID,SEQ,U),BYTES=10,START=1
FIELD NAME=MAKE,BYTES=20,START=11
FIELD NAME=MODEL,BYTES=20,START=31
FIELD NAME=YEAR,BYTES=4,START=51
FIELD NAME=/SX1
LCHILD NAME=(MAKEMOD,VEHSI),PTR=INDX
XDFLD NAME=MMIDX,SRCH=(MAKE,MODEL),SUBSEQ=/SX1
DBDGEN
FINISH
END
```

The following example shows the DBD from Figure 307 after it has been converted to HALDB. As with other secondary index DBD conversions, the ACCESS parameter on the DBD statement is changed to PSINDEX, the DATASET statement is deleted, and the RKSIZE parameter is added to the LCHILD statement. Because the /SX field is used as a subsequence field, the BYTES parameters on the SEGM and FIELD statements are increased by four.

Figure 309. Example: HALDB secondary index DBD using a /SX field

```
DBD    NAME=VEHSI,ACCESS=PSINDEX
SEGM  NAME=MAKEMOD,BYTES=48,PARENT=0
FIELD NAME=(NAMES,SEQ,U),BYTES=48,START=1
LCHILD NAME=(AUTO,VEHICLE),INDEX=MMIDX,RKSIZE=10
DBDGEN
FINISH
END
```

Registering the indexed HALDB master database with DBRC

The indexed HALDB master database must be registered with DBRC.

Register the HALDB master database with DBRC by using either the Partition Definition utility or the DBRC batch command INIT.DB.

Defining the partitions of the indexed database to DBRC

HALDB partition definitions are stored in DBRC RECON data sets.

When defining partitions, you must have update authority for the RECON data sets.

Define the partitions to DBRC by using either the Partition Definition utility or the DBRC batch command INIT.PART.

Registering the PSINDEX HALDB master database with DBRC

The indexed PSINDEX HALDB master database must be registered with DBRC.

Register the HALDB PSINDEX with DBRC by using either the Partition Definition utility or the DBRC batch command INIT.DB.

Defining the partitions of the PSINDEX database to DBRC

HALDB partition definitions for the PSINDEX are stored in DBRC RECON data sets.

HALDB PSINDEX records are often much larger than non-HALDB secondary index records for two reasons:

- The pointers in PSINDEX records are different. Each PSINDEX record includes the 28-byte extended pointer set (EPS). Non-HALDB secondary index records have either a 4-byte direct pointer or a symbolic pointer. Symbolic pointers are the length of the concatenated key.
- The root keys of the target segments are stored in the PSINDEX records. Non-HALDB secondary index records do not store the root keys. When allocating your HALDB secondary index partitions, you must account for this increased size.

The total size of the secondary index is usually easy to estimate. All secondary index entries are fixed length. The DBDGEN utility reports this length. The total length is indicated by the RECORDSIZE under RECOMMENDED VSAM DEFINE CLUSTER PARAMETERS in the output of the DBDGEN utility for the HALDB database. The number of entries does not change during the conversion. You can determine the current number of entries from the output of several utilities. When you re-create a secondary index as part of a reorganization process, the HISAM Reload utility or any tool that you use reports this information.

Alternatively, you can use the REC-TOTAL value from the z/OS LISTCAT command to find the number of records in the existing secondary index data set.

Finally, you can use the HALDB Migration Aid utility to find the number of records. You can use the number of records, their length, and free space requirements to estimate the size of an entire HALDB secondary index. For example, if you had 1 000 000 entries, a record length of 48 bytes, and wanted an additional 25% for free space, the index would require 60 000 000 bytes.

The size requirements of each partition depend on the number of index entries in the partition. You can estimate this from the output of the HALDB Migration Aid utility (DFSMAID0) or by other means. If you already know the number of secondary index entries in a key range, you do not need to use the Migration Aid utility. Otherwise, you should use the utility.

The HALDB Migration Aid utility provides accurate information about the number of records in a key range and key range boundaries. It does not provide accurate information about the number of bytes that are required for a segment or partition. For this reason, you should not use the MAX control statement when reading a secondary index. If you know the number of partitions that you want, use the NBR control statement. If you know the high keys that you want, use KR control statements. When an NBR statement is used, the high keys that are reported for each partition are accurate. When you use KR control statements, the number of segments reported for each partition is accurate.

The following example output shows some of the output of the HALDB Migration Aid utility for a secondary index. This output was generated with a NBR=4 control statement. The reports for partitions 1, 2, and 3 are omitted here. Only the report for partition 4 and the total database are shown. The numbers of segments that are reported in the segments column are correct. Partition 4 will contain 158518 secondary index segments, and the entire index will contain 634078 segments. The information in the bytes and prefix-incr columns is incorrect, as it is for all such reports for secondary indexes. Nevertheless, you need only the number of segments in a partition to calculate its space requirements.

```
partition 4 :

   minimum key =

      +0000   f0f0f1f6 f0f8f1f1 f5f0f0f0 f0f0f1f3  |0016081150000013|
      +0010   f1f7                                  |17              |

   maximum key =

      +0000   f0f0f2f1 f1f0f3f0 f0f0f0f0 f0f0f0f5  |0021103000000005|
      +0010   f4f6                                  |46              |

                      segments        bytes   prefix-incr   length-incr
   1) 'ORDRCUST'        158518      3804432       1268144             0
SUM)                    158518      3804432       1268144             0

--------------------------------------------------------------------------------

sum of partitions:
                      segments        bytes   prefix-incr   length-incr
   1) 'ORDRCUST'        634078     15217872       5072624             0
SUM)                    634078     15217872       5072624             0


--------------------------------------------------------------------------------
```

You can use the record size that is reported by the DBDGEN utility and the number of segments for a partition to estimate the size requirement for the partition. If necessary, you can add free space and room for expansion of the partition.

Important: Do not use the numbers of bytes and prefix-incr for secondary indexes reported by the HALDB Migration Aid utility to estimate the size requirements for a partition, because these reports are inaccurate. The reports are inaccurate because the segment prefixes for secondary indexes are larger than the prefixes for segments in the indexed database. The HALDB Migration Aid utility does not adjust for secondary indexes. The HALDB Migration Aid utility can be used to find high keys and the number of secondary index entries for each partition.

Allocating the indexed database data sets
Allocate the database data sets for each partition in the indexed database.

The data sets you need to allocate are the ILDS and, if your HALDB database is a PHIDAM database, the primary index.

Related concepts:

"Naming conventions for HALDB partitions, ddnames, and data sets" on page 26

Related tasks:

"Allocating PSINDEX VSAM KSDS data sets"

"Allocating database data sets" on page 854

Allocating PSINDEX VSAM KSDS data sets

Allocate a VSAM KSDS data set for each partition of the PSINDEX by using the IDCAMS DEFINE statement.

The following parameters are required when allocating the data sets for the PSINDEX partitions:

- KEY values
- RECORDSIZE values
- INDEXED
- REUSE

The output of the DBDGEN utility is useful when you allocate secondary index data sets. The output for the secondary index lists the required parameters for the IDCAMS definition under RECOMMENDED VSAM DEFINE CLUSTER PARAMETERS. The following example shows IDCAMS DEFINE statements to allocate a secondary index data set.

```
DEFINE CLUSTER(                         -
    NAME(JOUKO3.HALDB.DB.PEOSKSI.A00001) -
    INDEXED                             -
    RECORDSIZE(86 86)                   -
    CYL(20 5)                           -
    SHAREOPTIONS(3 3)                   -
    REUSE                               -
    KEY(29,50)                          -
    FREESPACE(10,10)                    -
    CONTROLINTERVALSIZE(4096)           -
    VOLUMES(TOTIMN)                     -
)
```

Related tasks:

"Allocating logically related database data sets" on page 874

"Allocating the indexed database data sets" on page 866

Initializing the partitions

Before you can use HALDB partitions, you must initialize them.

To initialize partitions, you can use either the IMS Database Prereorganization utility (DFSURPR0) or the IMS HALDB Database Data Set Initialization utility (DFSUPNT0). Alternatively, you can use the IBM IMS High Performance Load tool to initialize partitions.

Related concepts:

"HALDB partition initialization" on page 185

Selecting an ILDS update method

You have three options for how you update the ILDS data set when you load the target database of a secondary index. The option you choose impacts the performance of the load process.

Use one of the following control statement specifications to select an update method for the ILDS:

- To update the ILDS as each target segment is loaded into the indexed database by the HD Reorganization Reload utility (DFSURGL0), do not specify an ILDS control statement when you run the HD Reorganization Reload utility. This method is the most time consuming.

- To have the HD Reorganization Reload utility update the ILDS after the indexed database has been reloaded, specify the ILDSMULTI control statement when you run the HD Reorganization Reload utility. This process uses a multi-threaded update process, which can be much more efficient than updating the entries as the target segments are loaded for the following reasons:
  - The writes of the ILDS entries to different partitions are overlapped
  - The writes are sequential
  - The writes are done in load mode, which avoids the overhead of CI and CA splits

  Additionally, you can create free space. Future reorganizations might benefit from having this free space for their updates to the ILDSs.

  The ILDSMULTI control statement is valid only when converting a database to HALDB.

- To update the ILDS by a means other than the HD Reorganization Reload utility after the indexed database has been reloaded, specify the NOILDS control statement. If you choose this option, you can use the HALDB Index/ILDS Rebuild utility (DFSPREC0) to update the ILDS.

  Using the NOILDS control statement provides the fastest reload, and the HALDB Index/ILDS Rebuild utility can update the ILDS for each partition in parallel; however, the HALDB Index/ILDS Rebuild utility reads each partition to update its ILDS. Optionally, the HALDB Index/ILDS Rebuild utility can rebuild the ILDS in VSAM load mode, which can improve performance and includes free space in the ILDS.

  You can also use the NOILDS control statement with the HD Reorganization Reload utility when reorganizing an existing HALDB database.

Recommendation: If you are converting a database to HALDB that has a large number of secondary index entries, use the ILDSMULTI control statement with the HD Reorganization Reload utility when you reload the database.

Related tasks:

"Loading the indexed database and its secondary indexes"

Loading the indexed database and its secondary indexes

To load the indexed database and its secondary index, use the HD Reorganization Reload utility (DFSURGL0).

The input to the HD Reorganization Reload utility is the sorted output from the HD Reorganization Unload utility.

Other than choosing a method of updating the ILDS, there are no special considerations for loading indexed databases and their secondary indexes.

After the migration reload, the pointers in the secondary index are no longer accurate and IMS cannot use them to retrieve the target segments in the indexed database until they are healed; however, you do not need to do anything to fix them. To heal them, IMS waits until the target segment is needed and then locates

the target segment through the ILDS of the target partition, at which time IMS updates the target segment's pointer in the secondary index.

Related tasks:

"Selecting an ILDS update method" on page 867

Creating image copies

The reload process sets the *image copy needed* flag for each data set other than ILDSs and PHIDAM primary indexes.

Create image copies of the flagged database data sets and the PSINDEX data sets.

Cleaning up DFSMDA members and HIDAM primary index DBDs after converting to PSINDEX

After the conversion process is complete and you are certain that you will not need to revert the database to its non-HALDB state, you can delete any DFSMDA members for the old database and discard the DBDs for any HIDAM primary index.

Related tasks:

"Cleaning up DFSMDA members and HIDAM primary index DBDs" on page 855

Converting logically related HDAM or HIDAM databases to HALDB

Converting logically related HDAM or HIDAM databases to HALDB PHDAM or PHIDAM is similar to converting a simple database to HALDB; however, HALDB support for logical relationships might require changes to your logical relationships and all of the logically related databases must be converted to HALDB at the same time.

Create image copies of your database data sets and back up your RECON data sets and your existing DBD definitions.

In addition to the steps for converting logically related databases to HALDB, this section describes the concepts and issues that you should consider before beginning the conversion process.

Databases that are logically related to each other must be converted to HALDB at the same time. IMS does not support logical relationships between HALDB and non-HALDB databases.

HALDB supports only two types of logical relationships: unidirectional and bidirectional with physical pairing. HALDB does not support virtual pairing. You must convert bidirectional logical relationship that use virtual pairing to bidirectional logical relationships that use physical pairing.

HALDB does not use symbolic pointers or hierarchical pointers. HALDB PHIDAM databases do not support the use of twin forward only pointers (TWIN or T).

You do not need to change the DBDs for logical databases when the physical databases that they reference are migrated. DBDs for logical databases have ACCESS=LOGICAL in their DBD statements.

Related tasks:

"Backing up existing database information" on page 849

"Converting simple HDAM or HIDAM databases to HALDB PHDAM or PHIDAM" on page 850

Unloading the existing database

Unload the existing databases by using the HD Reorganization Unload utility (DFSURGU0) with the appropriate control statement.

The control statements you can use with the DFSURGU0 utility include:

- If your logically related databases do not use secondary indexes, use the MIGRATE=YES control statement.
- If your logically related databases use secondary indexes, use the MIGRATX=YES control statement to create unload files for the secondary indexes. You must also complete the steps documented in "Converting HDAM or HIDAM databases with secondary indexes to HALDB" on page 856.

When you unload a database for conversion to HALDB that has a logical child, its logically related database might also be read. The HD Reorganization Unload utility reads the logically related database when any of the following conditions exists:

- The logical relationship uses physical pairing.
- The VIRTUAL option is specified (the logical parent's concatenated key is not stored in the logical child).
- The logical relationship uses symbolic pointing.
- The database that is being unloaded contains the virtual logical child.

Stated another way, there are only two cases in which the logically related database is not read. They are:

- The database that is being unloaded contains the real logical child of a virtually paired relationship that uses direct pointing, and the logical parent's concatenated key is stored in this logical child.
- The logical relationship is unidirectional and uses direct pointing, and the logical parent's concatenated key is stored in the logical child.

When the HD Reorganization Unload utility reads the logically related database, either the logical parent or the paired logical child of each logical child is read. That is, each instance of a logical relationship requires a read of the related data. These reads are random. Each read might require a physical I/O, which can greatly increase the elapsed time of the unload.

You must provide buffer pools for any logically related database that is read by the unload. The DFSVSAMP DD statement for the HD Reorganization Unload utility must include definitions for these buffers.

The following figure illustrates a migration of two logically related databases to HALDB databases.
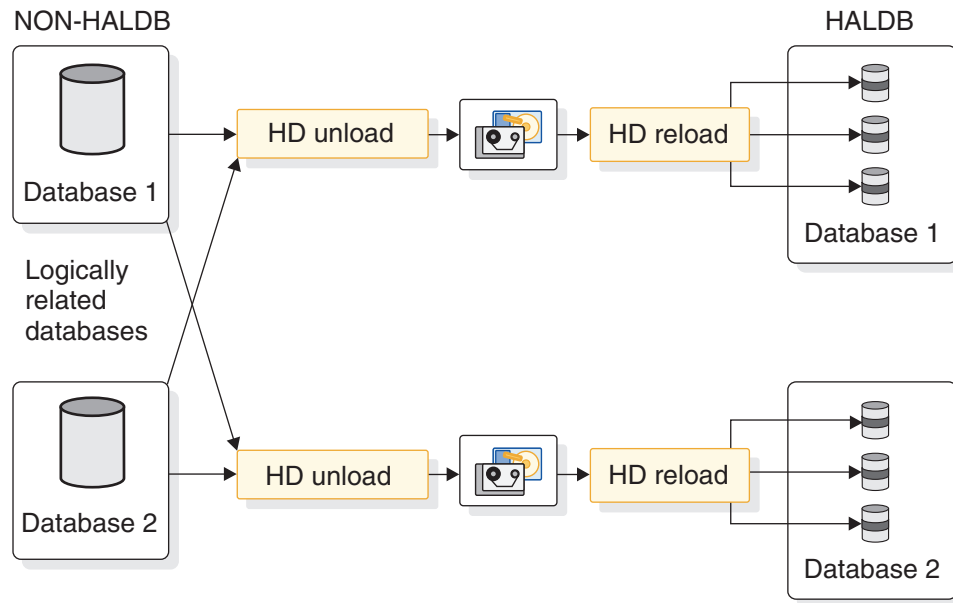
Figure 310. Migrating databases with logical relationships

Defining DBD statements for logically related HALDB databases

Define the DBD statements for each logically related HALDB database.

The statements that you need to modify are:
- The DBD statement
- The SEGM statement
- The LCHILD statement
- The XDFLD statement
- The FIELD statement

HALDB databases do not use the DATASET statement.

Changing pointer options for logical relationships

Logically related databases might require changes to pointers in addition to the changes that are required for databases that do not have logical relationships.

HALDB does not use symbolic pointers. You must change any logical relationship that has been implemented with symbolic pointing. You can do this by adding the LPARNT keyword to the PTR parameter on the SEGM statement for the logical child.

Although symbolic pointing is not used, the concatenated key of the logical parent is stored in the logical child, regardless of the keywords that you use in the PARENT parameter. That is, PHYSICAL or P is used, even when you specify VIRTUAL or V. To avoid the warning message that is issued when VIRTUAL or V is specified, you should specify the PHYSICAL or P keyword.

Recognizing virtual pairing

You can determine if you have virtual pairing by examining your DBDs. If your HDAM or HIDAM database contains a SEGM statement with a SOURCE parameter, this segment is a virtual logical child. The first subparameter of the SOURCE parameter identifies the real logical child. The third subparameter identifies the database in which the real logical child resides. In the following example of a HIDAM DBD with a logical relationship, the NAMESKIL segment in the DBD contains a SOURCE parameter. It is a virtual logical child.

The source parameter references the SKILNAME segment in the SKILLINV database in Figure 312, in which SKILNAME is a real logical child.

Figure 311. Example: HIDAM DBD with a logical relationship

```
DBD     NAME=PAYROLDB,ACCESS=HIDAM
DATASET DD1=PAYHIDAM,BLOCK=1648,SCAN=3
SEGM    NAME=NAMEMAST,PTR=TWINBWD,RULES=(VVV),              X
        BYTES=150
LCHILD NAME=(INDEX,INDEXDB),PTR=INDX
LCHILD NAME=(SKILNAME,SKILLINV),PAIR=NAMESKIL,PTR=DBLE
FIELD  NAME=(EMPLOYEE,SEQ,U),BYTES=60,START=1,TYPE=C
SEGM    NAME=NAMESKIL,PARENT=NAMEMAST,PTR=PAIRED,           X
        SOURCE=((SKILNAME,DATA,SKILLINV))
FIELD  NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
DBDGEN

FINISH

END
```

Figure 312. Example: HDAM DBD with a logical relationship

```
DBD     NAME=SKILLINV,ACCESS=HDAM,RMNAME=(DFSHDC40,1,500,824)
DATASET DD1=SKILHDAM,BLOCK=1648,SCAN=0
SEGM  NAME=SKILMAST,BYTES=31,PTR=TWINBWD
FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
SEGM   NAME=SKILNAME,                                       X
       PARENT=((SKILMAST,DBLE),(NAMEMAST,P,PAYROLDB)),      X
       BYTES=80,PTR=(LPARNT,LTWINBWD,TWIN)
FIELD NAME=(EMPLOYEE,SEQ,U),START=1,BYTES=60,TYPE=C
DBDGEN
FINISH
END
```

Converting virtual pairing to physical pairing

If your existing logically related databases use virtual pairing, you need to modify the DBD statements to convert them to HALDB.

To convert a virtually paired logical relationship to physical pairing, complete the following steps:

1. Add an LCHILD statement under the parent of the segment that was the real logical child. In Figure 311, this is the SKILMAST segment. Make the following changes to this LCHILD statement:

   • In the new LCHILD statement, include a NAME parameter that specifies the segment that had the SOURCE parameter. In Figure 311, this is NAMESKIL.

   • In the PAIR parameter, specify the same segment that was specified in the SOURCE parameter. This segment was the real logical child. In Figure 311, this is SKILNAME.

2. Find the LCHILD statement that references the segment that was the real
   logical child. In Figure 311 on page 872, this is the LCHILD statement in the
   PAYROLDB database that includes NAME=(SKILNAME,SKILLINV). Make the
   following changes to this LCHILD statement:

   • Delete the PTR parameter.
   • If there is a RULES parameter, delete it.

3. Find the SEGM statement that defined the virtual logical child. In Figure 311 on
   page 872, this is the NAMESKIL segment. Make the following changes to the
   SEGM statement:

   • Add a BYTES parameter. The lengths of the fixed intersection data in the
     paired logical children must be the same. The BYTES parameter includes
     both the fixed intersection data and the logical parent's concatenated key
     (LPCK). You can calculate the BYTES value for this segment by taking the
     BYTES value from the paired logical child's SEGM statement, adding the
     LPCK size for this segment, and subtracting the LPCK size for the paired
     logical child. In Figure 311 on page 872, the paired logical child for the
     NAMESKIL segment is the SKILNAME segment. Its size is 80 bytes. The
     LPCK for the NAMESKIL segment is the TYPE field in the SKILMAST
     segment. Its size is 21 bytes. The LPCK for the SKILNAME segment is the
     EMPLOYEE field in the NAMEMAST segment. Its size is 60 bytes. The
     BYTES parameter for the NAMESKIL segment should be 41, which is 80 + 21
     - 60.
   • Delete the SOURCE parameter.
   • Change the PARENT parameter so that it also references its logical parent. In
     Figure 311 on page 872, this is the SKILMAST segment. The PARENT
     parameter should also include the PHYSICAL or P keyword.
   • Change the PTR parameter specifications. Include PAIRED. Include TWIN, T,
     TWINBWD, TB, NOTWIN, or NT. Do not use NOTWIN or NT unless you
     will have a maximum of only one instance of this segment type under a
     parent.

4. Find the SEGM statement for the segment that was the real logical child. In
   Figure 311 on page 872, this is the SKILNAME segment. Make the following
   changes to this SEGM statement:

   • Change the PARENT parameter so that neither SNGL nor DBLE is specified.
     If VIRTUAL or V was specified, change this to PHYSICAL or P.
   • Change the PTR parameter specifications. Delete LTWIN, LT, LTWINBWD, or
     LTB keywords if they exist. Specify both the LPARNT and PAIRED
     keywords.

Figure 313 and Figure 314 on page 874 show the DBDs from Figure 311 on page
872 and Figure 312 on page 872 after they have been converted to HALDB
databases with physical pairing. In addition to the changes that were made for
logical relationships, other changes for the conversion to HALDB have been made.
These are changes in the ACCESS parameter on the DBD statements and deletion
of the DATASET statements.

Figure 313. Example: PHIDAM DBD with a logical relationship

```
DBD    NAME=PAYROLDB,ACCESS=PHIDAM
SEGM   NAME=NAMEMAST,PTR=TWINBWD,RULES=(VVV),                    X
       BYTES=150
LCHILD NAME=(SKILNAME,SKILLINV),PAIR=NAMESKIL
FIELD  NAME=(EMPLOYEE,SEQ,U),BYTES=60,START=1,TYPE=C
SEGM   NAME=NAMESKIL,                                            X
```

```
                    PARENT=((NAMEMAST),(SKILMAST,P,SKILLINV)),              X
                    BYTES=41,PTR=(TWIN,PAIRED)
             FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
             DBDGEN
             FINISH
             END
```

Figure 314. Example: PHDAM DBD with a logical relationship

```
             DBD   NAME=SKILLINV,ACCESS=PHDAM,RMNAME=(DFSHDC40,1,500,824)
             SEGM  NAME=SKILMAST,BYTES=31,PTR=TWINBWD
             FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
             LCHILD NAME=(NAMESKIL,PAYROLDB),PAIR=SKILNAME
             SEGM  NAME=SKILNAME,                                          X
                    PARENT=((SKILMAST),(NAMEMAST,P,PAYROLDB)),             X
                    BYTES=80,PTR=(TWIN,PAIRED)
             FIELD NAME=(EMPLOYEE,SEQ,U),START=1,BYTES=60,TYPE=C
             DBDGEN
             FINISH
             END
```

Related tasks:

"Defining HALDB database DBD statements" on page 851

Deleting the database information from the RECON data set

Delete database information for each logically related database from the RECON
data sets.

This information must be deleted prior to the new HALDB database being
registered in the next step. You can delete the database information by using the
DBRC DELETE.DB command.

Registering each HALDB master database with DBRC

Register each HALDB master database with DBRC by using either the Partition
Definition utility or the DBRC batch command INIT.DB.

Defining the partitions to DBRC

Define the HALDB partitions in the DBRC RECON data set.

When defining partitions, you must have update authority for the RECON data
sets.

Define the partitions to DBRC by using either the Partition Definition utility or the
DBRC batch command INIT.PART.

Allocating logically related database data sets

Allocate the database data sets, including the indirect list data set for each partition
and the primary index for PHIDAM databases.

Related concepts:

"Naming conventions for HALDB partitions, ddnames, and data sets" on page 26

Related tasks:

"Allocating database data sets" on page 854

"Allocating PSINDEX VSAM KSDS data sets" on page 867

Initializing the partitions

Before you can use HALDB partitions, you must initialize them.

Initialize the partitions in your databases by using either the Database
Prereorganization utility (DFSURPR0) or the HALDB Partition Data Set
Initialization utility (DFSUPNT0).

Related concepts:

"HALDB partition initialization" on page 185

Selecting an ILDS update method

You can update the ILDS by using one of three possible methods when you load
the logically related databases.

The option that you choose impacts the performance of the load process. Because
loading logically related databases takes much longer than it does for other
databases, the option that you choose might have a considerable impact on
performance.

Use one of the following methods to update your ILDS:

- To update the ILDS as each logical parent of a unidirectional relationship is
  loaded or as each logical child of a bidirectional relationship is loaded into the
  database by the HD Reorganization Reload utility (DFSURGL0), do not specify
  an ILDS control statement when you run the HD Reorganization Reload utility.
  This method is the most time consuming.
- To have the HD Reorganization Reload utility update the ILDS after the logically
  related database has been reloaded, specify the ILDSMULTI control statement
  when you run the HD Reorganization Reload utility. This process uses a
  multi-threaded update process, which can be much more efficient than updating
  the entries as the target segments are loaded for the following reasons:
  - The writes of the ILDS entries to different partitions are overlapped
  - The writes are sequential
  - The writes are done in load mode, which avoids the overhead of CI and CA
    splits

  Additionally, you can create free space. Future reorganizations might benefit
  from having this free space for their updates to the ILDSs.

  The ILDSMULTI control statement is valid only when converting a database to
  HALDB.
- To update the ILDS by a means other than the HD Reorganization Reload utility
  after the logically related databases have been reloaded, specify the NOILDS
  control statement. If you choose this option, you can use the HALDB
  Index/ILDS Rebuild utility (DFSPREC0) to update the ILDS.

  Using the NOILDS control statement provides the fastest reload, and the
  HALDB Index/ILDS Rebuild utility can update the ILDS for each partition in
  parallel; however, the HALDB Index/ILDS Rebuild utility reads each partition to
  update its ILDS. Optionally, the HALDB Index/ILDS Rebuild utility can rebuild
  the ILDS in VSAM load mode, which can improve performance and includes
  free space in the ILDS.

  You can also use the NOILDS control statement with the HD Reorganization
  Reload utility when reorganizing an existing HALDB database.

Related tasks:

"Loading each database as a HALDB database"

Loading each database as a HALDB database

Use the HD Reorganization Reload utility (DFSURGL0) to load HALDB databases
with logical relationships.

Unless you specify otherwise, when a logical parent of a unidirectional relationship is loaded or a logical child of a bidirectional relationship is loaded into a partition, an entry is created in the ILDS of that partition. Otherwise, no special considerations exist for loading logically related databases during a conversion.

After the migration reload, the pointers in the logically related databases are no longer accurate and IMS cannot use them to retrieve the target segments until they are healed; however, you do not need to do anything to fix them. To heal them, IMS waits until the target segment is needed and then locates the target segment through the ILDS of the target partition, at which time IMS corrects the target segment's pointer.

Related tasks:

"Selecting an ILDS update method" on page 875

Creating image copies

Create image copies of the database data sets.

The *image copy needed* flag is set for each database data set other than ILDSs and PHIDAM primary indexes. You must take image copies of the flagged data sets.

Cleaning up DFSMDA members and HIDAM primary index DBDs

HALDB databases do not use DFSMDA members for dynamic allocation and they do not require a separate DBD for the primary index of a PHIDAM database.

HALDB databases use information stored in the DBRC RECON data set to enable dynamic allocation of the database data sets and do not use DFSMDA members. After the conversion process is complete and you are certain that you will not need to revert the database to its non-HALDB state, you can delete any DFSMDA members for the old database and discard the DBDs for any HIDAM primary index.

IMS obtains the information that is required to generate the PHIDAM primary index from the PHIDAM DBD. If you converted a HIDAM database to a HALDB PHIDAM database, after the conversion process is complete and you are certain that you will not need to revert the database to its non-HALDB state, you can discard the DBD for the primary index of the old HIDAM database.

Related tasks:

"Cleaning up DFSMDA members and HIDAM primary index DBDs" on page 855

Changing the database name when converting a simple database to HALDB

When converting a database to HALDB, you can choose to change the database name. However, if you do, you will be creating extra work for yourself because you will also have to change all of your PCBs to conform to the new name. You should continue to use the same database name.

To change the name of a database when you convert the database to HALDB, follow these steps:

1. Create a RECON list before deleting the records for the database. The old information is retained in RECON as long as necessary.
2. Unload the old database.
3. Remove the DBD from DBDLIB and ACBLIB.
4. Delete all MDA members that refer to the old database.

5. Perform a DBDGEN on the old database name as a logical database with the source being the new HALDB database.
6. Define the HALDB database by using DBDGEN, ACBGEN, and either the HALDB Partition Definition utility or the DBRC commands INIT.DB and INIT.PART.

Restoring a non-HALDB database after conversion

The process of restoring HDAM or HIDAM databases that were migrated to PHDAM or PHIDAM is referred to as *fallback*.

Fallback supports the conversion of the following types of logical relationships:
- Unidirectional HALDB databases to unidirectional non-HALDB databases
- Physically paired HALDB databases to physically paired non-HALDB databases

Fallback from HALDB maintains the order of physical twin segments, including segments that are non-keyed and those that have a non-unique key.

Primary indexes are re-created, not unloaded. Secondary indexes are re-created by the reload utility process. User data is not preserved.

Understanding the requirements of fallback

Before attempting to restore a HALDB database to a non-HALDB database, there are several requirements and considerations you should take into account.

The requirements for fallback include:
- You must perform a concurrent fallback of all of the databases that are logically related.
- If logical children or secondary indexes exist, prefix resolution and prefix update utilities must be installed on your system.
- Before you can use the restored database, you must restore all of the related databases and secondary index databases.

Recommendation: Before you use the database, but after you reload and perform any prefix resolution or prefix updates, you should take an image copy of all data sets, including the primary index, by using one of the image copy utilities. The image copy utilities run DBRC to validate input and record results, which ensures that you have a backup copy of the database that can serve as an effective point of recovery in case of failure.

Restoring the database before updates are made

Restoring a database to its HDAM or HIDAM structure is simplest if you have not updated the database since you converted it to HALDB.

To fall back to HDAM or HIDAM if the database has not been updated since it was converted to HALDB, perform the following steps:
1. Change the DBD to its former specifications
2. Delete the HALDB database information from the RECON data set
3. If the non-HALDB database data sets have been deleted, restore them from the last image copies and any necessary logs. You will need logs only if the database was updated after you created the last image copies but before you started the conversion process.
4. Register the non-HALDB database with DBRC
5. Take an image copy of the database data sets

Restoring the database after updates are made

If you have updated the database after converting it to HALDB, your options for restoring the database to its non-HALDB state are limited.

If the data no longer fits in the VSAM 4 GB size limit or the OSAM 8 GB size limit of a non-HALDB database, you cannot restore it without taking extraordinary steps. The natural growth of the database or even the conversion of virtual pairs to physical pairs can cause the database to grow beyond these non-HALDB size limits. In the unlikely event that you need to restore the non-HALDB database structure and your data no longer fits into the non-HALDB database size limitations, contact IBM Software Support.

If the amount of data is still within the space limitations of a non-HALDB database, you can restore the database by completing the following steps:

1. Unload the database by using the HD Reorganization Unload utility (DFSURGU0) and the FALLBACK=YES control statement. The HD Reorganization Reload utility locates the paired logical children and saves the information that is needed for fallback in the prefix of the output data. The prefix that is created by the HD Reorganization Unload utility contains the information that is required to create the new segment prefix when the data is reloaded.
2. Restore the DBD of the non-HALDB database.
3. Delete the HALDB database information from the RECON data set.
4. Register the non-HALDB database with DBRC.
5. Run the Database Prereorganization utility (DFSURPR0) and specify the DBR control statement.
6. Reload the non-HALDB database by using the HD Reorganization Reload utility (DFSURGL0).
7. If the database has secondary indexes, restore the secondary indexes by rebuilding them after the fallback process is complete for the indexed database.
8. If the database is logically related to another database, you must perform fallback for all of the logically related databases at the same time.
9. Take an image copy of the database data sets.

Related tasks:

"Restoring a secondary index database"

"Restoring a database that uses logical relationships" on page 879

Restoring a secondary index database

To restore a secondary index database to its non-HALDB state, rebuild it as part of the fallback process for the indexed database.

You can use the same rebuilding process that you would use when you reorganize a non-HALDB database.

You can run one of the following utilities to rebuild the secondary index database:
- IMS Prefix Resolution
- HISAM Unload
- HISAM Reload
- IMS Index Builder
- Any tool that builds secondary indexes from the indexed database

The fallback process for the indexed database is the same as that described in the following topics:
- "Restoring the database before updates are made" on page 877
- "Restoring the database after updates are made" on page 878

Related tasks:

Restoring a database that uses logical relationships

To restore logically related databases, you use the normal utilities for resolving logical relationships as part of a reorganization of a non-HALDB database.

You can use the following utilities:
- Database Prereorganization utility (DFSURPR0)
- Database Prefix Resolution utility (DFSURG10)
- Database Prefix Update utility (DFSURGP0)

If your non-HALDB databases used virtual pairing, you probably converted the virtual pairing to physical pairing when you converted the logically related databases to HALDB, because HALDB does not support virtual pairing. Unfortunately, you cannot restore virtual pairing as part of the fallback process. The DBD statements for the non-HALDB database you are restoring must include the definitions for physical pairing. Moreover, the additional space that physically paired segments require can prevent you from being able to fall back.

To restore virtual pairing after a fallback, complete the following steps:
1. Perform a fallback on current physically paired databases.
2. Reorganize the current database.
3. Change the logical relationship to virtually paired databases.

Logical child segments have special considerations in the fallback process. Non-HALDB IMS databases offer a virtual key storage option that does not store the concatenated key of the logical parent in the logical child; in normal retrieval, the key is built and the user application has access to the concatenated key in the data. For all logical child segments that are unloaded, you must drop the logical parent's concatenated key if you choose the virtual key storage option. The unloaded segments are reloaded as real segments that are part of a physically paired relationship. An unload drops the logical parent's concatenated key only when the HD Reorganization Unload utility (DFSURGU0) performs a fallback unload.

You cannot restore the database from physical pairing directly to virtual pairing and preserve the logical sequence of the virtual logical child.

Related tasks:

Converting databases to DEDB

Converting a database to DEDB can be performed in a few steps; however, you need to perform a number of preliminary steps also.

If your database requires logical relationships, a secondary index, or fixed-length segments, DEDBs cannot be used.

You need to do the following before changing your database to DEDBs:

- Determine whether or not your application programs can tolerate the FH (data unavailable) status code.
- Determine whether or not your database can tolerate a randomizing routine (might not be a problem when changing from HDAM).
- Recalculate database space, particularly when using DEDB features such as partitioning and data set replication.
- Determine which pointers are available to use.

To change to DEDBs:

1. Unload your database using the existing DBD and one of the following:
   - Your unload program
   - The HD Reorganization Unload utility if database records are in physical root key sequence
2. Code a new DBD for the DEDBs.
3. Execute the DBD generation.
4. For non-VSAM data sets, delete the old database space and define the new database space. For VSAM data sets, delete the space allocated for the old clusters and define space for the new clusters.
5. Run the DEDB initialization utility (DBFUMIN0).
6. Run the user DEDB load program.

Part 6. Appendixes

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample

programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Programming interface information

This information documents Product-sensitive Programming Interface and Associated Guidance Information provided by IMS, as well as Diagnosis, Modification or Tuning Information provided by IMS.

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service. Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a section or topic, or by a Product-sensitive programming interface label. IBM requires that the preceding statement, and any statement in this information that refers to the preceding statement, be included in any whole or partial copy made of the information described by such a statement.

Diagnosis, Modification or Tuning information is provided to help you diagnose, modify, or tune IMS. Do not use this Diagnosis, Modification or Tuning information as a programming interface.

Diagnosis, Modification or Tuning Information is identified where it occurs, either by an introductory statement to a section or topic, or by the following marking: Diagnosis, Modification or Tuning Information.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Bibliography

This bibliography lists all of the publications in the IMS 15 library.

Title	Acronym	Order number
IMS Version 15 Application Programming	APG	SC27-6778
IMS Version 15 Application Programming APIs	APR	SC27-6779
IMS Version 15 Commands, Volume 1: IMS Commands A-M	CR1	SC27-6780
IMS Version 15 Commands, Volume 2: IMS Commands N-V	CR2	SC27-6781
IMS Version 15 Commands, Volume 3: IMS Component and z/OS Commands	CR3	SC27-6782
IMS Version 15 Communications and Connections	CCG	SC27-6783
IMS Version 15 Database Administration	DAG	SC27-6784
IMS Version 15 Database Utilities	DUR	SC27-6785
IMS Version 15 Diagnosis	DGR	GC27-6786
IMS Version 15 Exit Routines	ERR	SC27-6787SC19-4208
IMS Version 15 Installation	INS	SC27-6788
IMS Version 15 Licensed Program Specifications	LPS	GC27-6799
IMS Version 15 Messages and Codes, Volume 1: DFS Messages	MC1	GC27-6789
IMS Version 15 Messages and Codes, Volume 2: Non-DFS Messages	MC2	GC27-6790
IMS Version 15 Messages and Codes, Volume 3: IMS Abend Codes	MC3	GC27-6791
IMS Version 15 Messages and Codes, Volume 4: IMS Component Codes	MC4	GC27-6792
IMS Version 15 Operations and Automation	OAG	SC27-6793
IMS Version 15 Release Planning	RPG	GC27-6794
IMS Version 15 System Administration	SAG	SC27-6795
IMS Version 15 System Definition	SDG	GC27-6796
IMS Version 15 System Programming APIs	SPR	SC27-6797
IMS Version 15 System Utilities	SUR	SC27-6798

Index

Special characters

/CK operand 354
/SX field
   modifying PSINDEX DBD for
     HALDB 863
   sorting output of HD Reorganization
     Unload utility 859
/SX operand 354

A

abnormal termination in logical
  relationships 317
ACB (application control block)
   building 528
   description 528
   generation 528
ACB library
   HALDB alter function 760
ACB members
   online modification 796
ACBGEN (Application Control Block
  Generation) utility 801
   adding randomizing routines
     online 799
ACBs (application control blocks)
   IMS management of
     implementing database
       design 489, 490, 491, 492, 493
access methods
   BSAM (Basic Sequential Access
     Method) 567
   changing DL/I access methods 841
   converting DL/I access methods 841
   database
     hierarchical direct 144
   hierarchical direct 144
   HISAM 128
   IMS access methods 12, 124
   introduction 12
   operating system access methods 12
   OSAM (Overflow Sequential Access
     Method) 567
   OSAM (overflow sequential access
     methods)
     used by HD 160
   QSAM (Queued Sequential Access
     Method) 567
   VSAM
     HISAM 128
accessibility
   features xiii
   keyboard shortcuts xiii
accessing segments
   HDAM (Hierarchical Direct Access
     Method) 171
   HIDAM (Hierarchical Indexed Direct
     Access Method) 171
   HISAM (Hierarchical Indexed
     Sequential Access Method) 133

accessing segments *(continued)*
   HSAM (Hierarchical Sequential Access
     Method) 126
   PHDAM (Partitioned Hierarchical
     Direct Access Method) 171
   PHIDAM (Partitioned Hierarchical
     Indexed Direct Access Method) 171
add programs, use in loading a
  database 574
administration
   database
     overview 3
aids
   for test databases
     DL/I test program 555
AL (available length) field 162
algorithm
   first fit
     assigning VSO DEDB areas to data
       spaces 234
algorithms
   Fast Path buffer allocation
     algorithm 480
   Fast Path buffer allocation algorithm
     for BMPs 485
   Fast Path buffer allocation algorithm
     for CCTL threads 485
   HD space search algorithm 176
allocating data sets
   for HALDB conversion 854
   for PSINDEX databases 867
allocation
   IMS data sets 567
   OSAM data sets 569
alter
   HALDB alter and online change 762
ALTER option
   ACB library 760
   ACBs, IMS-managed 758, 766
   batch application programs 769, 770
   data-sharing environments 768
   DDL 758
   HALDB partition processing,
     description 765
   HD Reorganization Reload utility
     (DFSURGL0) 766, 767
   HD Reorganization Unload utility
     (DFSURGU0) 766, 767
   IMS configuration requirements 758
   IMS directory 761
   logical relationships 772
   offline reorganization 766, 767
   online change 762
   OSAM data set block sizes 762
   OSAM data sets
     correcting ALTERSZE values 764
     setting ALTERSZE values 763
   overview 757
   querying status 771
   secondary indexes 772

ALTER option *(continued)*
   steps for altering the structure of an
     online database 758, 759
   stopping before completion 771
   terminating before completion 771
   utilities 769, 770
   VSAM data set CI sizes 762
   VSAM data sets
     correcting ALTERSZE values 764
     setting ALTERSZE values 763
alternate PCB statement 526
AM status code 292, 325
anchor point area 163
application control block (ACB)
   building 528
   description 528
   generation 528
Application Control Block Maintenance
  utility
   database implementation
     building the ACBs 528
   deleting a data capture exit routine
     online 802
application control blocks (ACBs)
   IMS management of
     implementing database
       design 489, 490, 491, 492, 493
application development
   IMS Application Development Facility
     II 555
application I/O area
   calculating storage 468
application programs
   basic load program 577
   BMP
     database uncommitted updates
       restriction 527
   parallel partition processing 193
   processing HALDB databases 189
   selective partition processing 189
Application programs
   SHISAM
     restart 142
application requirements, analyzing 4,
  421, 429
AREA statement
   overview 521
areas
   adding online 806
   authorizing structure connections for
     DEDB VSO 231
   block-level sharing of
     authorizing structure connections
       for DEDB VSO 231
   copying data sets 204
   DEDB
     design guidelines 460
     opening 199
     preopening 199
     reopening 200
     starting 201

IBM. ®

Spine information:

IMS   Version 15.1.0   Database Administration   IBM