IMS/ESA

Administration Guide: Database Manager

Version 6

SC26-8725-04

IMS/ESA

Administration Guide: Database Manager

Version 6

SC26-8725-04

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xiii.

Fifth Edition (October 1999) (Softcopy Only)

This edition replaces and makes obsolete the previous edition, SC26-8725-03. This edition is available in softcopy only. The technical changes for this edition are summarized under “Summary of Changes” on page xix and are indicated by a vertical bar to the left of a change.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, BWE/H3

P.O. Box 49023

San Jose, CA, 95161-9023

U.S.A.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information that has been exchanged, should contact:

IBM Corporation
555 Bailey Avenue, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This book is intended to help the database administrator manage IMS databases.

This book also documents General-use Programming Interface and Associated Guidance Information, Product-sensitive Programming Interface and Associated Guidance Information, and Diagnosis, Modification or Tuning Information provided by IMS.

General-use programming interfaces allow the customer to write programs that obtain the services of IMS.

General-use Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

General-Use Programming Interface

General-use Programming Interface and Associated Guidance Information...

End of General-Use Programming Interface

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of IMS. 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 chapter or section or by the following marking:

Product-Sensitive Programming Interface

Product-sensitive Programming Interface and Associated Guidance Information...

End of Product-Sensitive Programming Interface

Diagnosis, Modification or Tuning Information is provided to help the customer diagnose, modify, or tune IMS.

Attention: 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 chapter or section or by the following marking:

Diagnosis, Modification or Tuning Information

Diagnosis, Modification or Tuning Information...

End of Diagnosis, Modification or Tuning Information

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
Advanced Function Printing
AFP
BookManager
CICS
DATABASE 2
DataPropagator Nonrelational
DPropNR
DB2
ES/9000
IBM
IMS
IMS/ESA
IMS Client Server/2
MVS
MVS/DFP
MVS/ESA
OS/390
PSF
RACF
Resource Measurement Facility
RMF
SP
VTAM

Product Names

In this book, the licensed program “DB2 for MVS/ESA” is referred to as “DB2”.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Preface

This book helps you administer your IMS/ESA databases. It describes how to design, implement, and maintain different types of databases.

Summary of Contents

The *IMS/ESA Administration Guide: Database Manager* contains 16 chapters and appendixes as follows:

"Appendix A. Meaning of Bits in the Delete Byte" on page 407, describes the meaning of bits in the delete and prefix descriptor bytes.

"Appendix B. Replace, Insert, and Delete Rules for Logical Relationships" on page 409, describes how to specify rules in a physical DBD and a rules summary.

"Appendix C. Using OSAM as the Access Method" on page 447, describes how to use overflow sequential access method (OSAM) as an access method.

"Appendix D. Correcting Bad Pointers" on page 449, describes how to use reorganization to correct bad pointers.

Summary of Library Changes for V5 describes the IMS library changes for Version 5.

Appendixes are followed by a bibliography and an index.

Prerequisite Knowledge

Before using this book, you should understand basic IMS/ESA concepts and your installation's IMS/ESA system. IMS/ESA 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/ESA concepts explained here pertain only to administering the IMS/ESA database. You should also know the COBOL, PL/I, or assembler language, and how to use DL/I calls.

For an introduction to IMS, see IMS/ESA General Information on the web at:

`http://www.software.ibm.com/data/ims`

IMS/ESA Application Programming: Design Guide describes how to design and code an application program.

For definitions of terms used in this manual and references to related information in other IMS manuals, see *IMS/ESA Master Index and Glossary*.

Change Indicators

Technical changes are indicated in this publication by a vertical bar (|) to the left of the changed text. If a figure has changed, a vertical bar appears to the left of the figure caption.

Summary of Changes

Changes to the Current Edition of this Book

This edition, which is in softcopy only, includes technical and editorial changes.

Changes to This Book for V6

This book contains new and changed information about the following subjects:

- Fast Path Online Change
- DEDB Online Change
- Database Image Copy 2 Support
- OSAM Database Coupling Facility Caching
- Shared SDEPs
- Shared VSO

It also contains additional information on performance and HSSP considerations.

Changes have been made to the following chapters:

Chapter 1	Introduction
Chapter 6	Database Design Considerations for Full Function
Chapter 7	Designing a Fast Path Database
Chapter 9	Developing Your Test Database
Chapter 11	Implementing Your Database Design
Chapter 13	Monitoring Your Database

This edition incorporates new technical information for Version 6 as well as editorial changes and technical corrections made to previously published information.

Library Changes for Version 6

The IMS/ESA Version 6 library differs from the IMS/ESA Version 5 library in these major respects:

- *IMS/ESA Common Queue Server Guide and Reference*

  This new book describes the IMS Common Queue Server (CQS).
- *IMS/ESA DBRC Guide and Reference*

  This new book describes all the functions of IMS Database Recovery Control (DBRC).
- The IMS Application Programming summary books (*IMS/ESA Application Programming: Database Manager Summary*, *IMS/ESA Application Programming: Transaction Manager Summary*, and *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS Summary*) are no longer included with the IMS library.
- The Softcopy Master Index is not included.
- All information about IRLM 1.5 and data sharing using IRLM 1.5 has been removed from the IMS V6 books. If you use IRLM 1.5, and want to migrate to using IRLM 2.1 and Sysplex data sharing, see *IMS/ESA Release Planning Guide*.

- The chapter that was titled ″Database Control (DBCTL) Interface″ in the ″*IMS/ESA″ Customization Guide* has been revised for Open Database Access (ODBA) and moved to ″Appendix A, Using the Database Resource Adapter (DRA)″ in the ″*IMS/ESA″ Application Programming: Database Manager*.

Chapter 1. Introduction

About This Chapter

This chapter describes the tasks of database administration and discusses the key concepts and terms used in this book.

Database Administration Overview

The task of database administration is to design, implement, and maintain a database. The *IMS/ESA Administration Guide: Database Manager* describes the tasks involved in administering the Information Management Systems/Enterprise Systems Architecture (IMS/ESA) database manager. IMS/ESA (hereafter referred to as IMS) is composed of two parts: a database manager and a transaction manager. The database manager manages physical storage of records in the database. The transaction manager manages the terminal network, the input and output of messages, and online system resources. The administration of the IMS/ESA transaction manager is covered in the *IMS/ESA Administration Guide: Transaction Manager*.

Whenever possible, this book presents the various 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. Other tasks, however, are ongoing, and you do not perform them in any special sequence. It is important for you to grasp not only what the tasks are (see “Database Administration Tasks” on page 2), but also how they interrelate (see Figure 1 on page 4) in the overall process of developing a database system.

This book addresses each major task in a separate chapter. This chapter provides the following information:

- Database administration tasks
- Concepts and technology
- Optional functions
- How to define your database to IMS
- How application programs view the database

The database administrator should consider the advantages of using command level Data Language I (DL/I). For detailed information, see *IMS/ESA Application Programming: Database Manager* and *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*.

Whenever tasks differ for Customer Information Control System (CICS) users, a brief description about the differences is included.

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

DBCTL supports non-message-driven batch message processors (BMPs).

DBCTL has its own log and participates in database recovery. Locking is provided by IMS program isolation or Internal Resource Lock Manager (IRLM).

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

Database Administration Tasks

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 utilizes 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 the database's characteristics and how application programs will use it to IMS. 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 have the application control blocks (ACBs) of the database prebuilt or built 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. The routine monitoring you do 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. Examples of these changes include database organization, database hierarchies (or the segments and fields within them), and addition (or deletion) of IMS functions. 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/ESA Operations Guide*.

You can use Database Recovery Control (DBRC) in recovering your databases. If your databases are registered in RECON, 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

You must ensure that all database recoveries use the IMS/ESA Version 6 utilities, as opposed to Version 4 or Version 5.

Related Reading: For more information on using these utilities, see the *IMS/ESA Utilities Reference: System*, the *IMS/ESA Utilities Reference: Database Manager*, and the *IMS/ESA Utilities Reference: Transaction Manager*.

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.

Figure 1 depicts tasks in the database development process and shows how the chapters of *IMS/ESA Administration Guide: Database Manager* fit into this process.

Figure 1. Tasks in the Database Development Process

Concepts and Terminology

This section discusses the terms and concepts you need to understand to perform the administration tasks just outlined.

You must know the following to understand this section:

- What a database is and why you store your data in it rather than in several files (explained in IMS/ESA General Information on the web at:

  `http://www.software.ibm.com/data/ims`
- 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/ESA Application Programming: Database Manager*).

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*.

Figure 2 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

The segments within a database record exist in a *hierarchy*. A hierarchy is the order

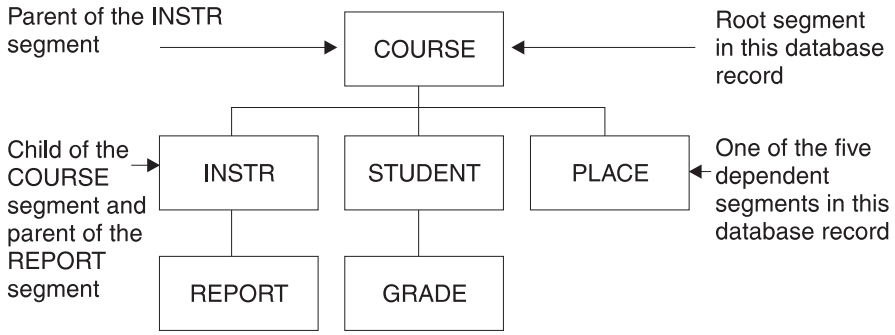


Figure 2. A Database Record in a School Database

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

The COURSE segment is called the *root segment*. Only one root segment exists within a database record. All other segments in the database record (such as: INSTR, REPORT, STUDENT, GRADE, and PLACE) are called *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.

One other thing to note about dependency in the database record, is that 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. 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 used to describe segments so far (root, dependent, parent, and child) describe the *relationship* between segments. The terms *segment type* and *segment occurrence*, however, distinguish between a type of segment in the database (the COURSE segment or the INSTR segment) and a specific segment (the course segment for a math course). The database record looked at so far is really the design you might come up with for a database record. The database record shows the segment types that are going to be in the database. Figure 3 shows an actual database record, based on this design.

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 Figure 3, 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, they do not have the same parent.

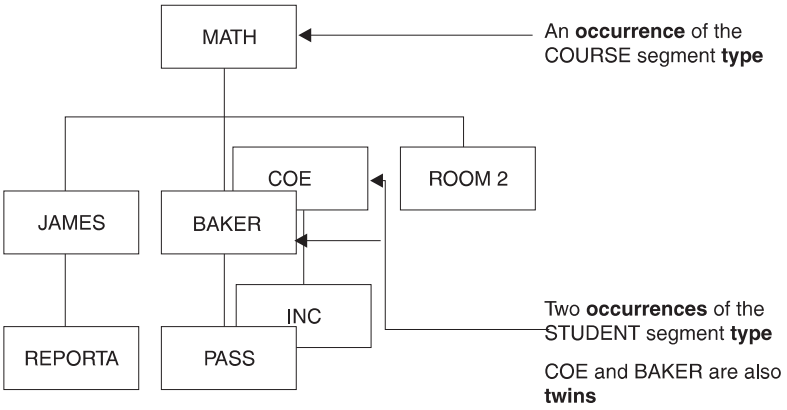


Figure 3. The School Database Record in Storage

The following section discusses the hierarchy in more detail. Subsequent sections 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

A database is composed of a series of database records, records contain segments, and 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 hierarchic 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 Figure 4.

The sequence goes from the top of the hierarchy to the bottom in the first (leftmost) *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*."

Figure 4 shows sequencing of segment types. Observe the segment occurrences and note the sequence in which segments are stored.

Figure 5 shows the same database record, but this time it is an actual record rather than an abstract design.

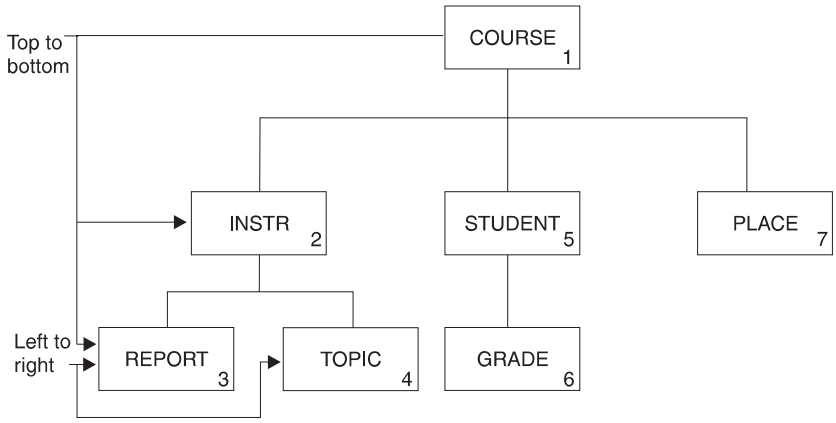


Figure 4. Sequence in a Hierarchy (Showing Segment Types Only)

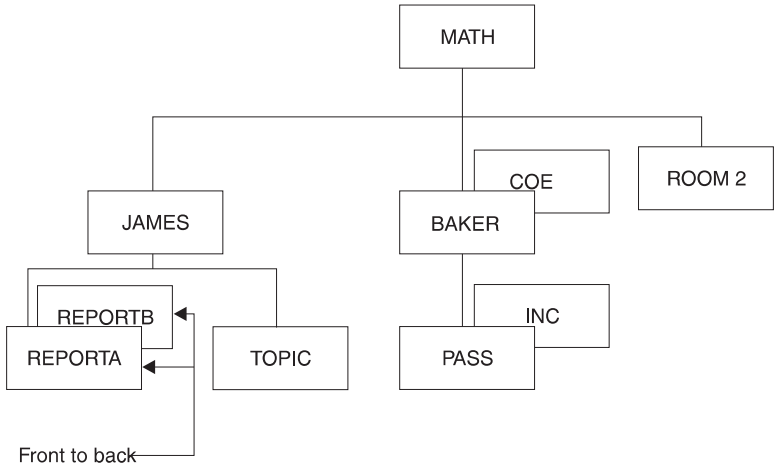


Figure 5. Sequence in a Hierarchy (Showing Segment Types and Occurrences)

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; so 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 hierarchic path until you reach the grade segment PASS. So, 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 hierarchic 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 hierarchic 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

Other terms that show the numbering sequence in a hierarchy are: *movement* and *position*. 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. The terms movement and position are used when talking about how segments are accessed when an application program issues a call.

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 5 on page 7), 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

A final term you need to know about hierarchies is: *level.* Level is the position of a segment in the hierarchy in relation to the root segment. The root segment is always on level one. Figure 6 on page 9 illustrates levels.

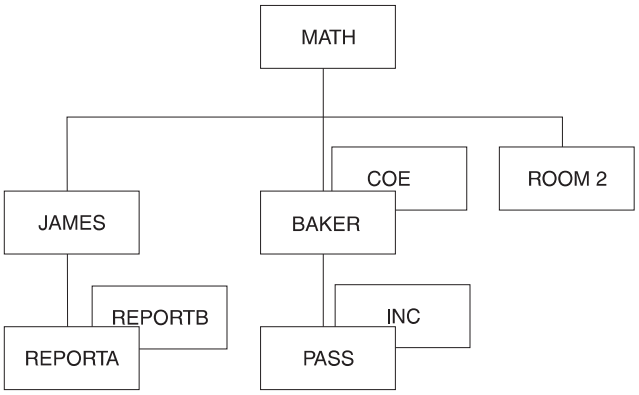


Figure 6. Levels in the Database

The Database

IMS allows you to define nine 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 MVS 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.

Table 1 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 the MVS Access Methods They Can Use

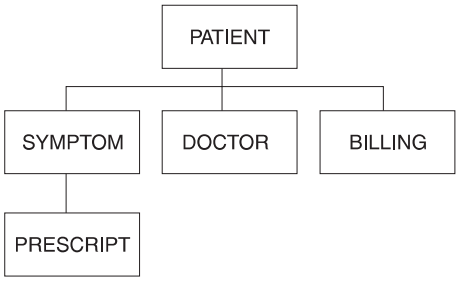
Type of Database	IMS Access Method Used	IMS or Operating System Access Methods that Can Be Used
HSAM	Hierarchical Sequential Access Method	BSAM or QSAM
SHSAM	Simple Hierarchical Sequential Access Method	BSAM or QSAM
HISAM	Hierarchical Indexed Sequential Access Method	VSAM
SHISAM	Simple Hierarchical Indexed Sequential Access Method	VSAM
GSAM[1]	Generalized Sequential Access Method	BSAM
HDAM	Hierarchical Direct Access Method	VSAM or OSAM
HIDAM	Hierarchical Indexed Direct Access Method	VSAM or OSAM
MSDB[2]	Main Storage Database	N/A
DEDB[1]	Data Entry Database	Media Manager

Notes:

1. Only available to BMPs through DBCTL.
2. Not applicable to DBCTL.

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 following medical database record, and put it in the school database.



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 Figure 7 can be stored.

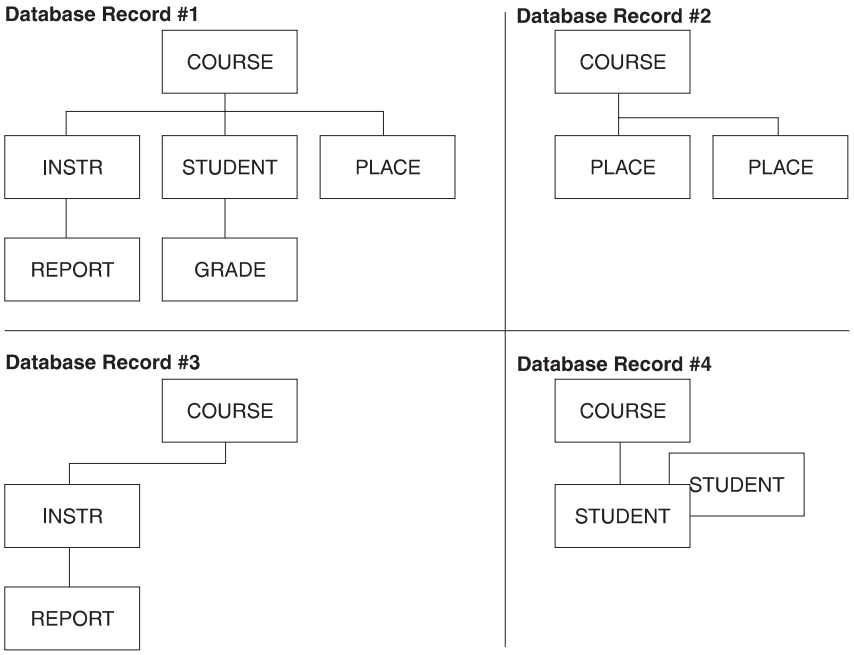
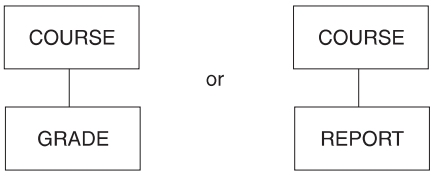


Figure 7. Example of Records That Can Be Stored in the School Database

However, no segment can be stored unless its parent is also stored. For example, you could *not* store:



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.

Figure 8 on page 12 shows the format of both a fixed-length and a variable-length segment. 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 for reasons explained later.

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.

For information on MSDB and DEDB segments, see “Main Storage Databases (MSDBs)” on page 195 and “Data Entry Databases (DEDBs)” on page 201.

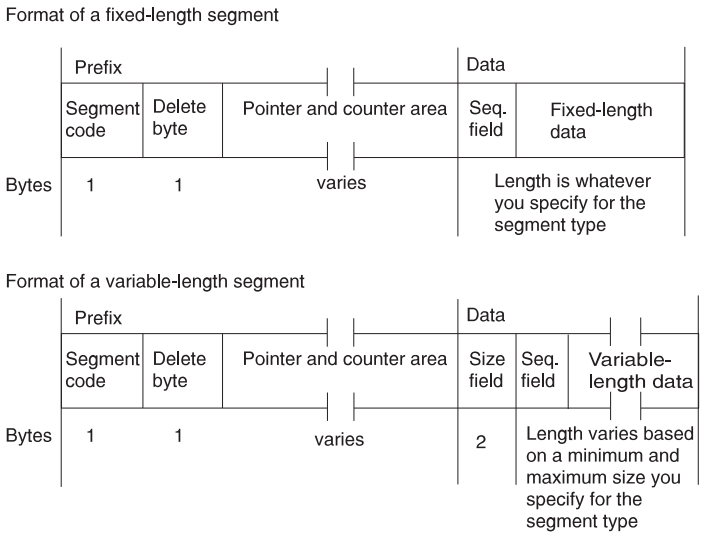


Figure 8. Formats of Segments

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 hierarchic 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.

For information on the meaning of each bit in the delete byte, see “Appendix A. Meaning of Bits in the Delete Byte” on page 407.

Pointer and Counter Area

The pointer and counter area exists in HDAM and HIDAM 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. (These addresses only exist in HDAM, HIDAM, and in some special cases, HISAM databases.)
- Counter information is used when logical relationships, an optional function of IMS, are defined. (Counter information can exist in HISAM, HDAM, and HIDAM databases.)

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 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.

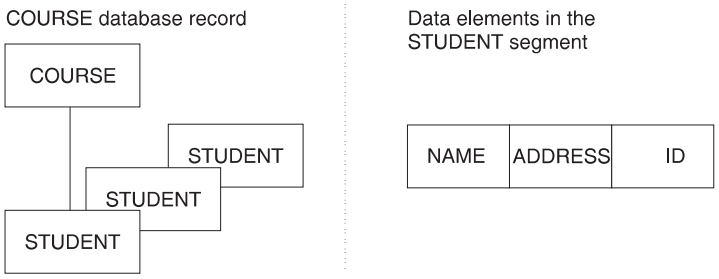
The application program accesses segments in a database using the name of the segment type. If an application program needs to reference part of a segment, a field name can be defined to IMS for that part of the segment. Field names are used in segment search arguments (SSAs) to qualify calls. An application 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.

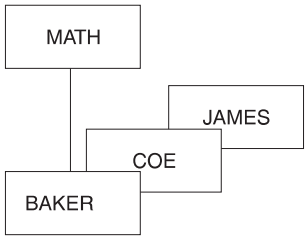
The Three Data Portion Field Types

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 following database record, there are three segment occurrences of the STUDENT segment, and the STUDENT segment has three data elements:



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 alphabetic sequence as follows:



When defined in a root segment of a HISAM, HDAM, or HIDAM database, a sequence field gives an application program access to a specific root segment. Because each database record has only one root segment, this means that an application program has access to a specific database record. When a sequence field is defined, a database does not need to be searched sequentially to find a

specific database record. Also, database records can be retrieved sequentially in HISAM and HIDAM databases when a sequence field is defined in the root segment.

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.

The next section of this chapter deals with the optional database function which you might need to use. Once the option functions are explored, the last two sections of this chapter briefly describe how to define a database to IMS, and how an application program views a database.

Optional Functions

IMS has several optional functions you can use for your database. These are discussed briefly below and described in detail in “Chapter 5. Choosing Additional Database Functions” on page 83. You need a cursory understanding of these functions before reading the rest of the book because they may be referred to before they are actually described.

The functions are as follows:

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. So 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 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 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 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 (so it does not need to 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 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 utilized.

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 databases. You can also use Data Capture exit routines to perform tasks other than data propagation.

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 databases asynchronously. You can also use Asynchronous Data Capture to perform tasks other than data propagation.

DPROPNR (DataPropagator NonRelational) allows you to propagate the changed data to or from IMS and DB2 both synchronously and asychronously.

Related Reading: for more information on DPROPNR see *Data Propagator NonRelational MVS/ESA An Introduction*.

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 hierarchic 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.

How To Define Your Database to IMS

Define the characteristics of your database to IMS by coding and generating a DBD (database description). A DBD is a series of macro instructions that describes a database’s organization and access method, the segments and fields in a database record, and the relationship between types of segments.

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 coding and generating a PSB (program specification block).

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.

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 PSB.

Chapter 2. Participating in Reviews

About This Chapter

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 sections of this chapter describe the reviews typically conducted during development of a database system. The types of reviews are:

- 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

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.

Your role 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 described in this book.

General Information About Reviews

The sections of this chapter describe reviews typically held during system development. (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 persons 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 first design review takes place after initial functional specifications for the system are complete. Its purpose is to ensure that all user requirements have been identified and that design assumptions are consistent with objectives. 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

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

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.

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

At this point in the design review process, you are designing hierarchies and starting to design the database. These tasks are described in “Chapter 3. Analyzing Data Requirements” on page 23, “Chapter 4. Designing a Fast Path Database” on page 33, “Chapter 5. Choosing Additional Database Functions” on page 83, and “Chapter 6. Database Design Considerations for Full Function” on page 165.

Design Review 4

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

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. These tasks

are described in “Chapter 3. Analyzing Data Requirements” on page 23, “Chapter 4. Designing a Fast Path Database” on page 33, and “Chapter 9. Developing Your Test Database” on page 265.

Code Inspection 1

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.

The objective of this review 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. 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 the database administration tasks defined in “Chapter 9. Developing Your Test Database” on page 265, “Chapter 11. Implementing Your Database Design” on page 277, and “Chapter 12. Loading Your Database” on page 285.

Code Inspection 2

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

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 (described in “Chapter 13. Monitoring Your Database” on page 309) to make sure your database still meets the performance expectations you have established.

Security Inspection

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

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 defined in “Chapter 16. Establishing Security” on page 403.

Post-Implementation Review

It is highly recommended that you conduct a post-implementation review. The 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.

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.

Post-Implementation Review

Chapter 3. Analyzing Data Requirements

About This Chapter

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 chapter, you need to be familiar with the terminology and examples explained in the introductory chapter on application design in *IMS/ESA Application Programming: Design Guide*. That chapter of the design guide explains how to develop local views for the business processes in an application.

Included in this chapter are the following topics:

Local View
: Introduces you to the local view examples and explains the information that makes up a local view.

Designing a Conceptual Data Structure
Explains how you can develop a conceptual data structure based on the local views for the business processes in an application.

Implementing the Structure with data language 1 (DL/I)
Explains how you implement the structure you have developed with DL/I. The considerations explained are: assigning data elements to segments and resolving data conflicts with DL/I.

Local View

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 chapter 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 chapter are for the following business processes in an education application:

Current Roster

Schedule of Classes

Instructor Skills Report

Instructor Schedules

The following information summarizes the local views developed in the introductory chapter on application design in *IMS/ESA Application Programming: Design Guide*.

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

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

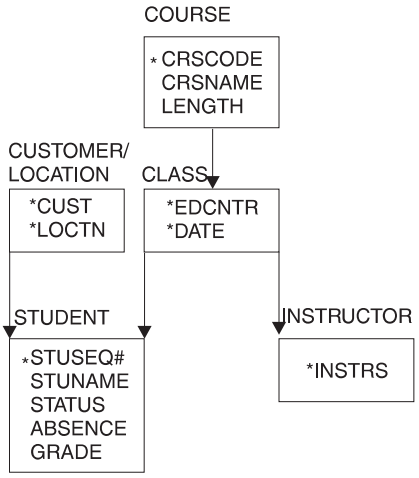


Figure 9. Current Roster Conceptual Data Structure

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

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

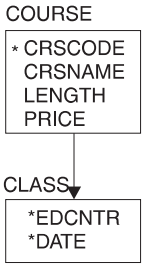


Figure 10. Schedule of Classes Conceptual Data Structure

Schedule of Classes Mappings

The only mapping for this local view is:

Course ◄——►► Class

Local View 3. Instructor Skills Report

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

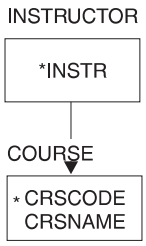


Figure 11. Instructor Skills Report Conceptual Data Structure

Instructor Skills Report Mappings

The only mapping for this local view is:

Instructor ◄——►► Course

Local View 4. Instructor Schedules

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

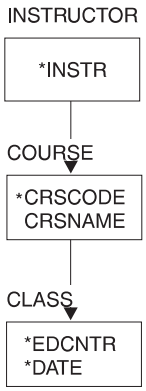


Figure 12. Instructor Schedules Conceptual Data Structure

Instructor Schedules Mappings

The mappings for this local view are:

Instructor ◄——►► Course

Course ◄——►► Class

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 (Figure 9 on page 25), 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 know as, carrying duplicate data).

Related Reading: "Resolving Data Conflicts" on page 30 explains the kinds of data conflicts that secondary indexes and logical relationships can resolve.

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 Table 2). 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 2. 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 Figure 13.

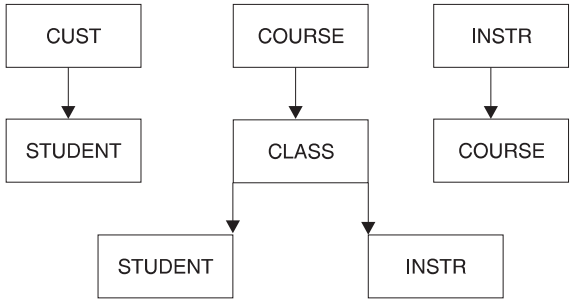


Figure 13. Education Data Structures

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. “Analyzing Requirements for Logical Relationships” on page 30 explains how.

Implementing the 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 have developed should be grouped into segments. Also, consider how DL/I can solve any existing data conflicts in the structure. The following 2 sections of this chapter explains how you assign data elements to segments, and how DL/I can resolve data conflicts.

Assigning Data Elements to Segments

Once 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 figure. The key and its associated data elements become the segment content.

Data Aggregate	Key	Data Element
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.

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 hierarchic 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 Figure 13 on page 29, 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 Figure 14 on page 31. The CUST/STUDENT hierarchy would be a logical structure.

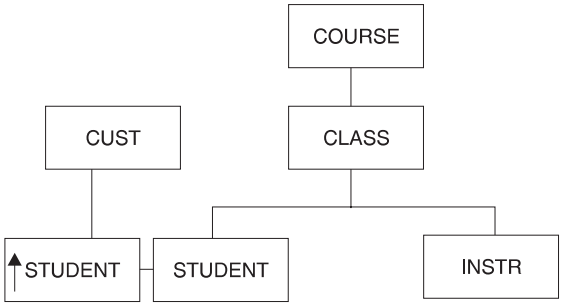


Figure 14. Education Hierarchies

This kind of logical relationship is called unidirectional, because the relationship is "one way."

The other conflict you can see in Figure 13 on page 29, 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 Figure 15, eliminating the need to carry duplicate data.

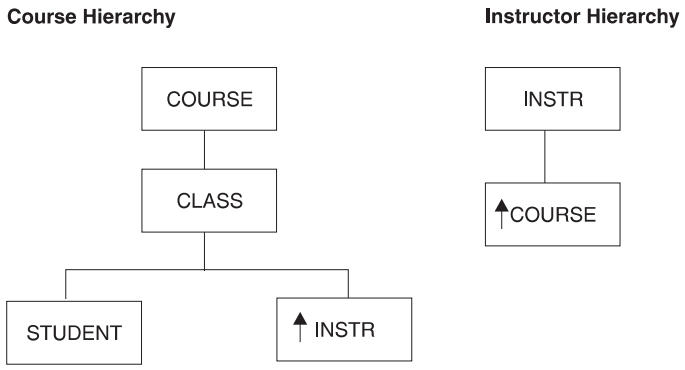


Figure 15. Bidirectional Logical Relationships

Chapter 4. Designing a Fast Path Database

Choosing a Database Type

IMS allows you to define nine different database types. Each type has different organization processing characteristics. Except for DEDB and MSDB, all the database types are discussed in this chapter. For information on DEDBs and MSDBs, see “Data Entry Databases (DEDBs)” on page 201 and “Main Storage Databases (MSDBs)” on page 195.

Choosing a Database Type

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

Each database type has its own access method. The following figure lists each type and the access method it uses:

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 [1]	Generalized Sequential Access Method
HDAM	Hierarchical Direct Access Method
HIDAM	Hierarchical Indexed Direct Access Method
DEDB	Data Entry Database
MSDB	Main Storage Database

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

Sequential Storage Method

The first four databases in the list use the sequential method of accessing data. With this method, the hierarchic 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 for reasons you will see later, no concept of hierarchy, database record, or segment exists in GSAM databases.

Direct Storage Method

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

For quick reference, see Table 9 on page 213 for a summary of HSAM, HISAM, HDAM, HIDAM, DEDB, and MSDB database characteristics.

Performance Considerations Overview

All databases are not created equal. You will want to make an informed decision regarding the type of database organizations which will best serve your purposes. The following figure briefly summarizes performance characteristics of the various database types; highlighting efficiencies and deficiencies of hierarchic sequential, hierarchic direct and general sequential databases.

For information on DEDBs and MSDBs, see "Data Entry Databases (DEDBs)" on page 201 and "Main Storage Databases (MSDBs)" on page 195.

1. Not applicable to CICS DBCTL

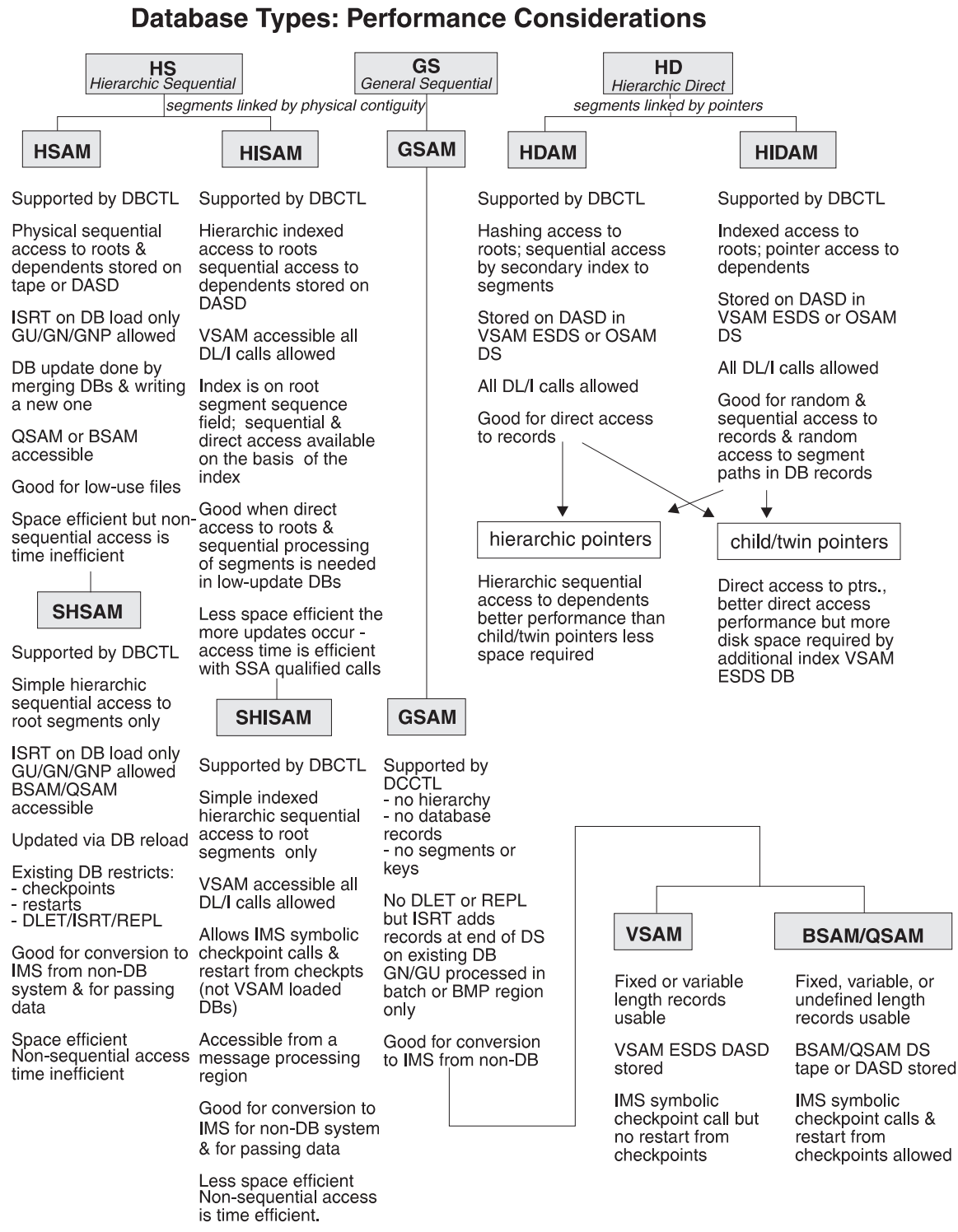


Figure 16. Performance Considerations for Different Database Types

IMS Databases

IMS databases are hierarchic databases that are accessed through Data Language I (DL/I calls). IMS makes it possible for application programs to retrieve, replace, delete, and add segments to IMS databases. 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).

Choosing a Database Type

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 Reading: For more information on converting a batch job to a BMP, see *IMS/ESA Application Programming: Design Guide* and *IMS/ESA Administration Guide: System*.

Databases Supported with DBCTL

Database Control (DBCTL) supports all IMS full-function databases:

HSAM
HISAM
SHSAM
SHISAM
HDAM
HIDAM

GSAM databases can be accessed only in IMS BMP regions and are not available to transaction-managers in a DBCTL environment. 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.

Databases Supported with DCCTL

Data communications control (DCCTL) supports GSAM databases in BMP regions and DB2 databases through the External Subsystem Attach Facility. DCCTL does *not* support full-function databases.

Related Reading: Information on ESAF is contained in *IMS/ESA Operations Guide*.

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 batch 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 hierarchic 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 facility
- 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

Although the uses of HSAM are limited because of its processing characteristics, it is used for applications requiring sequential processing only. 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 hierarchic 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. Figure 17 on page 38 shows one HSAM database record and how it appears when stored on tape.

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 (see Figure 17 on page 38) 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 hierarchic 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.

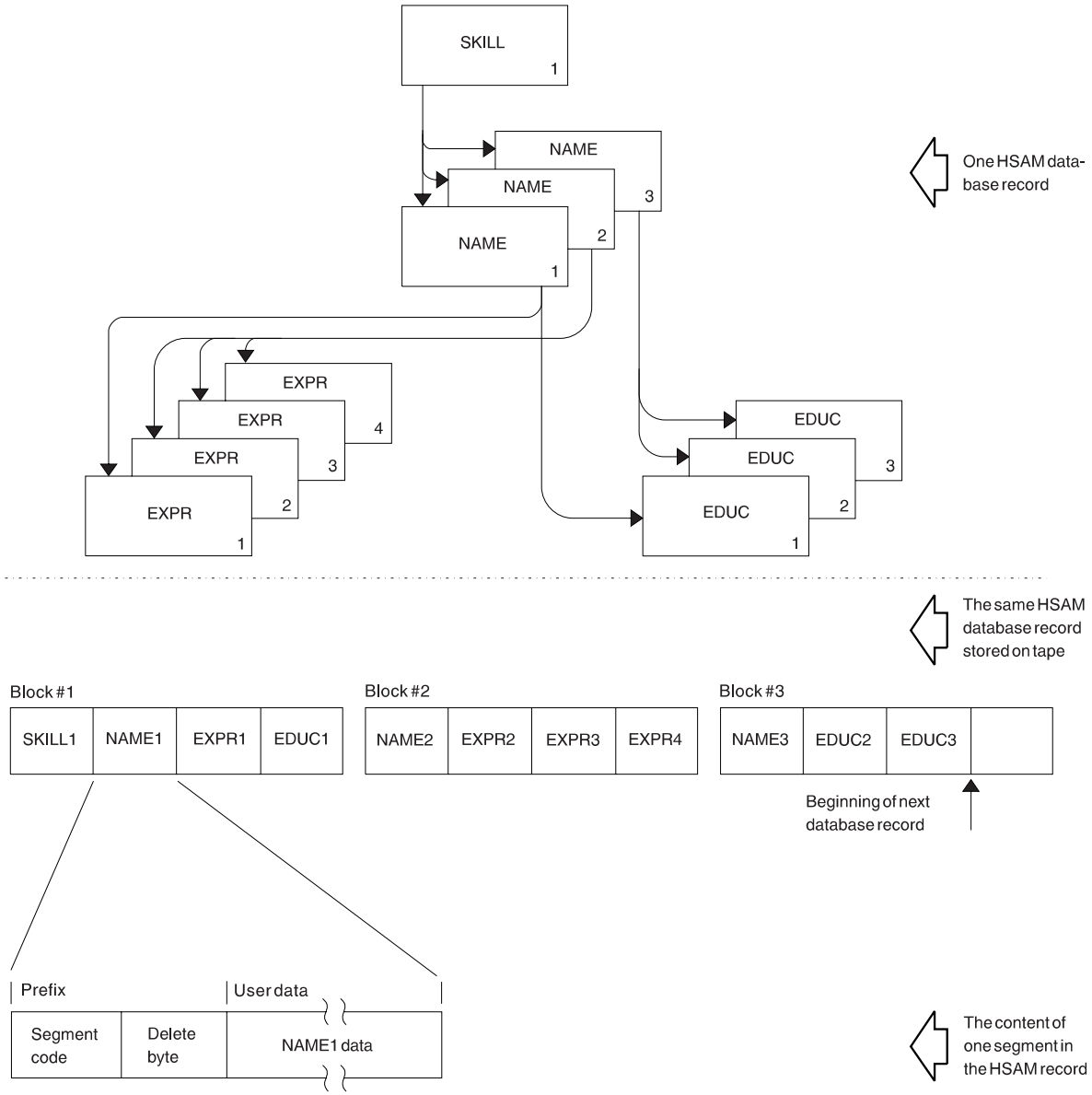


Figure 17. HSAM Database Record Stored on Tape

Once 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 (see Figure 18).

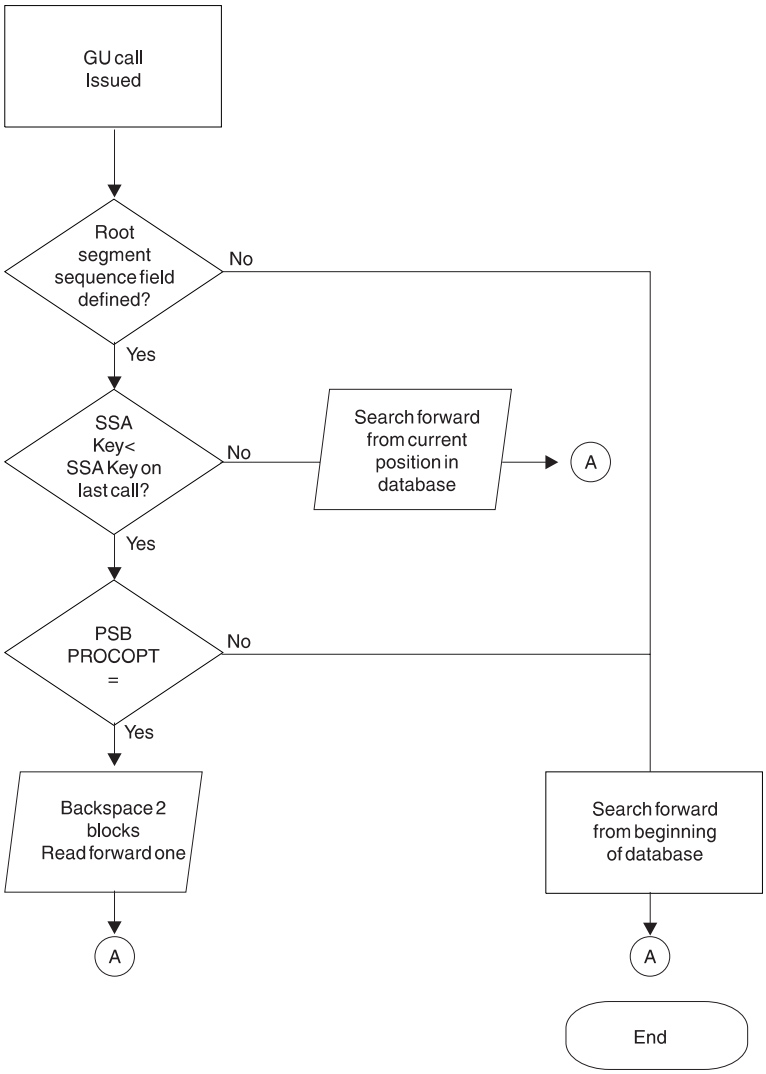


Figure 18. GU Calls against an HSAM Database

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 PROCOPY=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 desired 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

hierarchic 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. Figure 19 illustrates this process.

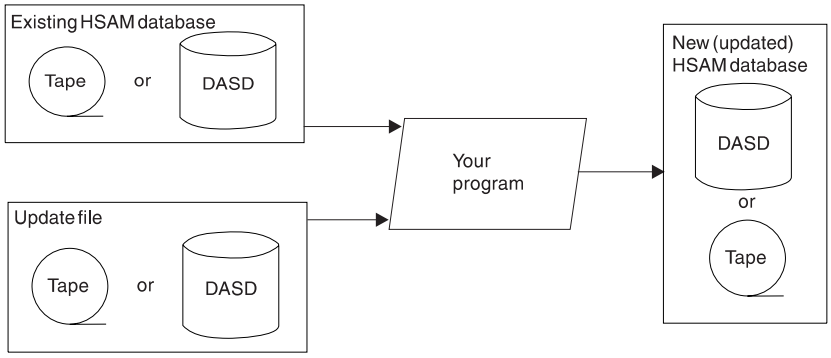


Figure 19. Updating an HSAM Database

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 facility
- Data Capture exit routines
- Field-level sensitivity
- Logging, recovery, and reorganization

Except for logging and recovery, each of these options is discussed in detail in later parts of this book. Information on logging and recovery is contained in *IMS/ESA DBRC Guide and Reference* and *IMS/ESA DBRC Guide and Reference*.

When to Use HISAM

HISAM is typically used for databases that require direct access to database records and sequential processing of segments in a database record. It is a good candidate for databases with the following characteristics:

- Most database records are about the same size.
- The database does not consist of relatively few root segments and a large number of dependent segments.
- Applications do not depend on a heavy volume of root segments being inserted after the database is initially loaded.
- Deletion of database records is minimal.

More detailed information on the uses of HISAM, requiring a working knowledge of how a HISAM database is organized and processed, is under “Using Variable-Length Segments” on page 140.

How a HISAM Record is Stored

HISAM database records are stored in two data sets. The first data set, called 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 second data set, called 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 later in this chapter). 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.

Figure 20 on page 42 shows four HISAM database records as they are initially stored on the primary and overflow data sets.

In storage, a HISAM segment (see Figure 20) 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 hierarchic sequence. The second byte of the prefix is the delete byte.

Each logical record in the primary data set contains the root plus all dependents of the root (in hierarchic sequence) for which there is enough space. The remaining

segments of the database record are put in the overflow data set (again in hierarchic 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 (the case for the first and second database record in Figure 20), 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.

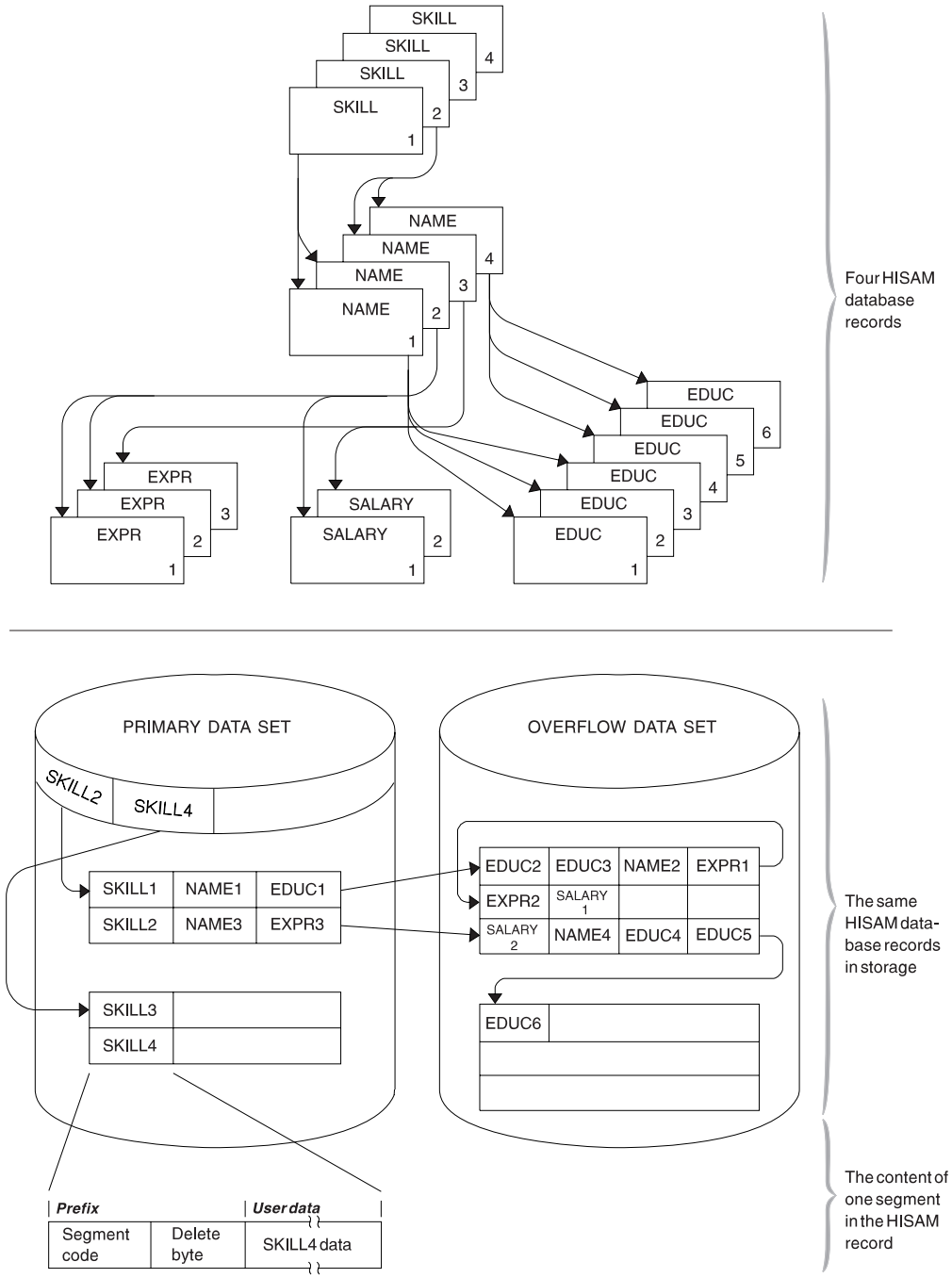


Figure 20. HISAM Database Records in Storage

Diagnosis, Modification or Tuning Information

Figure 21 shows the structure of a logical record in a HISAM database.

End of Diagnosis, Modification or Tuning Information

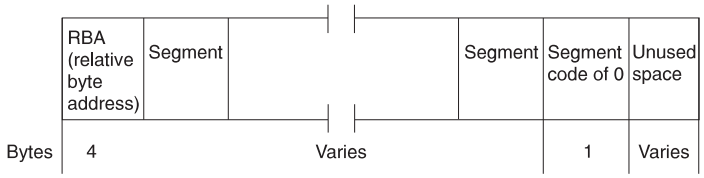


Figure 21. Format of a Logical Record in a HISAM Database

Diagnosis, Modification or Tuning Information

Logical Record

- 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 hierarchic 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.

End of Diagnosis, Modification or Tuning Information

Accessing Segments

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: Figure 22 on page 44 shows how insertion takes place when a free logical record exists. 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.

No Free Logical Record Exists: Figure 23 on page 45 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 for the DFSVSAMP data set. The OPTIONS statement is described in *IMS/ESA Installation Volume 2: System Definition and Tailoring*. See also “Choosing An Insert Strategy” in “Chapter 6. Database Design Considerations for Full Function” on page 165.

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 Figure 22, 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.

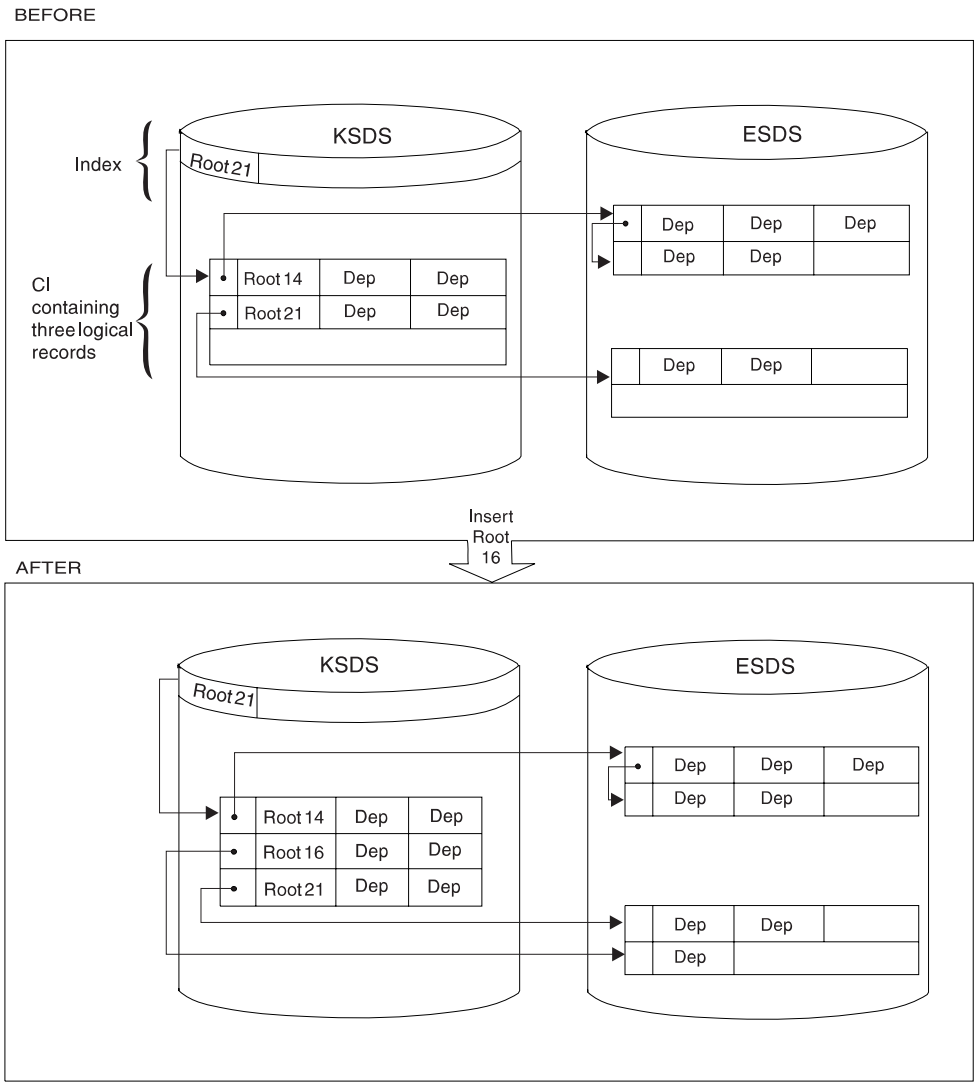


Figure 22. Inserting a Root Segment into a HISAM Database (Free Logical Record Exists in the Control Interval)

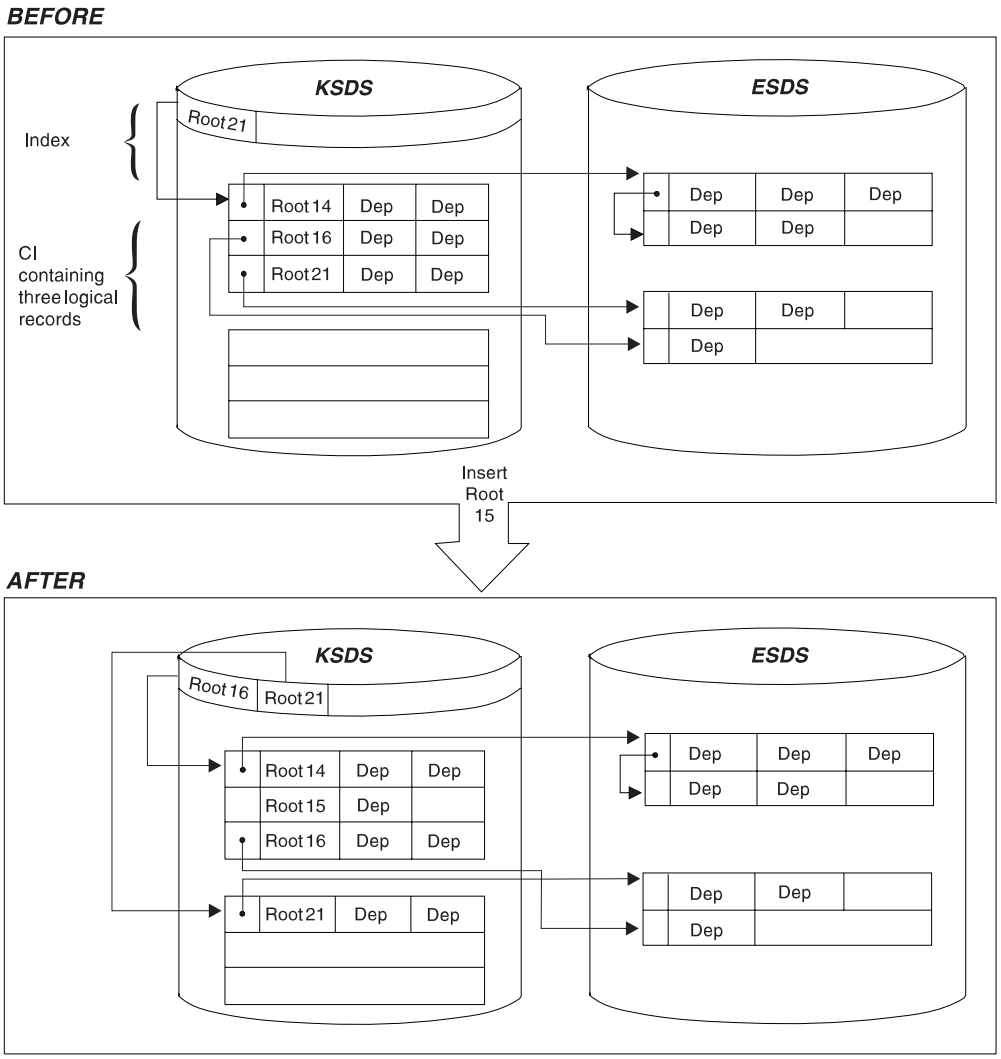


Figure 23. Inserting a Root Segment into a HISAM Database (No Free Logical Record Exists in the Control Interval)

Inserting Dependent Segments

Dependent segments inserted into a HISAM database after initial load are inserted in hierarchic 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.

Figure 24 on page 46 shows how segment insertion takes place when there is enough space in the logical record. The new dependent is stored in its proper hierarchic position in the logical record by shifting the segments that hierarchically follow it to the right in the logical record.

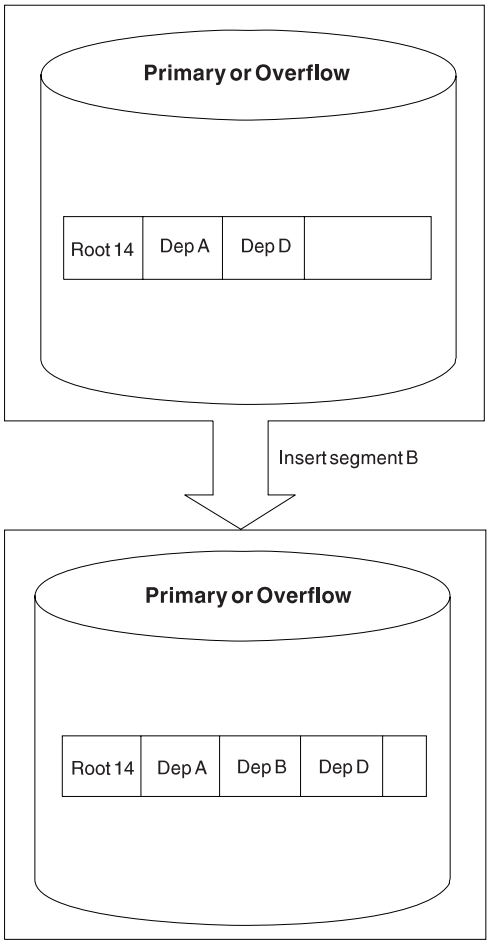


Figure 24. Inserting a Dependent Segment into a HISAM Database (Space Exists in the Logical Record)

Figure 25 on page 47 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 hierarchic 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.

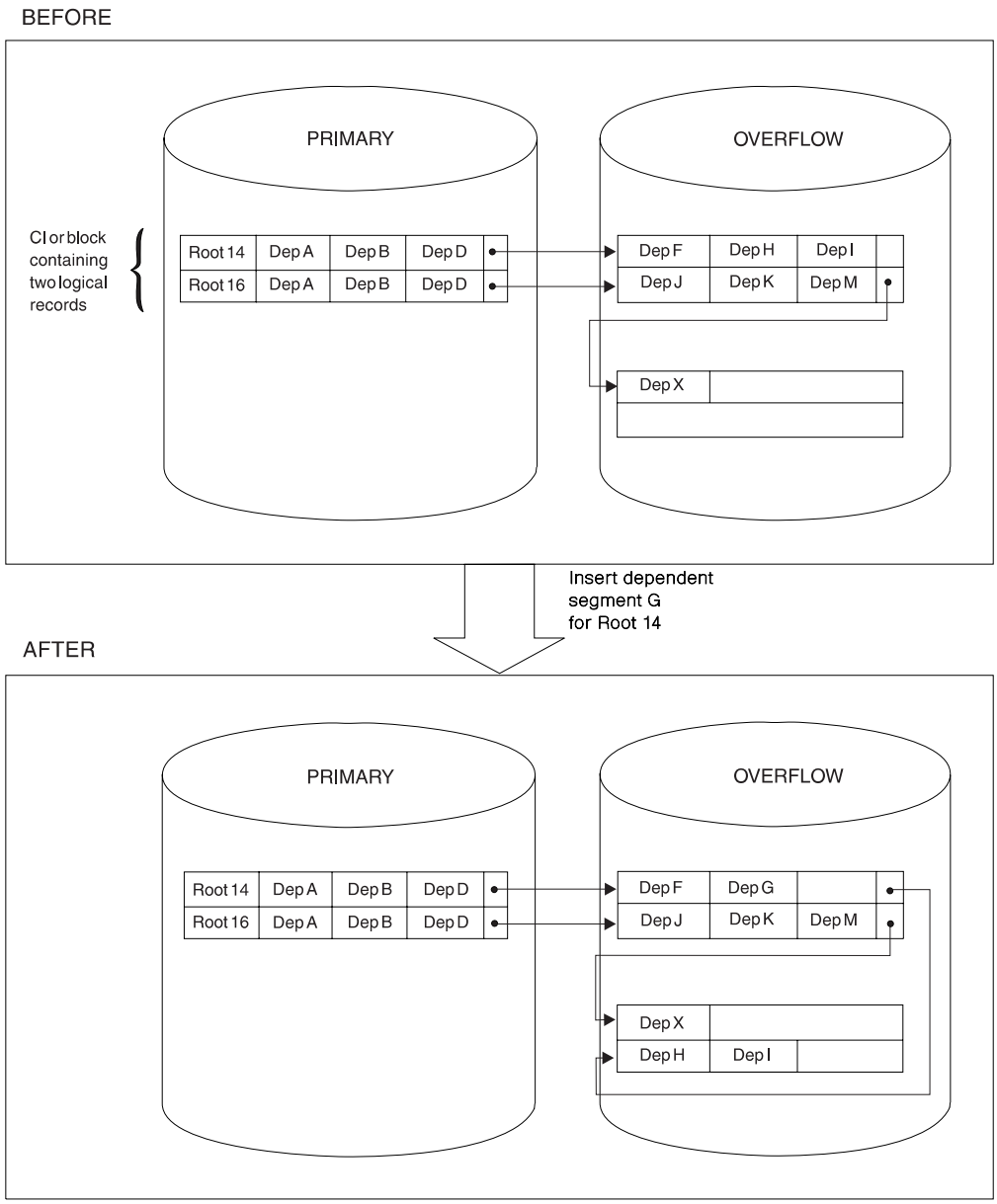


Figure 25. Inserting a Dependent Segment into a HISAM Database (No Space Exists in the Logical Record)

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. 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 Figure 26 and in Figure 27.

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.

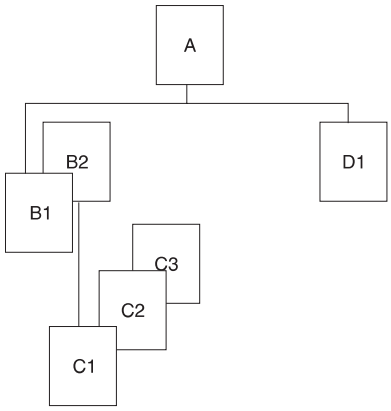


Figure 26. The Hierarchic Segment Layout on the Database

A request to access segment D1 is made. Although segments B2, C1, C2, and C3 cannot be accessed, they still exist in the database so they must still be "searched through" even though they are inaccessible.

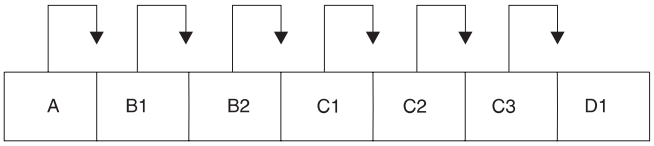


Figure 27. Accessing a HISAM segment that hierarchically follows deleted segments

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. Refer to the IMS System Definition Reference Guide for information on the erase parameter of the DBD statement.

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.

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.

The implications of replacing segments when variable-length segments are used is discussed under "Using Variable-Length Segments" on page 140.

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.

  This allows 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.

The implications of using HISAM with logical relationships or secondary indexes are discussed in the sections describing those functions later in the chapter.

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 in two situations.

Situation 1 - Converting from a non-database system to IMS

SHSAM, SHISAM, and GSAM databases allow existing programs, using MVS 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 MVS data sets.

Situation 2 - Passing data

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.

The following sections describe each of the three database types. Table 3 on page 52 is a chart comparing SHSAM, SHISAM, and GSAM.

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 MVS 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 section. Before deciding to use SHSAM, read the section 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 facility
- 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 via 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

In addition to those situations described in the introduction to this section, SHISAM is useful if you need an application program that accesses MVS data sets to use the IMS symbolic checkpoint call.

The IMS symbolic checkpoint call makes restart easier than the MVS basic checkpoint call. If the MVS 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 section 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 facility

GSAM Databases

GSAM databases are sequentially organized databases designed to be compatible with MVS data sets. GSAM databases can be on a data set previously created or one later accessed by the MVS 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. 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. GSAM is designed to be compatible with MVS data sets. The GSAM database has no hierarchy, database records, segments or keys.

GSAM IMS Symbolic Checkpoint Call

In addition to those situations described in the introduction to this section, GSAM is useful if you need an application program that accesses MVS data sets to use the IMS symbolic checkpoint call. The IMS symbolic checkpoint call makes restart easier than the MVS 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.

Because GSAM databases are supported in a DCCTL environment, you may 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 facility
- Field-level sensitivity
- Data Capture exit routines
- Logging or reorganization
- Multiple data set groups

If you have application programs that need access to both IMS and MVS data sets, you can use SHSAM, SHISAM, or GSAM. Which one you use depends on what functions you need. Table 3 compares the characteristics and functions available for each of the three types of databases.

Table 3. Comparison of SHSAM, SHISAM, and GSAM Databases

Characteristics and Functions	SHSAM	SHISAM	GSAM
Hierarchic structure applicable?	NO	NO	NO
Segment prefix exist?	NO	NO	NO
Variable-length records used?	NO	NO	YES
Checkpoint/restart possible?	NO	YES[1]	YES[1]
Compatible with non-IMS data sets?	YES	YES	YES
Can VSAM be used as the operating system access method?	NO	YES	YES
Can BSAM be used 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
Logging available?	NO	YES	NO
GET calls allowed?	YES	YES	YES
ISRT calls allowed?	YES[2]	YES	YES[3]
Supported for CICS-DBCTL?	YES	YES	NO
Supported for DCCTL?	NO	NO	YES

:

[1] Using symbolic checkpoints

[2] To load database only

[3] Allowed only at the end of the data set

HDAM and HIDAM Databases

Hierarchical direct access method (HDAM) and hierarchical indexed direct access method (HIDAM) databases have many similarities and are referred to as HD databases.

HD databases differ from sequentially organized databases in two important ways. First, they use the direct method of storing data, and the hierarchic sequence of segments in the database is maintained by having segments point to one another. Except for a few special cases, each segment 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. Their position, once stored, is fixed, and they do not “move around” in the database when subsequent processing takes place. Instead, pointers are updated to reflect processing changes.

HD databases also differ from sequentially organized ones in that 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 are stored on direct-access devices in either a VSAM ESDS or an OSAM data set. See “Appendix C. Using OSAM as the Access Method” on page 447 for information on OSAM data sets. The storage organization in HDAM and HIDAM is basically the same. Their primary difference is in the way their root segments are accessed. In HDAM, each root segment’s storage location is found using a randomizing module. The randomizing module examines the root’s key and, through an arithmetic technique, computes the address of a pointer to the root segment. In HIDAM, each root segment’s storage location is found by searching an index. This index, unlike the index used with HISAM, requires use of a *second database*, an index database. IMS loads and maintains this index database. The advantage of the HDAM randomizing module is that the I/O operations required to search an index are eliminated. In addition, no need exists to update an index after a root segment is inserted or deleted.

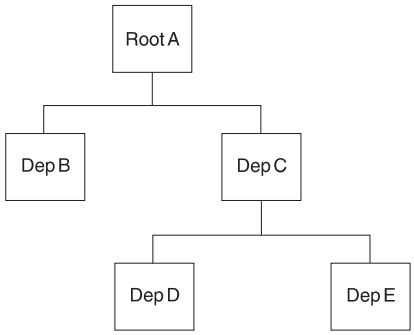
All DL/I calls can be issued against HD databases. In addition, the following options are available:

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

Except for logging and recovery, each of these options is discussed in detail in subsequent sections of this chapter. For information on logging and recovery, see *IMS/ESA Operations Guide* and *IMS/ESA Sample Operating Procedures*.

When to Use HDAM

HDAM databases are typically used when you need primarily direct access to database records. The randomizing module provides fast access to the root segment (and therefore the database record). HDAM databases also give you fast access to paths of segments as specified in the DBD in a database record. For example, in the following database record, if physical child pointers are used they can be followed to reach segments B, C, D, or E. A hierarchic 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.



You cannot process HDAM database records in key sequence unless the randomizing module you use stores root segments in physical key sequence. More information on HDAM is addressed later in this chapter.

When to Use HIDAM

HIDAM 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, because the HIDAM 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.

What You Need to Know About HD Databases

Before looking in detail at how HD databases are stored and processed, you need to become familiar with:

The various types of pointers you can specify for a HD database

The general format of the database

The use of special fields in the database

Types of Pointers You Can Specify: The hierarchic sequence of segments in a database record using the sequential access methods is maintained by keeping segments physically adjacent to each other in storage. In the HD access methods, segments in a database record are kept in hierarchic 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.

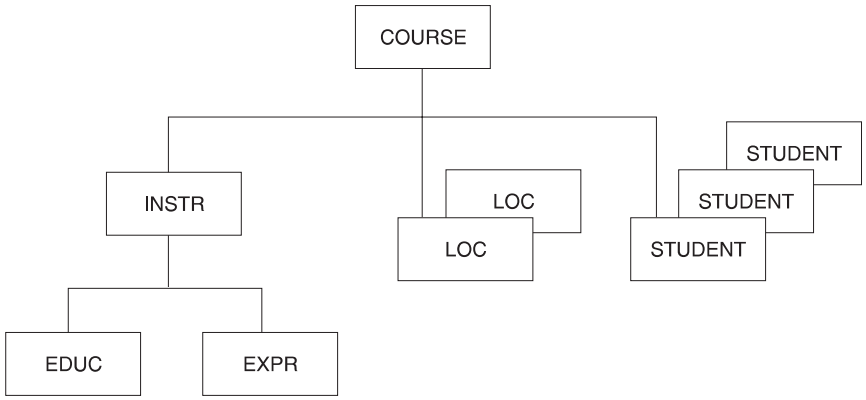
Several different types of direct-address pointers exist, and you will see how each works in the sections that follow. However, there are three basic types:

- Hierarchic pointers, which point from one segment to the next in either forward or forward and backward hierarchic 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 hierarchic sequence, use hierarchic 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 section. In the section called "Mixing Pointers," how pointers can be mixed is discussed. In the following sections, each type of pointer is illustrated, and the database record on which each illustration is based is as follows:



Hierarchic Forward Pointers: With hierarchic forward (HF) pointers, each segment in a database record points to the segment that follows it in the hierarchy. Figure 28 shows hierarchic forward pointers:

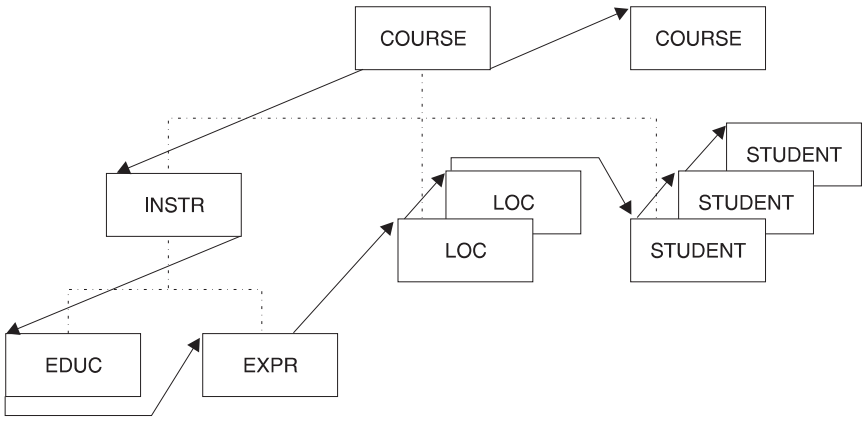


Figure 28. Hierarchic Forward Pointers

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 hierarchic sequence and processing does not require a significant number of delete operations. If there are a lot of delete operations, hierarchic 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. HF pointers are specified by coding PTR=H in the SEGM statement in the DBD.

Hierarchic Forward and Backward Pointers: With hierarchic 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. Figure 29 shows how HF and HB pointers work.

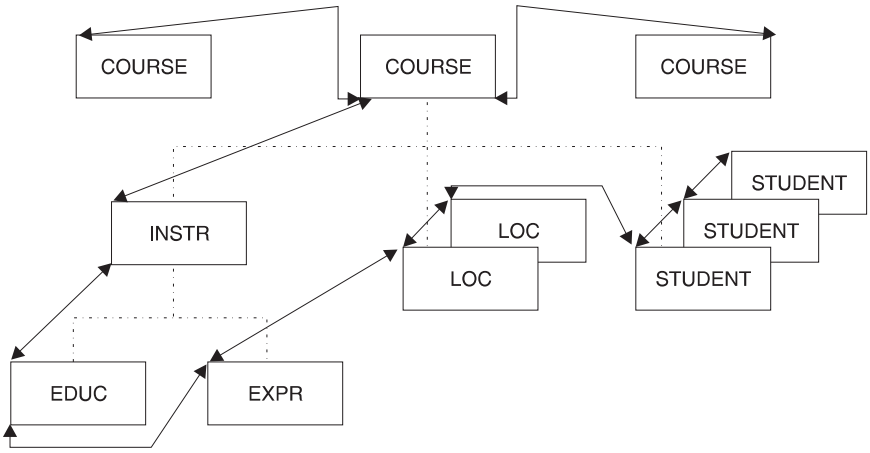


Figure 29. Hierarchic Forward and Backward Pointers

HF pointers work in the same way as the HF pointers described in the preceding section.

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

HF and HB pointers are specified by coding PTR=HB in the SEGM statement in the DBD.

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. Figure 30 shows PCF pointers:

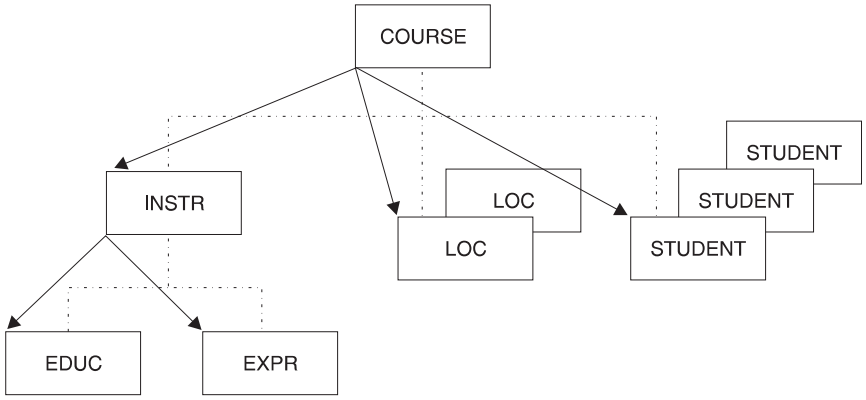


Figure 30. Physical Child First Pointers

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 (explained later) can be used to form this connection. PCF pointers should be used when segments in a database record are typically processed randomly and sequence fields are either defined for the segment type. If not, new segments are not inserted at the end of all existing segment occurrences. 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.

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. Figure 31 shows PCF and PCL pointers:

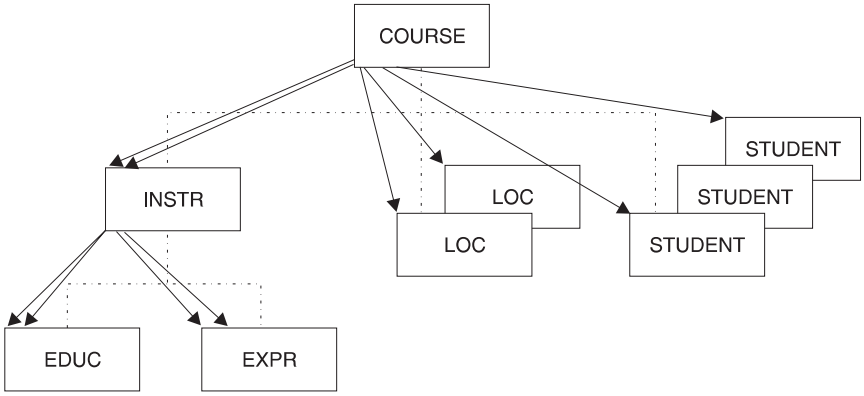


Figure 31. Physical Child First and Last Pointers

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 (explained later) 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.

A parent segment can have SNGL specified on one immediately dependent child segment type and DBLE specified on another. Figure 32 on page 59 shows specifying PCF and PCL pointers.

Coding these pointers in the DBD:

```
DBD
SEGM A
SEGM B  PARENT=((name.SNGL))   (specifies PCF pointer only)
SEGM C  PARENT=((name.DBLE))   (specified PCF and PCL pointers)
```

Results in these pointers being created:

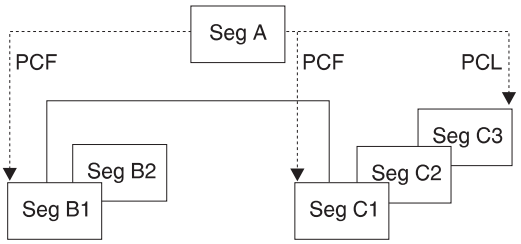


Figure 32. Specifying PCF and PCL Pointers

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. Figure 33 on page 60 illustrates this.

Note that PTF pointers can be specified for root segments. When this is done in an HDAM database, the root segment points to the next root in the database chained off the same root anchor points (RAP). (RAPs are explained in a following section called “General Format of HD Databases and Use of Special Fields.”) If no more root segments are chained from this RAP, the PTF pointer is zero.

When PTF pointers are specified for root segments in HIDAM database, the root segment does *not* point to the next root in the database. What happens is explained in a subsequent section called “Use of RAPs in a HIDAM Database.” The important thing for you to know now is that 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. This increases access time. This problem is eliminated if you specify PTF *and* physical twin backward (PTB) pointers (discussed next).

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. PTF pointers are specified by coding PTR=T in the SEGM statement in the DBD. This is the SEGM statement for the segment containing the pointer. The combination of PCF and PTF pointers is used as the default when pointers are not specified in the DBD. Figure 33 show PTF pointers:

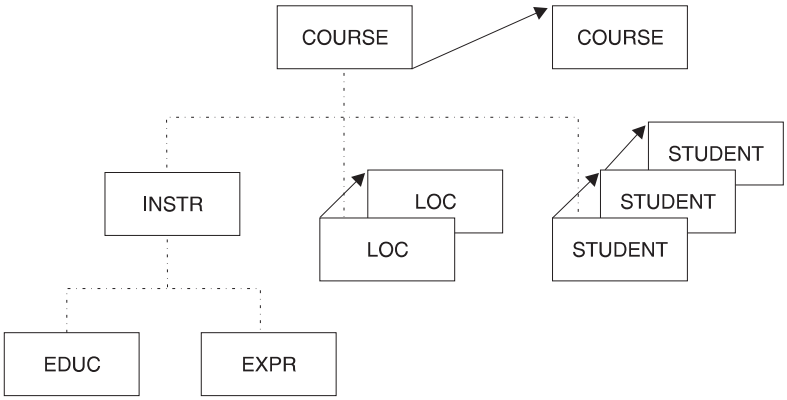


Figure 33. Physical Twin Forward Pointers

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. Figure 34 illustrates how PTF and PTB pointers work.

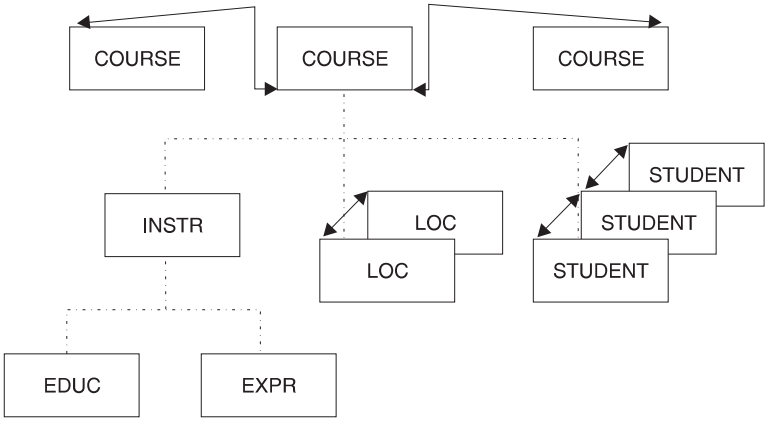


Figure 34. Physical Twin Forward and Backward Pointers

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 database when you need fast sequential processing of database records. By using PTB pointers in root segments, database records can be sequentially processed without intervening references to the HIDAM index. PTB pointers improve performance when deleting a segment in a twin chain accessed by a virtually paired logical relationship. This happens when the delete that causes DASD space to be released occurs on a delete from the logical access path.

Eight bytes are needed for the PTF and PTB pointers in each segment occurrence of a given segment type. PTF and PTB pointers are specified by coding PTR=TB in the SEGM statement in the DBD.

Mixing Pointers: Because pointers are specified by segment type, the various types of pointers can be mixed within a database record. However, only hierarchic 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	Hierarchic forward
HF and HB	Hierarchic 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

Figure 35 on page 62 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. Figure 35 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 hierarchic pointers must contain the same number of pointers (twin or hierarchic) 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).

Sequence of Pointers in a Segment's Prefix:

Diagnosis, Modification or Tuning Information

When a segment contains more than one type of pointer, pointers are put in the segment's prefix in the following sequence:

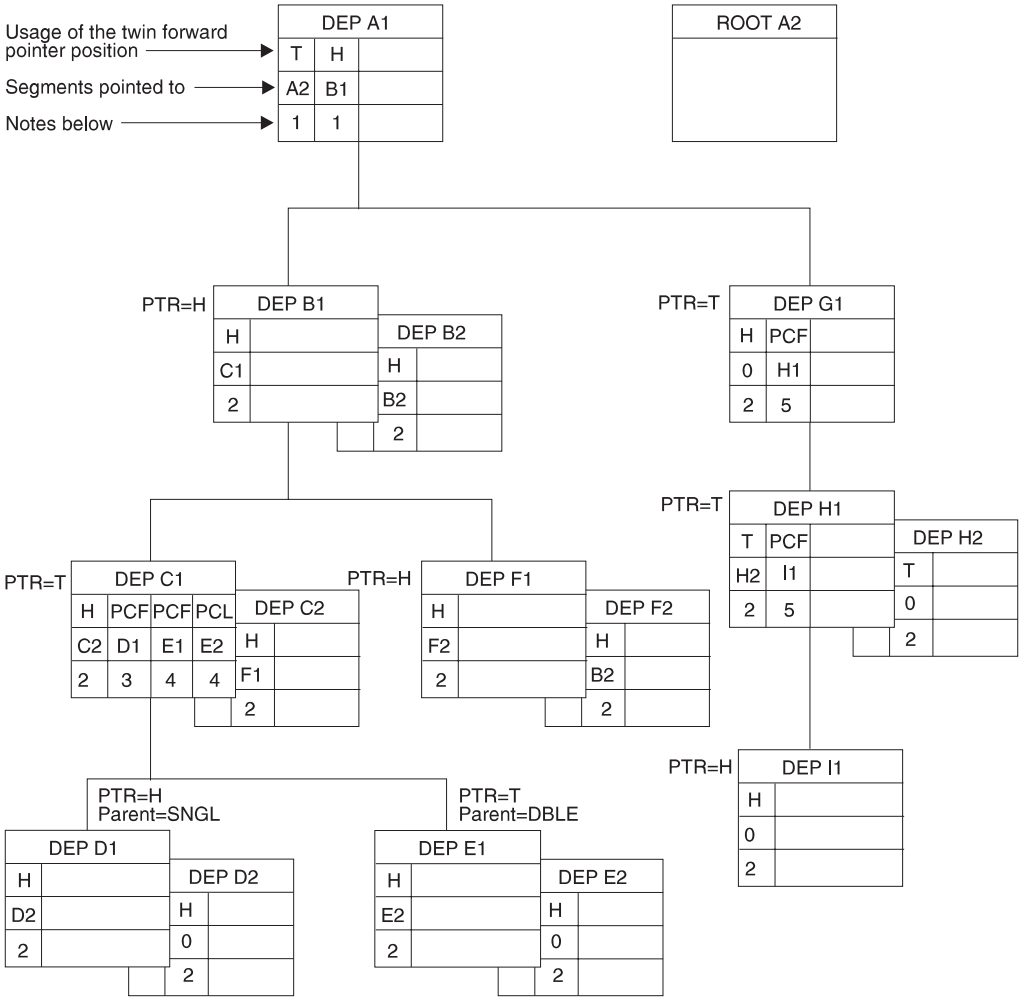
HF | HB or TF | TB | PCF | PCL

End of Diagnosis, Modification or Tuning Information

Diagnosis, Modification or Tuning Information

Figure 35 shows an example of mixing pointers in a database record.

End of Diagnosis, Modification or Tuning Information



Notes:

1 Caused by specifying PTR=H on the root segment
2 If PTR=H, usage is hierarchical (H); otherwise, usage is twin (T)
3 Caused by specifying PTR=T on segment type C and PARENT=SNGL on segment type D
4 Caused by specifying PTR=T on segment type C and PARENT=DBLE on segment type E
5 Caused by specifying PTR=T on this segment type

Figure 35. Mixing Pointers

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. This section looks at the general layout of the database special fields.

The databases referred to here are the HDAM and the HIDAM database. The HIDAM contains database records. HIDAM has an additional database, the index database, that is allocated by you but loaded and maintained by IMS. This section examines the index database when dealing with the storage of HIDAM records. Figure 36 shows the general format of an HD database and some of the special fields used in it.
HD databases use a single data set, that is either a VSAM ESDS or an OSAM data

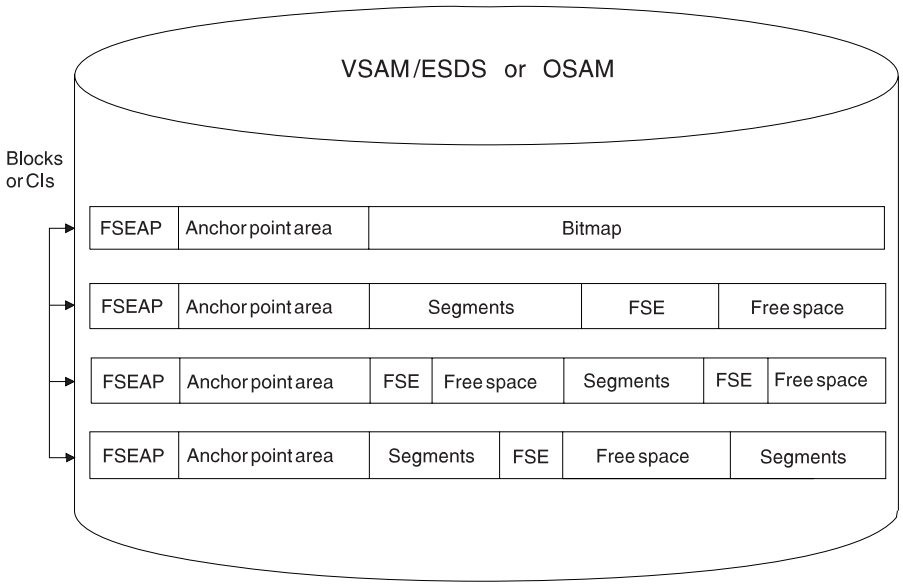


Figure 36. Format of an HD Database and Special Fields in It

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 have it done by the Database Description Generation (DBDGEN) utility. The utility 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 Figure 36 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.

Diagnosis, Modification or Tuning Information

Examine the four fields illustrated in Figure 36. 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:

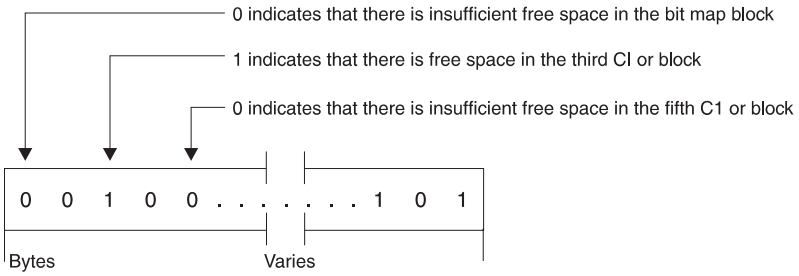
- This list item contains diagnosis, modification, or tuning information.

  Bit map. Bit maps 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 the bit map 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 bit map 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 bit map and the first CI is reserved. The first bit map 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 bit map, another bit map 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.

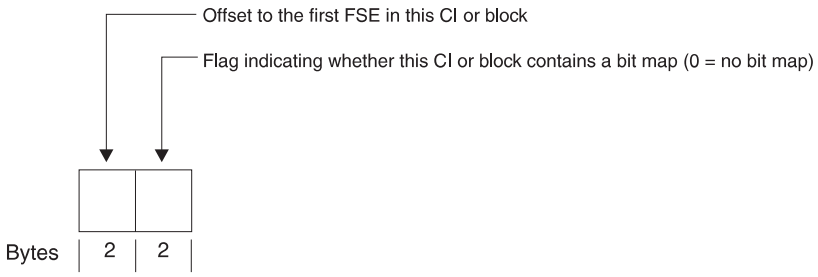
A bit map looks like this:



- This list item contains diagnosis, modification, or tuning information.

  Free space element anchor point (FSEAP). FSEAPs are made up of two 2-byte fields. The first 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 bit map. If the block or CI does not contain a bit map, 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 looks like this:



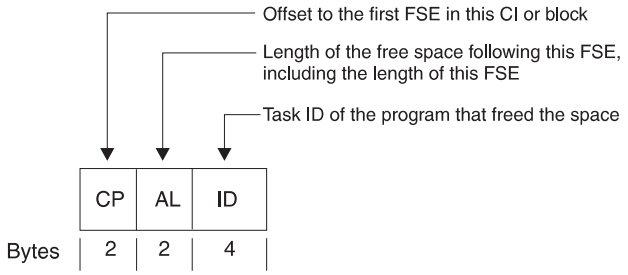
  The FSEAP in the first bit map block in an OSAM data set has a special use. It is used to contain the DBRC usage indicator for the database. The DBRC usage indicator is used at database open time for update processing to verify usage of the correct DBRC RECON data set.

- This list item contains diagnosis, modification, or tuning information.

  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.

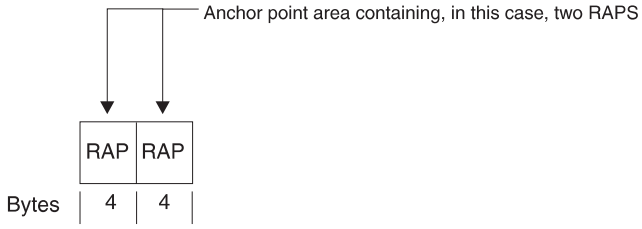
An FSE looks like this:



- This list item contains diagnosis, modification, or tuning information.

  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. In HDAM, you specify the number of RAPs you need on the RMNAME= parameter in the DBD statement. In HIDAM, RAPs only exist if PTR=T or PTR=H is specified for a root segment type. In addition, only one RAP per block or CI is generated. The way in which RAPs are used in HDAM and HIDAM differs, so RAPs will be examined further in the following sections describing how HDAM and HIDAM records are stored.

  An anchor point area in an HDAM database looks like this:



End of Diagnosis, Modification or Tuning Information

How HDAM Records Are Stored

HDAM 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 storage of dependent 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). You also specify the maximum number of bytes of a database record to be stored in the root addressable area.

You do this through a series of uninterrupted ISRT calls to another database record. (This is done in the BYTES operand on the RMNAME= parameter in the DBD statement).

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 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 hierarchic sequence following the root.

To store an HDAM database record, the user 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. It 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. 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. This is because the randomizing module always chains roots off a RAP 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.

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. This algorithm is explained in "How the HD Space Search Algorithm Works" on page 78. 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. (These ISRT rules are explained in *IMS/ESA Application Programming: Database Manager*.) All roots chained like this from a single anchor point area are called *synonyms*.

Figure 37 on page 69 shows two HDAM 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 randomizing module generated unique relative block and RAP numbers for each root. 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 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, hierarchic pointers are used instead of the combination of physical child/physical twin pointers. Each segment points to the next segment in hierarchic 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, Figure 37 does not show the various space management fields.

An HDAM segment in storage (see Figure 37) 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 hierarchic 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, hierarchic forward pointers are used. Therefore, the EXPR4 segment contains only one address, the address of the NAME3 segment.

How HIDAM Records Are Stored

A HIDAM database is actually composed of two databases. (HIDAM uses an index to get to a specific root segment rather than the root anchor points used by HDAM.) The first database contains the database records as the database. The second database contains the HIDAM index as the index database.

Loading a HIDAM Database: Root segments in a HIDAM 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 database, all root segments should be presented to the load program in ascending key sequence, and all dependents of a root should follow the root in hierarchic sequence. Figure 38 on page 70 shows two HIDAM database records and how they appear in storage after initial load. Note that HIDAM, unlike HDAM, has no root addressable or overflow area, just a series of blocks or CIs. 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 Figure 38 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.

Note how segments in a database record are chained together. In this case, hierarchic pointers were used instead of the combination of physical child/physical twin pointers. Each segment points to the next segment in hierarchic sequence. No RAPs exist in Figure 38. Although HIDAM databases can have RAPs, you probably do not need to use them. The reason for not using RAPs is explained in the next section.

In storage, a HIDAM segment (see Figure 38) 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 hierarchic 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,

hierarchic 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.

Creating an Index Segment: As each root is stored in a HIDAM database, IMS creates an index segment for the root and stores it in the index database. The index database consists of a single VSAM KSDS. The KSDS contains an index segment for each root in the database. When initially loading a HIDAM database, IMS will insert a root segment with a key of all X'FF's as the last root in the database.

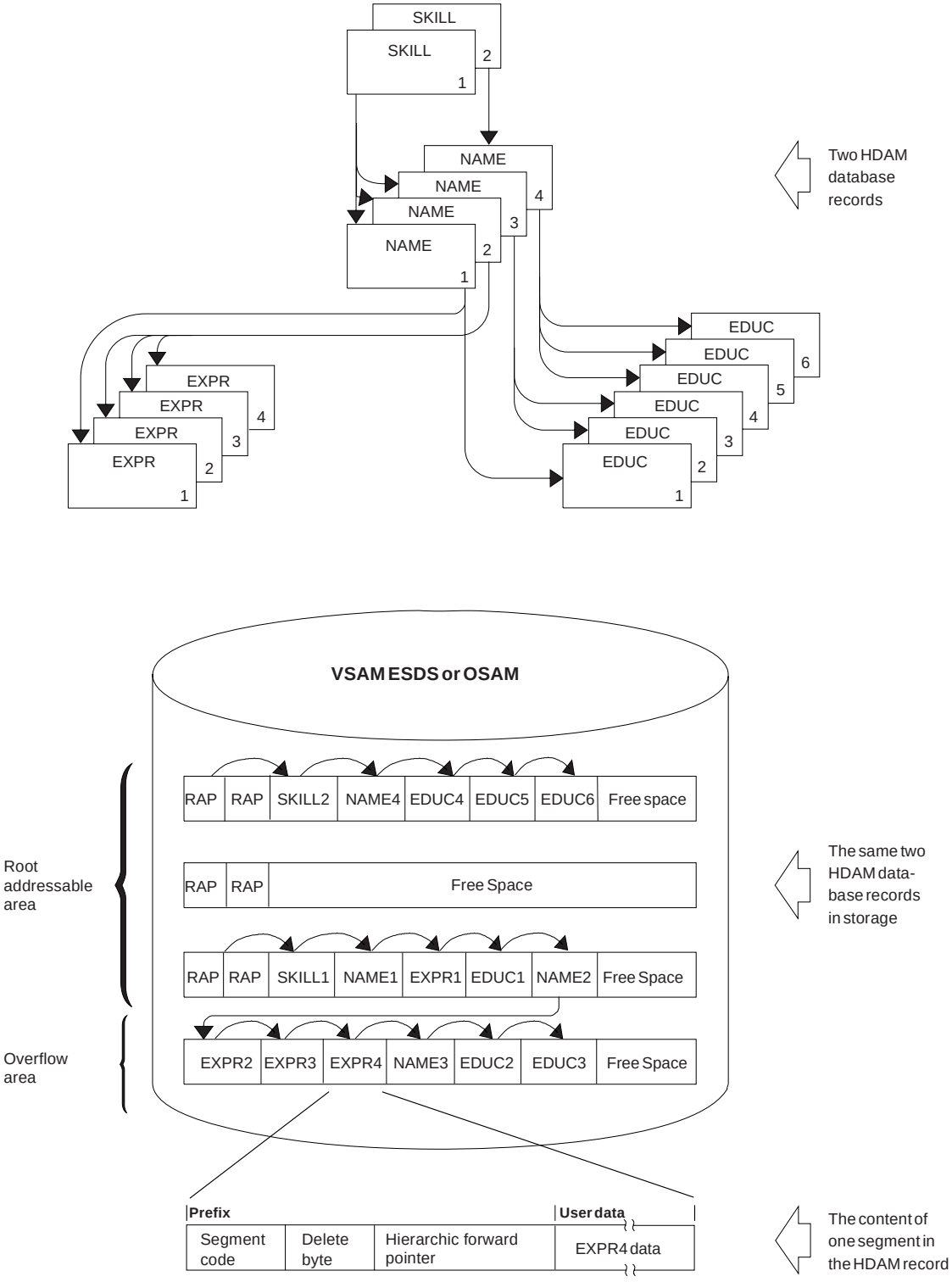


Figure 37. HDAM Database Records in Storage

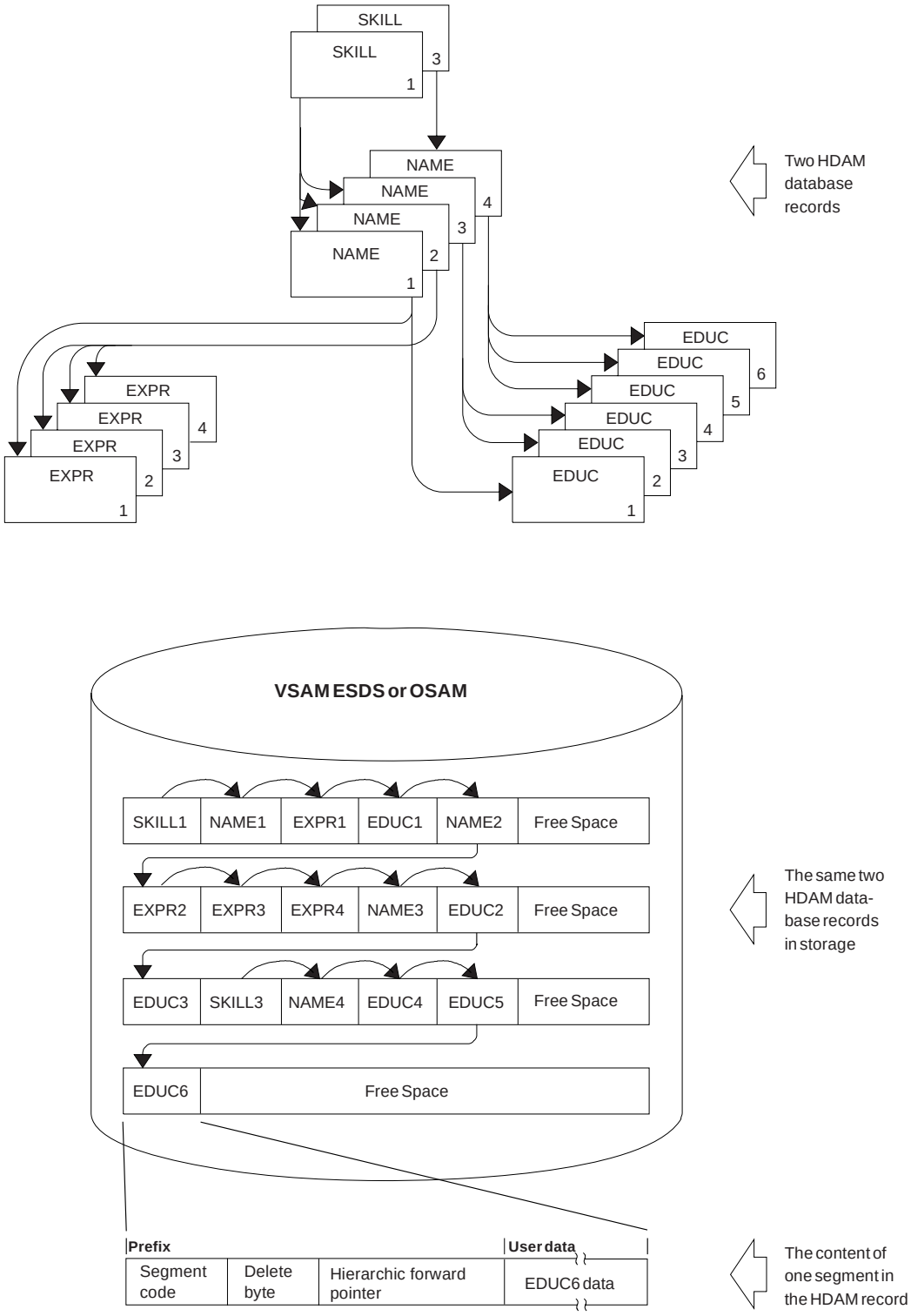
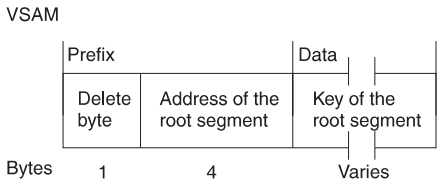


Figure 38. HIDAM Database Records in Storage

The format of an index segment looks like this:



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 database must have unique sequence fields. Each index segment is a separate logical record. Figure 39 shows the index database IMS would generate when the two database records in Figure 38 on page 70 were loaded.

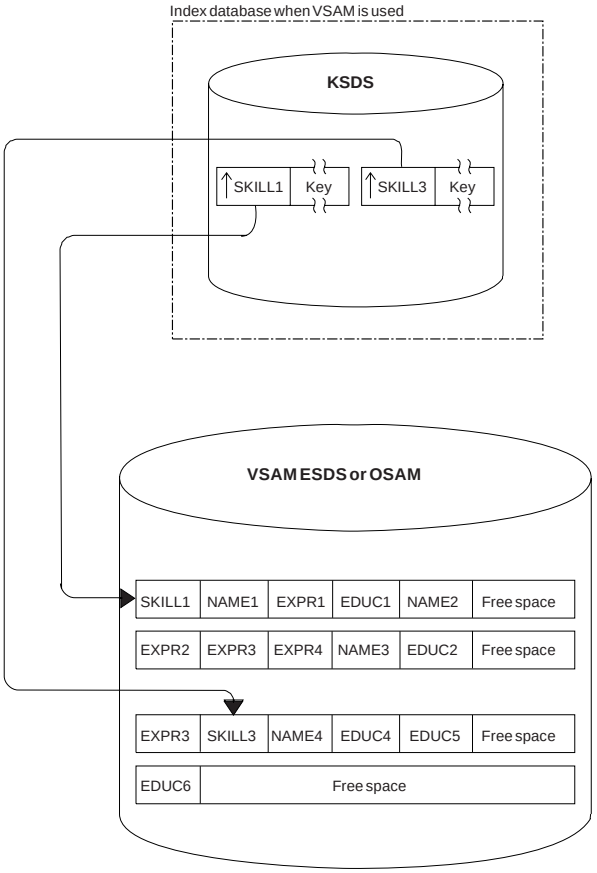


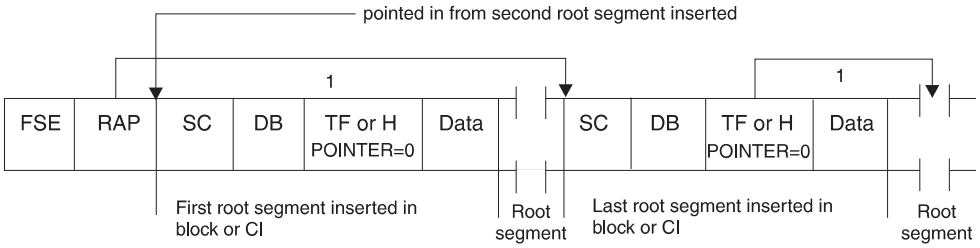
Figure 39. HIDAM Index Databases

Use of RAPs in a HIDAM Database: RAPs are used differently in HIDAM databases than they are in HDAM. In HDAM, 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, RAPs are only generated if you have specified 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. Thus, the RAP points

to the last root inserted into the block or CI, and the hierarchic or twin forward pointer in the first root inserted into the block or CI is made zero. The hierarchic or twin forward pointer in each of the other root segments in the block points to the previous root inserted in the block. Figure 40 shows what happens if you specify PTR=T or PTR=H for root segments in a HIDAM database.

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.



where:
FSE is the free space element
RAP is the root anchor point
SC is the segment code
DB is the delete byte
TF is twin forward
H is hierarchic forward

1 - if you specify PTR=H for a HIDAM root, you get an additional hierarchic pointer to the first dependent in the hierarchy.

Figure 40. What Happens If You Specify PTR=T or PTR=H for Root Segments in a HIDAM Database

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 or HIDAM. In an HDAM database, the randomizing module generates the root segment's (and therefore the database record's) location. In a HIDAM database, the HIDAM 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. Figure 41 shows how a dependent segment is found when PCF and PTF pointers are used.

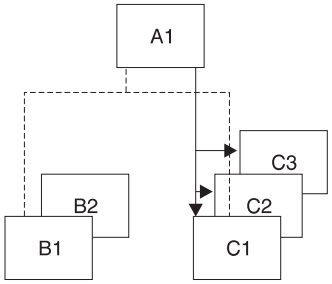


Figure 41. How Dependent Segments Are Found Using PCF and PTF Pointers

Unqualified Calls: When an unqualified call is issued for a segment, the way in which the search proceeds depends on:

- Whether the database is HDAM or HIDAM
- 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 or HIDAM.

Inserting Root Segments into an HDAM Database: After initial load, root segments are inserted into an HDAM database in exactly the same way they are inserted during initial load. This process is explained in “How HDAM Records Are Stored” on page 65.

Inserting Root Segments Into a HIDAM Database: After initial load, root segments are inserted into a HIDAM database as follows (see Figure 42 on page 74):

1. The HIDAM 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 by either moving existing index segments “over” to make room for the new one or by splitting the CI or control area (CA).
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. How this algorithm works is described in “How the HD Space Search Algorithm Works” on page 78.

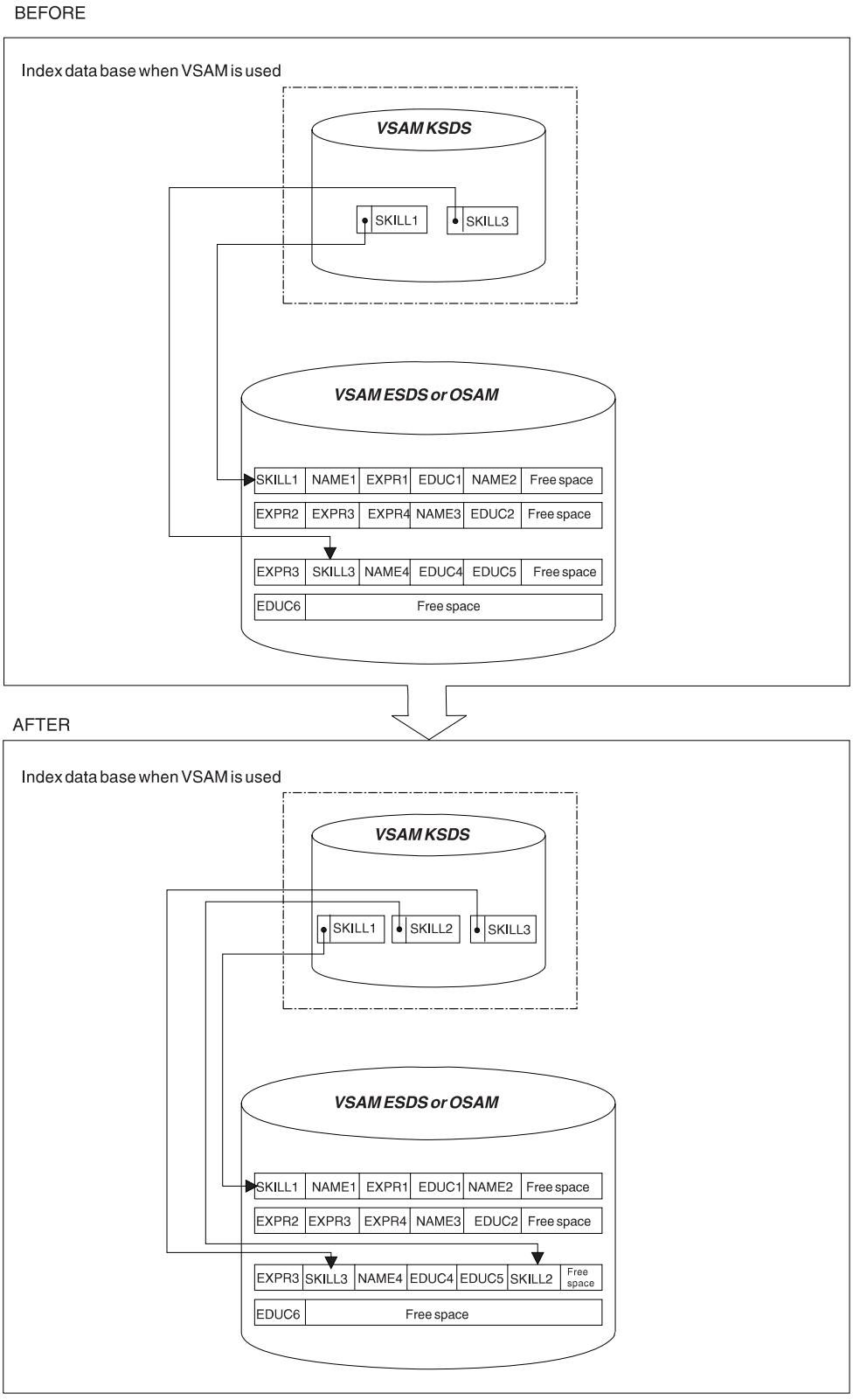


Figure 42. Inserting a Root Segment into a HIDAM Database

Updating the Space Management Fields When a Root Segment Is Inserted:
This section contains diagnosis, modification, or tuning information.

When a root segment is inserted into an HD database, the space management fields need to be updated. Figure 43 on page 76 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 (and therefore has a root addressable area).
- VSAM is the access method, so 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 bit map (in VSAM, the bit map is always in the second CI in the database). The second bit in the bit map 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.

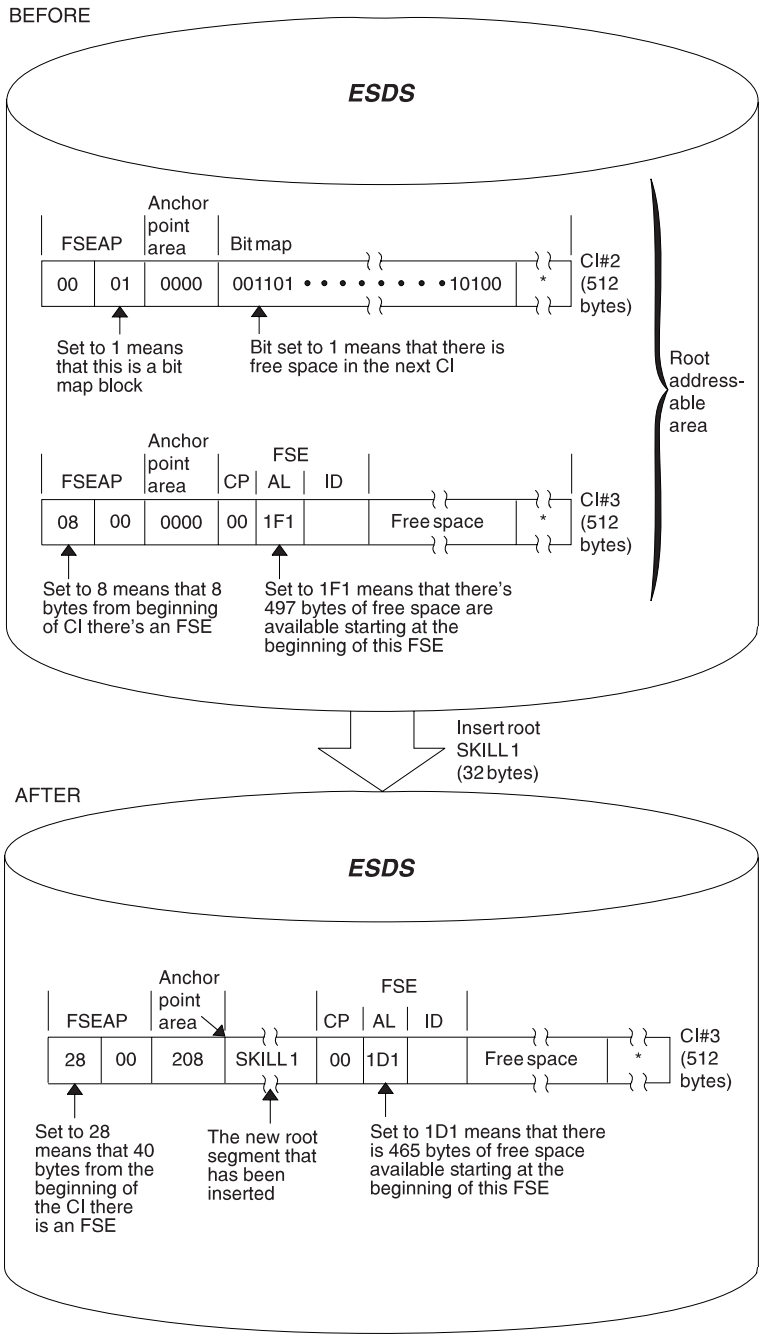
The SKILL1 root segment to be inserted is only 32 bytes long, so CI #3 has plenty of space to store SKILL1.

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  =  CI Size  x  CI#-1  +  Offset within CI
  value                           to root segment
  ```

  In this case, the pointer value is 1032 (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.



* = 7 bytes of control information in the logical record because VSAM is being used

Figure 43. Updating the Space Management Fields in an HDAM Database

Inserting Dependent Segments

After initial load, dependent segments are inserted into HD databases using the HD space search algorithm. How this algorithm works is described in "How the HD Space Search Algorithm Works" on page 78.

As with the insertion of root segments into an HD database, the various space management fields in the database need to be updated. (This process was explained and illustrated in the previous section, "Updating The Space Management Fields When a Root Segment Is Inserted.")

Deleting Segments

Diagnosis, Modification or Tuning Information

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. (This process was explained and illustrated in a previous section called “Updating The Space Management Fields When a Root Segment Is Inserted.”)

- The bit map 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 bit map says whether enough space exists in the block or CI to hold a segment of the *longest* type defined. So, if the deleted segment did not free up enough space for the longest segment type defined, the bit map 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 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.

End of Diagnosis, Modification or Tuning Information

Diagnosis, Modification or Tuning Information

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.

End of Diagnosis, Modification or Tuning Information

Replacing Segments

This section contains diagnosis, modification, or tuning information.

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

This section contains diagnosis, modification, or tuning information.

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.

Root Segment: The most desirable block depends on the access method. For HDAM 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 roots, if the root does not have a twin backward pointer, the most desirable block is the one containing the root with the next highest 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 hierarchic 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 (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 bit map). The second-most desirable block or CI is a block or CI that was left free when the database was loaded or reorganized. Every nth block or CI can be left free by specifying the FRSPC= keyword in the DATASET macro of the DBDGEN utility. If you do not specify in the FRSPC= keyword that every nth block or CI be left free, the HD space search algorithm will not search for the second-most desirable block or CI.

For more information on the FRSPC= and SEARCHA= keywords, see “Database Description (DBD) Generation” in *IMS/ESA Utilities Reference: System*.

All search ranges defined in the HD space search algorithm, excluding steps 9 through 11, 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 through 11, 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.

Look for space:

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 bit map. (The bit map 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 bit map.
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.

7. In any block or CI plus or minus *n* cylinders, as determined by consulting the bit map.
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 bit map. 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 bit map. (This step is not used when a HIDAM database is loaded.)

Notes:

If in load mode processing, step 2 and steps 5 through 8 are skipped.

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. While the PI lock manager provides four locking levels, 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, the anchor point. Therefore, for HDAM, 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.

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 terminates. 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.

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 HIDAM and HDAM Databases: If you access a HIDAM root via 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 you access an HDAM 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 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 updater commits. If the PI lock manager is used, locks are needed to access all root and dependent segments chained from the anchor until the updater 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.

Registering Databases

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 is successful, the error condition no longer exists and recovery is not needed.

Although databases need not be registered in DBRC in order for the error handling to work, it is highly recommended. If an error occurs on a non-registered database and the system terminates, the database could be damaged if the system is restarted and a `/DBR` command is not issued prior to accessing the database. The reason for this is that restart causes the error buffers to be restored as they were when the system terminated. If the same block had been updated during the batch run, the batch update would be overlaid.

Chapter 5. Choosing Additional Database Functions

About This Chapter

After you have determined the type of database that best suits your application's processing requirements, you are ready to determine which additional IMS functions you need to use.

This chapter explains the following functions and describes when and how to use them:

- Logical relationships
- Secondary indexes
- Variable-length segments
- Segment edit/compression facility
- Data Capture exit routines
- Field-level sensitivity
- Multiple data set groups

Notes:

1. These functions do not apply to GSAM, MSDB, HSAM, and SHSAM databases.
2. Only the variable-length segment function, the segment edit/compression facility, and the Data Capture exit routine apply to DEDBs.

Using Logical Relationships

Logical relationships is a function you can use to resolve conflicts in the way application programs need to view segments in the database. With logical relationships, you can:

- Give an application program access to segment types in an order other than the one defined by the hierarchy
- Give an application program access to 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

For example, suppose two databases exist, one for orders that a customer has placed and one for items that can be ordered. The first database is called the ORDER database; the second is called the ITEM database.

The ORDER database contains data such as:

Order number

Customer's name and address

Type of items ordered

Quantity of each item ordered

Delivery data

The ITEM database contains data such as:

- Type of items that can be ordered
- Quantity of each item in stock
- Quantity of each item in stock that has been ordered but not yet delivered

Defining a Logical Relationship

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 Figure 44 on page 86, 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. Figure 44 on page 86 is a simple implementation of a logical relationship. In this case, ORDER is the physical parent of ORDITEM. ORDITEM is the physical child of ORDER but 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.

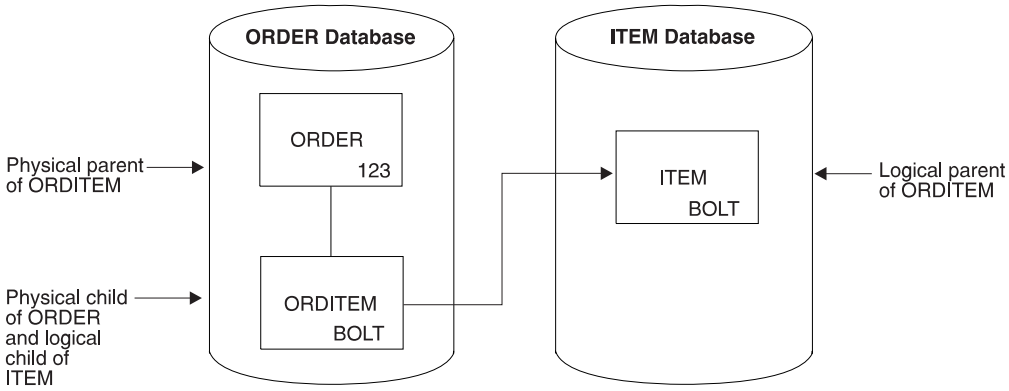


Figure 44. A Simple Logical Relationship

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.

There are three ways in which a logical relationship can be established or implemented. These methods of implementation are as follows:

Unidirectional logical relationship

Bidirectional physically paired logical relationship

Bidirectional virtually paired logical relationship

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. Figure 45 on page 87 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.

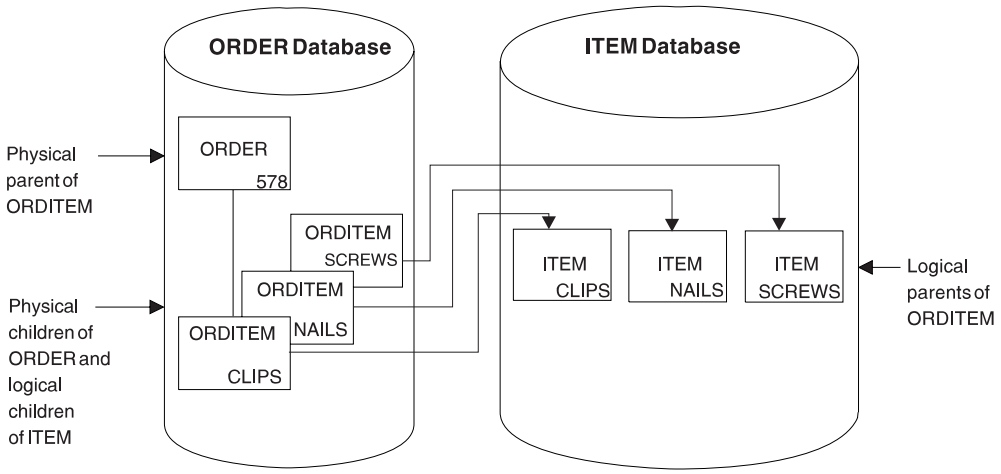


Figure 45. Unidirectional Logical Relationship

It is possible to establish two unidirectional relationships, as shown in Figure 46 . 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.

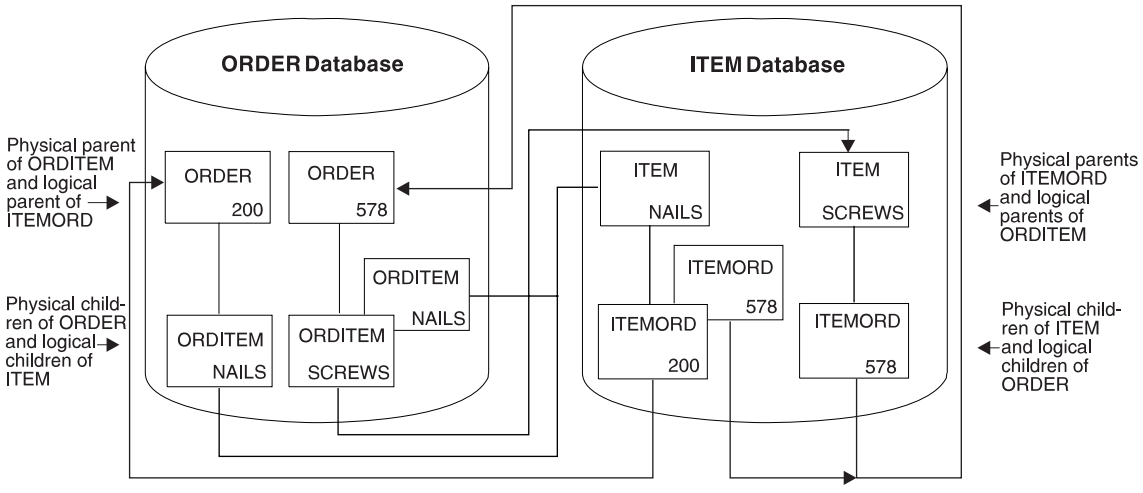


Figure 46. Two Unidirectional Logical Relationships

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. Figure 47 shows a bidirectional physically paired logical relationship that has been established between the ORDER and ITEM databases.

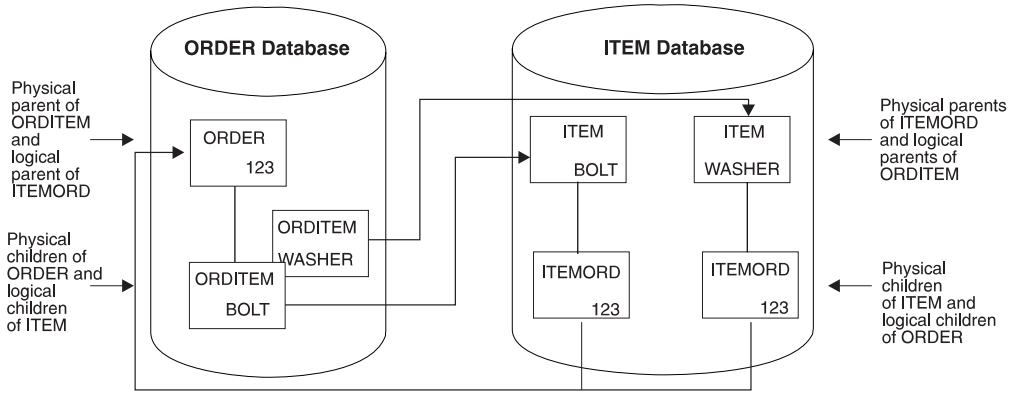


Figure 47. Bidirectional Physically Paired Logical Relationship

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 Figure 47 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 Figure 47 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.

Figure 48 on page 89 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.

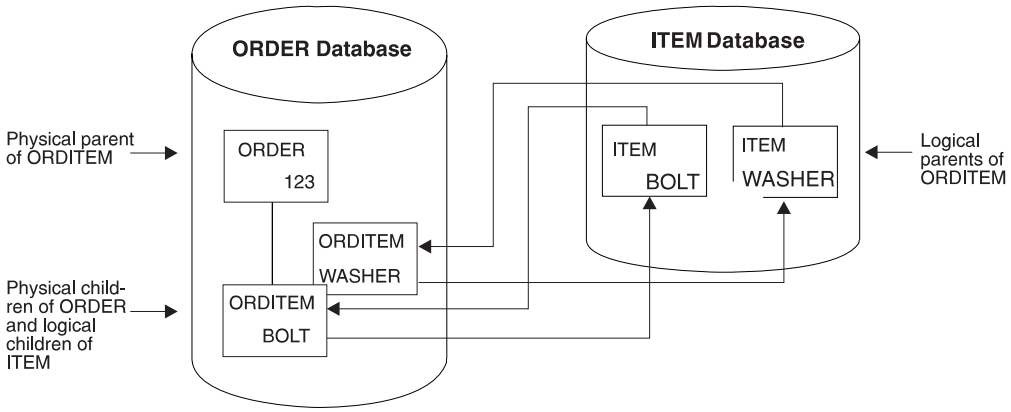


Figure 48. Bidirectionally Virtually Paired Logical Relationship

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.

Pointing and Pointers in Logical Relationships

In all logical relationships the logical child establishes a path between two segment types. The path is established by use of pointers. The following sections look at pointing in logical relationships and the various types of pointers that can be used. Four types of pointers can be specified for logical relationships:

- Logical parent pointer
- Logical child pointer
- Physical parent pointer
- Logical twin pointer

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 HIDAM database.

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 and HIDAM) 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. A direct LP pointer is stored in the logical child's prefix, along with any other pointers, and is four bytes long. Figure 49 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 hierarchic 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.

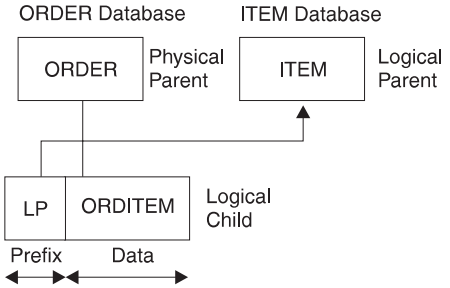


Figure 49. Direct Logical Parent (LP) Pointer

In Figure 49, 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. Figure 50 on page 91 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 the logical parent is in 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.

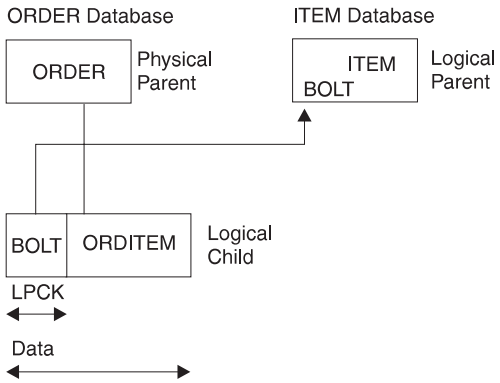


Figure 50. Symbolic Logical Parent (LP) Pointer

In Figure 50, 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.

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.

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. Figure 51 on page 92 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 explained in the next section.

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. Figure 51 shows a LCF pointer.

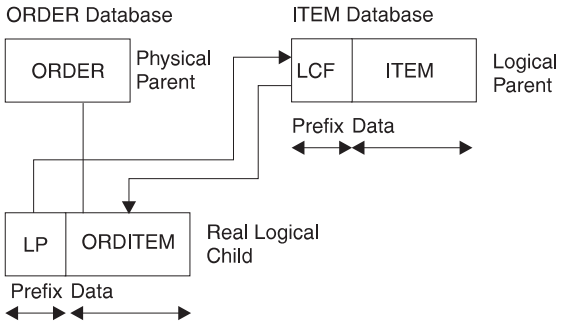


Figure 51. Logical Child First (LCF) Pointer (Used in Virtual Pairing Only)

In Figure 51, 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 Figure 51, 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. Figure 52 on page 93 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.

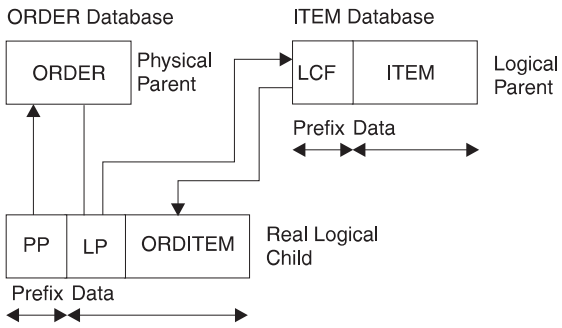


Figure 52. Physical Parent (PP) Pointer

In Figure 52, 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.

Figure 53 on page 94 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 Figure 53 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. Figure 53 on page 94 shows a LTF pointer.

Pointing and Pointers in Logical Relationships

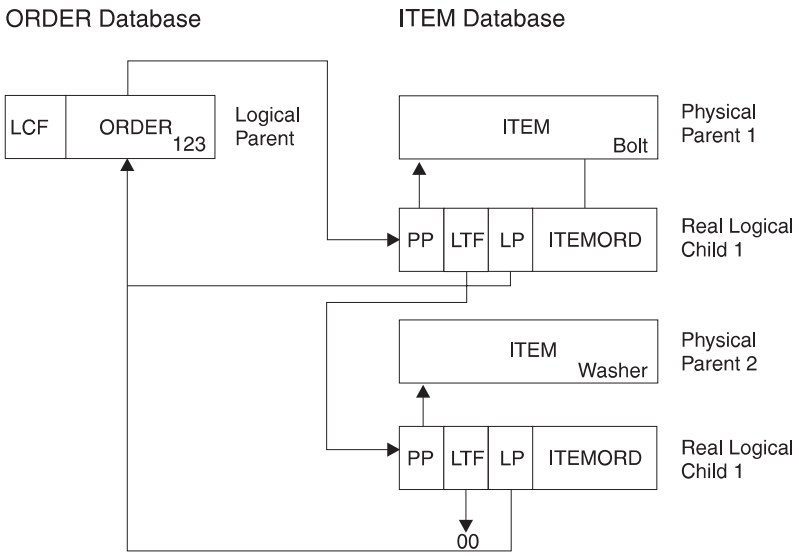


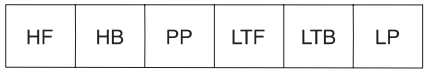
Figure 53. Logical Twin Forward (LTF) Pointer (Used in Virtual Pairing Only)

In Figure 53, 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 pointer and consists of the 4-byte RBA of the logical twin stored after it.

Sequence of Pointers in a Segment's Prefix

This section contains diagnosis, modification, or tuning information.

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:



or

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 Figure 54, 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 (FID) and variable intersection data (VID).

Fixed Intersection Data

Data stored in the logical child is called fixed intersection data (FID). 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 Figure 54, BOLT and WASHER are the LPCK, and the 5000 and 6000 are the FID. The FID can consist of several fields, all of them residing in the logical child segment.

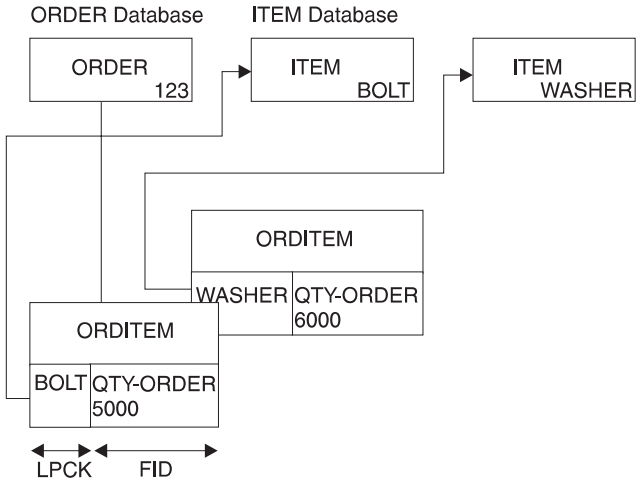


Figure 54. Fixed Intersection Data

Variable Intersection Data

VID 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.

Intersection Data

For each logical child occurrence, there can be as many occurrences of dependent segments containing intersection data as you need.

Figure 55 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 3. 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 VID can all exist at the same level in the hierarchy as shown in the figure, or they can be dependents of each other.

FID, VID, 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, VID 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. FID, on the other hand, must be stored on both sides of the relationship when physical pairing is used. IMS automatically maintains the FID 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 FID in a physically paired relationship.

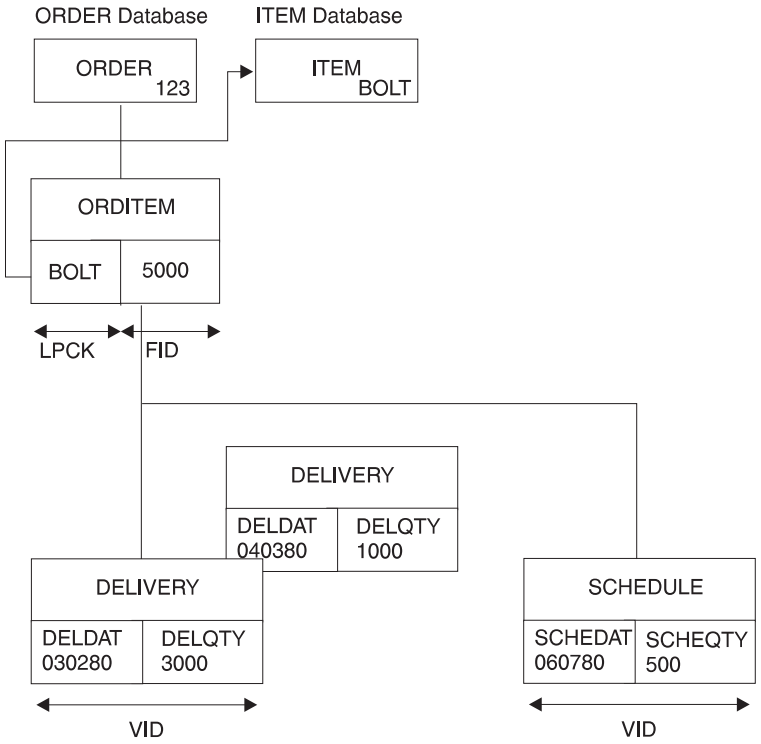


Figure 55. Variable Intersection Data

Establishing Logical Relationships Between Segments in the Same Database (Recursive Structures)

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. Figure 56 shows the list of parts needed to manufacture this bicycle. The number next to each part is the quantity of that part needed to make one Model 1 bicycle. 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. Figure 57 on page 98 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.

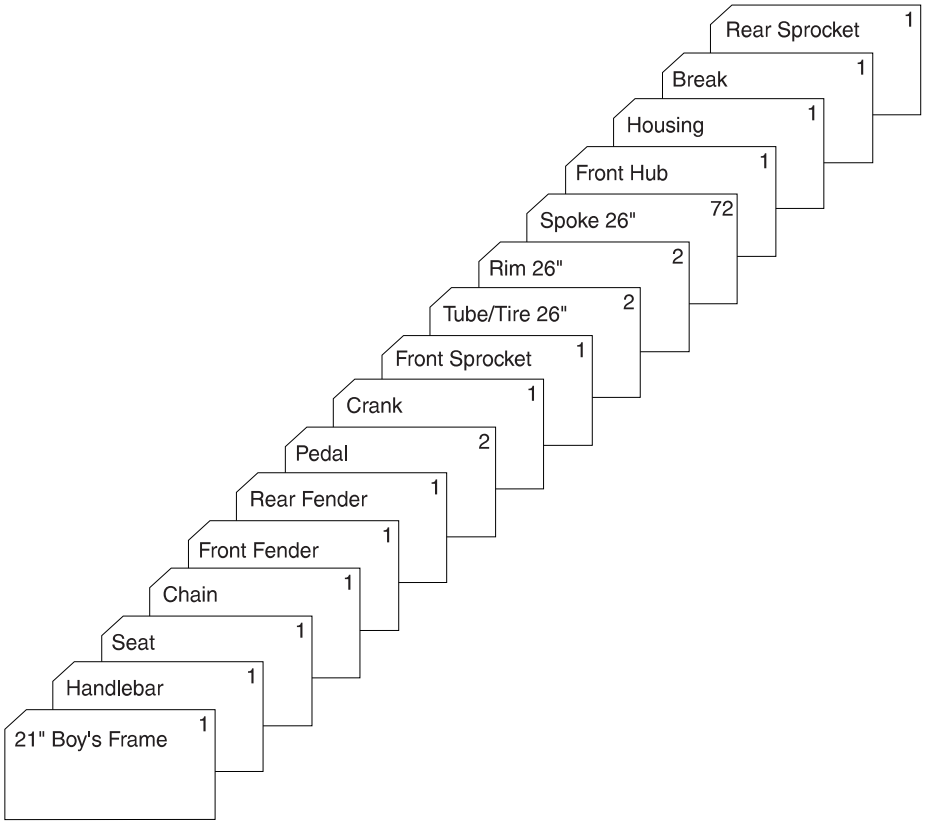


Figure 56. Model 1 Parts List

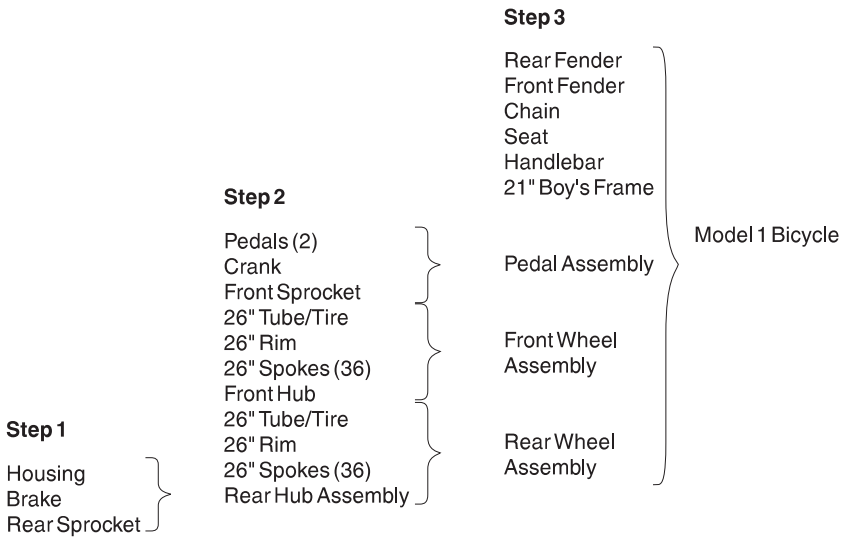


Figure 57. Assembly Steps to Make a Model 1 Bicycle

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. Figure 58 on page 99 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. Figure 59 on page 100 shows how this is done using the data for the Model 1 bicycle.

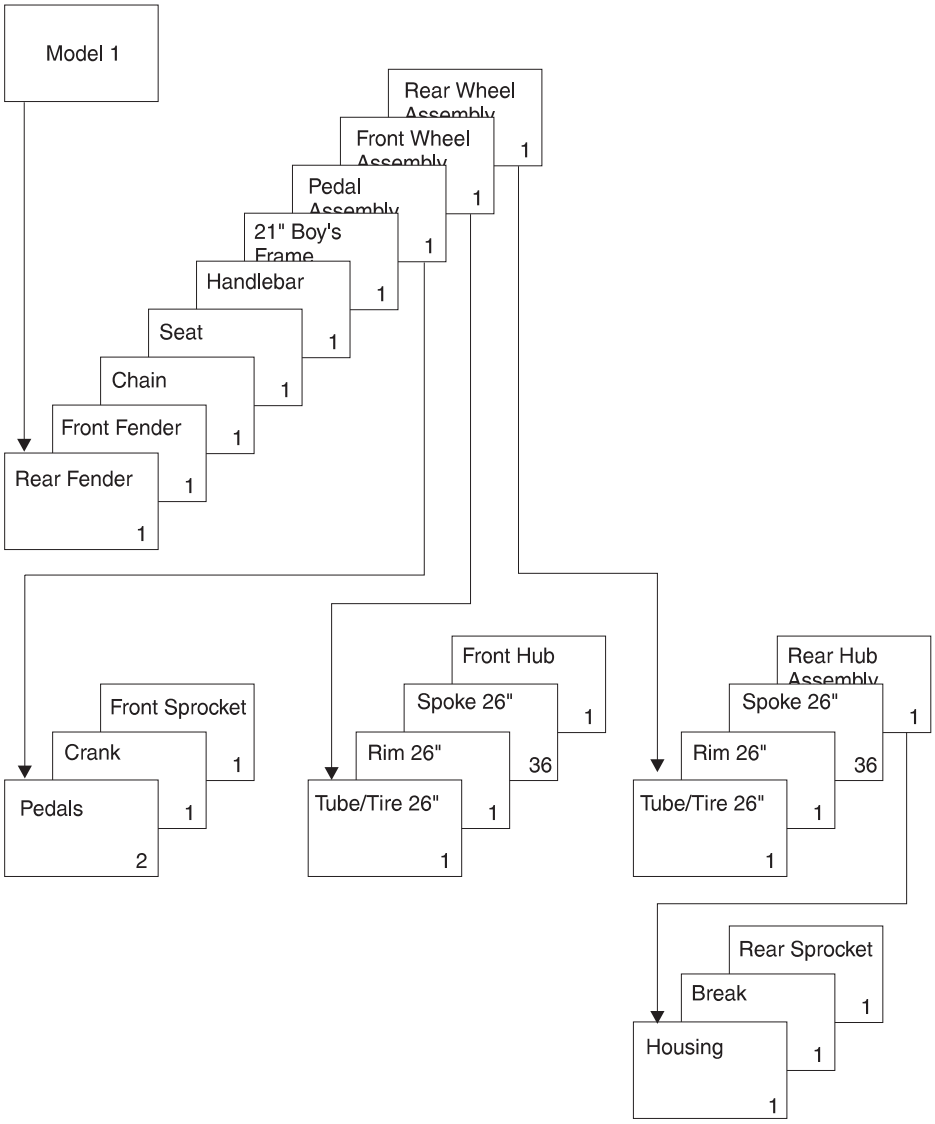


Figure 58. Model 1 Components and Subassemblies

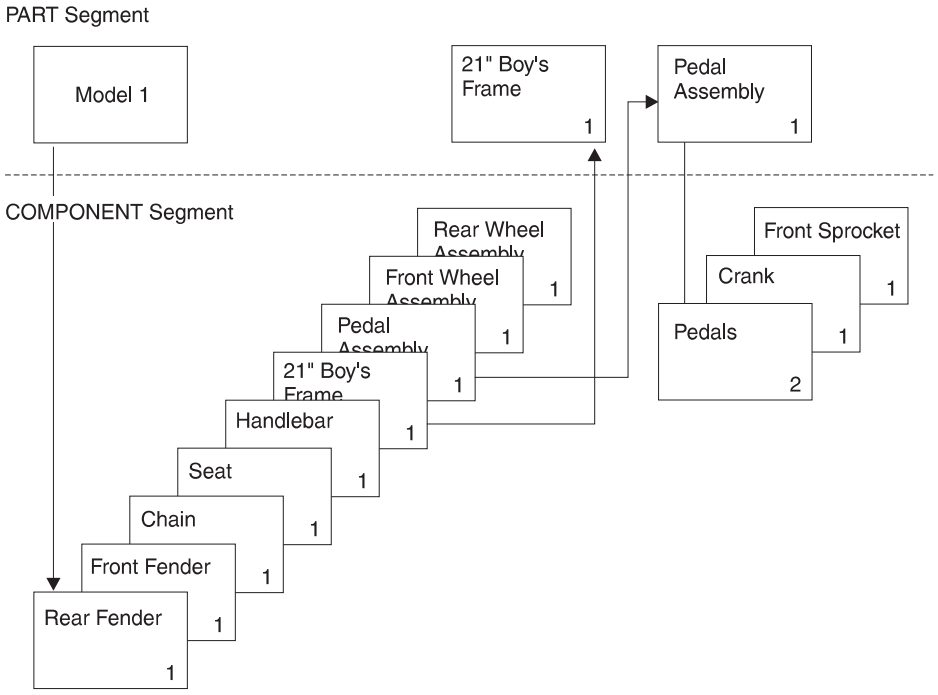


Figure 59. Database Records for the Model 1 Bicycle

In Figure 59, 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.

Figure 60 on page 101 shows the segments, in *addition* to those in Figure 59, 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 Figure 59. 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.

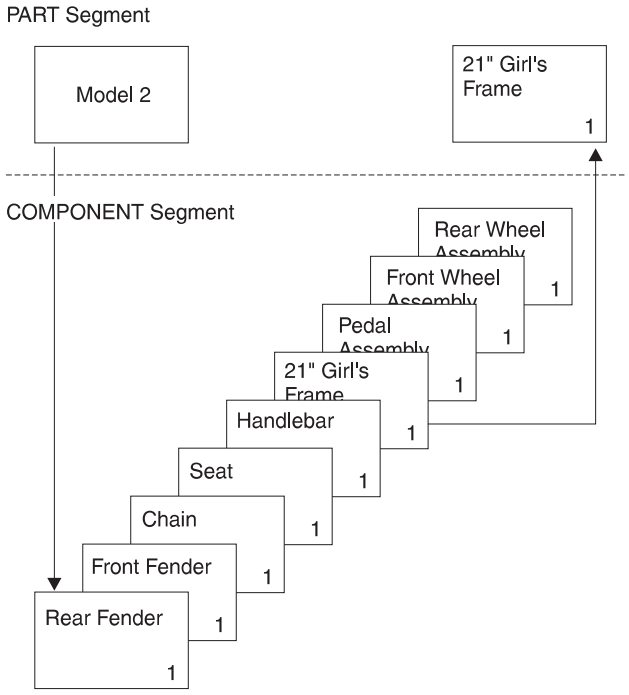


Figure 60. Extra Database Records Required for the Model 2 Bicycle

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 59 on page 100, 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.

Paths Used 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, as shown in Figure 61. Definition of the path and the concatenated segment type is done in what is called a logical database. The logical database is examined later in this chapter.

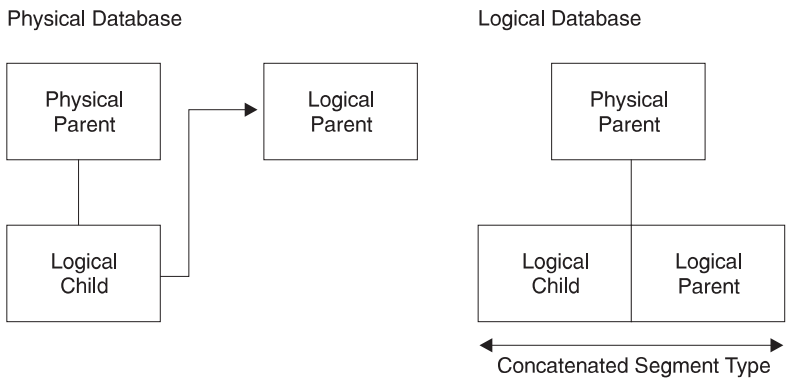


Figure 61. Defining a Physical Parent to Logical Parent Path in a Logical Database

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 Figure 62. Again, definition of the path is done in a logical database.

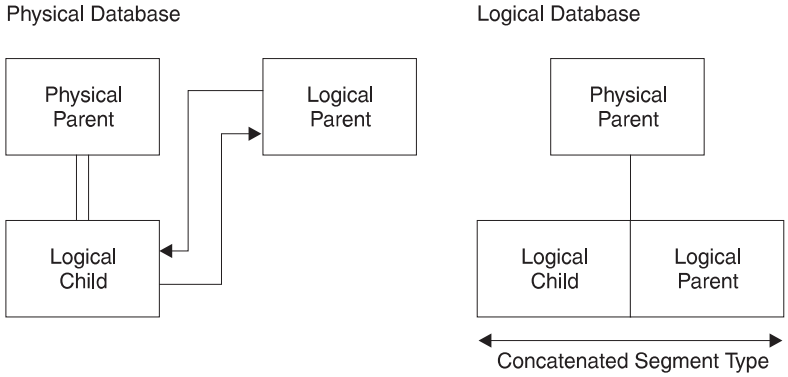


Figure 62. Defining a Logical Parent to Physical Parent Path in a Logical Database

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.

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 can be used for intersection data.

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. Figure 63 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.

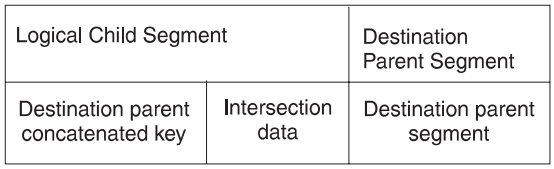


Figure 63. Format of a Concatenated Segment Returned to User I/O Area

Defining Sequence Fields for Databases Using Logical Relationships

To avoid potential problems in processing databases using logical relationships, unique sequence fields should be defined in all logical parent segments. 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 (described in “Chapter 14. Tuning Your Database” on page 323) 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.

Defining Sequence Fields for Real Logical Children

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.

Defining Sequence Fields for Virtual Logical Children

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.

Relationship of Control Blocks When a Logical Relationship Is Used

When a logical relationship is used, you must define the physical databases involved in the relationship to IMS. This is done using a *physical* DBD. In addition, many times you must define the logical structure of IMS since this is the structure 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.

Figure 64 shows the relationship between these three control blocks. It assumes that the logical relationship is established between two physical databases. The following sections explain how the physical and logical DBD are coded when a logical relationship is defined.

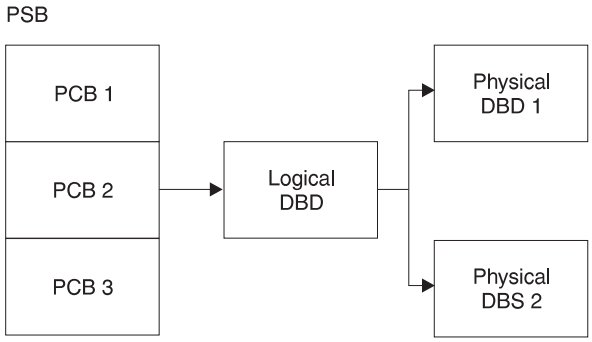


Figure 64. Relationship of Control Blocks When a Logical Relationship is Used

How to Specify Use of 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. Figure 66 on page 106 shows an example of how the physical DBD is coded.

In the SEGM statements of the following examples, 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.

Figure 65 shows the layout of segments. Figure 66 on page 106 shows physical DBDs for unidirectional relationships.

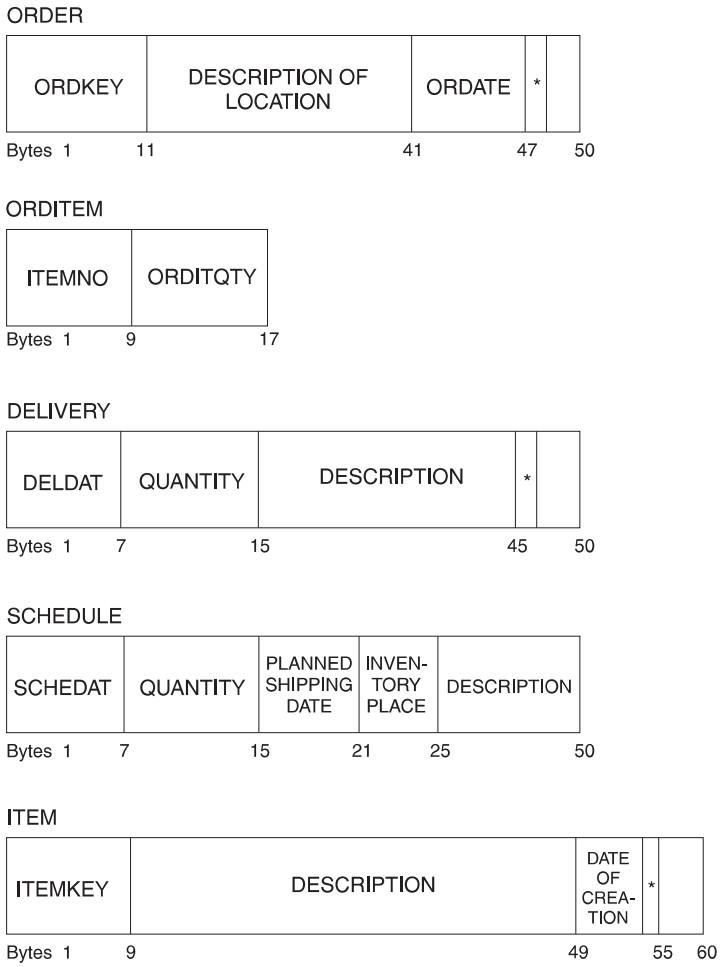


Figure 65. Layouts of Segments Used in the Examples

This is the hierarchic 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 FID, and DELIVERY and SCHEDULE are VID.

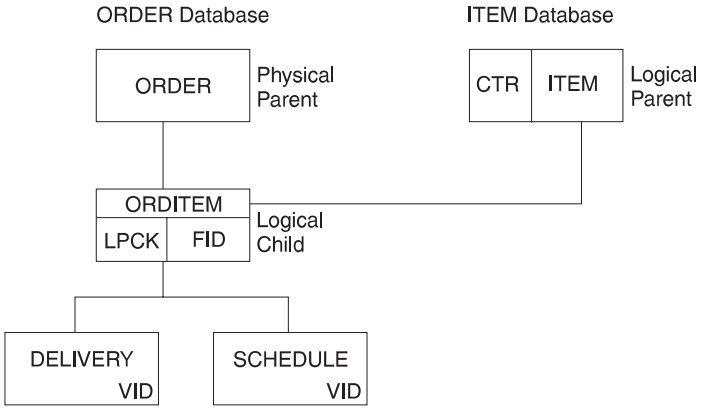


Figure 66. Physical DBDs for Unidirectional Relationship Using Symbolic Pointing

This is the DBD 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
```

This is the DBD 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
```

Notes to Figure 66:

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 FID. The LPCK is the key of the ITEM segment, which is 8 bytes long. The length of the FID 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 FID 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.

Specifying Bidirectional Logical Relationships

Figure 66 on page 106 shows the coding for a unidirectional relationship. When defining a bidirectional relationship with physical pairing, you need to include an LCHILD statement under both logical parents. In addition to other pointers, you need to include 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.

For more information on coding logical relationships, see *IMS/ESA Utilities Reference: Database Manager*.

Checklist of Rules for Defining Logical Relationships in Physical Databases

This section provides the list of rules that must be followed when defining logical relationships in physical databases. In all cases, references are to segment types, *not* occurrences.

Logical Child Rules

- A logical child must have a physical and a logical parent.
- A logical child can have only one physical and one logical parent.

Rules for Defining Logical Relationships

- 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

- 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

A physical parent of a logical child cannot also be a logical child.

How to Specify Use of Logical Relationships in the Logical DBD

To identify which segment types are used in a logical data structure, you must code a logical DBD. Figure 67 on page 109 shows an example of how the logical DBD is coded. The example is a logical DBD for the same physical databases defined in the previous section.

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.

Figure 67 on page 109 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 VID.

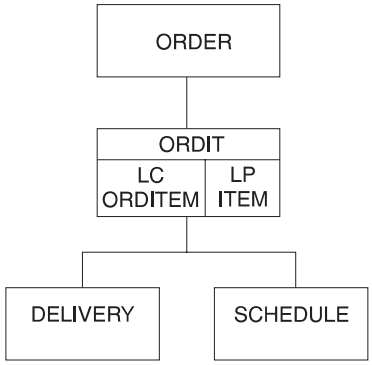


Figure 67. Logical DBD for a Unidirectional Relationship Using Symbolic Pointing

The following information is the *logical* DBD for the logical data structure shown above:

```
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 Figure 67:

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 66 on page 106.

- 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 VID 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.

Checklist of Rules for Defining Logical Databases

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 Figure 68 on page 111, 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 DB3. 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.

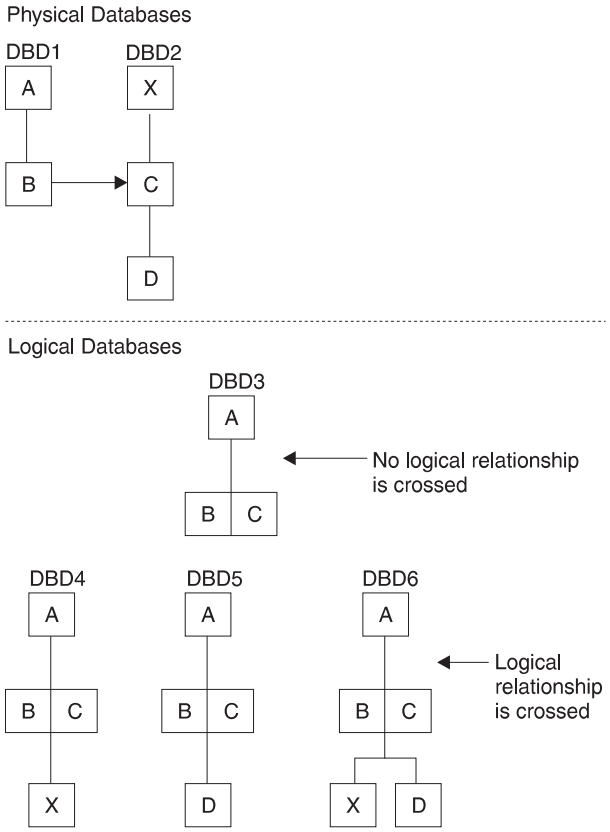


Figure 68. Definition of Crossing a Logical Relationship

Definition of First and Additional Logical Relationships Crossed

More than one logical relationship can be crossed in a hierarchic path in a logical database. Figure 69 on page 112 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 hierarchic 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.

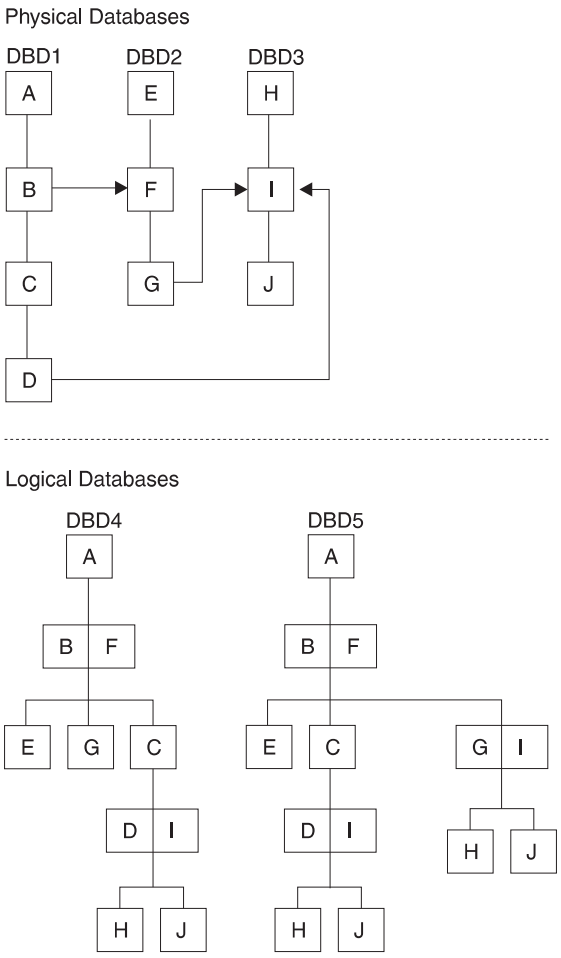


Figure 69. The First Logical Relationship Crossed in a Hierarchic Path of a Logical Database

In DBD5 in Figure 69, an additional concatenated segment type GI, is defined that was not included in DBD4. GI allows access to segments in the hierarchic 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 hierarchic path of a logical database, access to all segments in the hierarchic 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 Figure 70 on page 113.
- 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 Figure 70.

When an additional logical relationship is crossed in a logical database, access to all segments in the hierarchic path of the destination parent is made possible, just as in the first crossing.

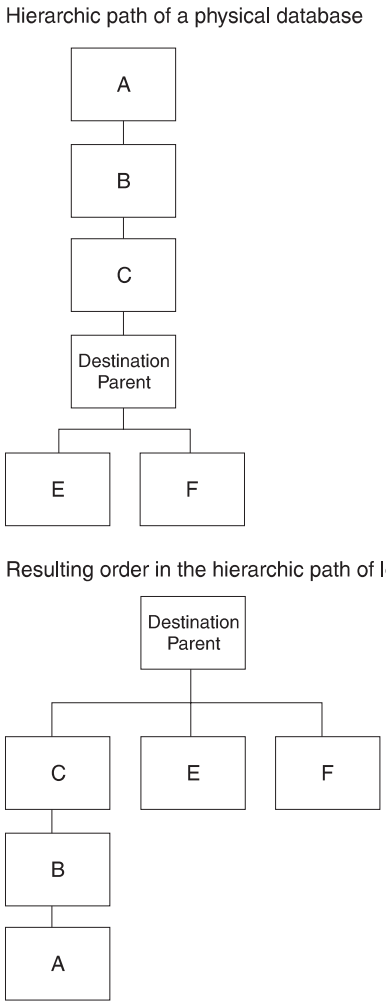


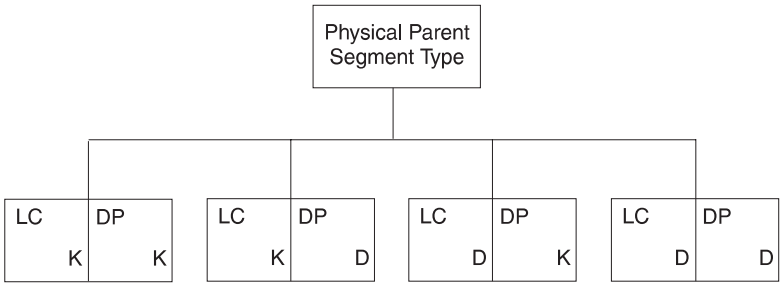
Figure 70. Logical Database Hierarchy Enabled by Crossing the First Logical Relationship

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/or 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 hierarchic 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 hierarchic 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. Figure 71 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.



where:

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

Figure 71. Single Concatenated Segment Type Defined Multiple Times with Different Combinations of Key and Data Sensitivity

Choosing Replace, Insert, and Delete Rules for Logical Relationships

You need to 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.

Figure 72 on page 115 shows example insert, delete, and replace rules. Think a minute about the following questions:

1. Should the CUSTOMER segment in Figure 72 on page 115 be insertable 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?

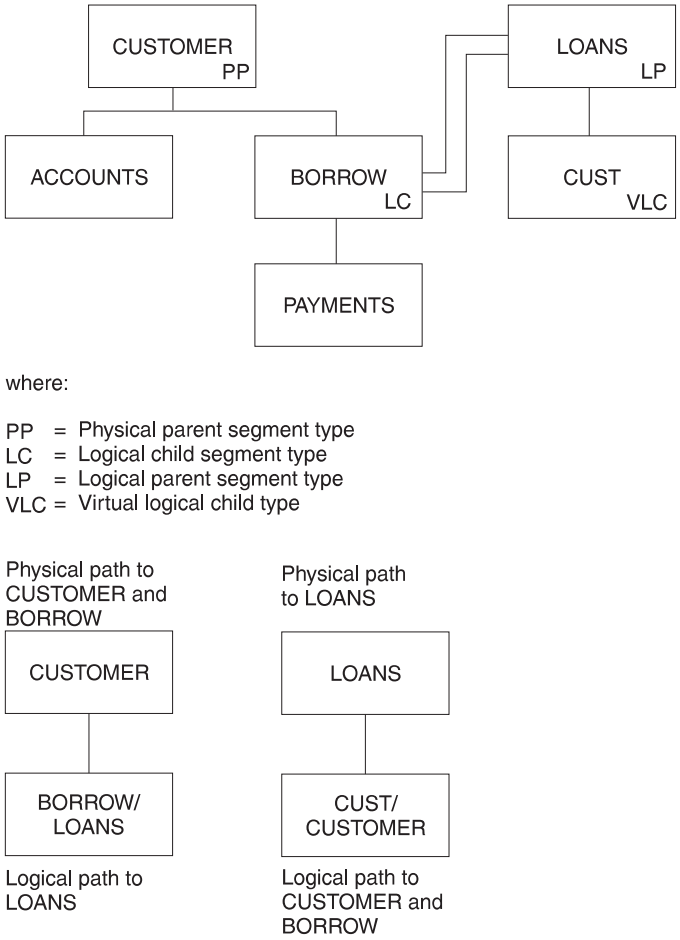


Figure 72. Example of the Replace, Insert, and Delete Rules

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, etc.) 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 terminate 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.

Rules for Logical Relationships

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. These rules are explained in detail in “Appendix B. Replace, Insert, and Delete Rules for Logical Relationships” on page 409.

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. These choices are explained in this section.

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 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

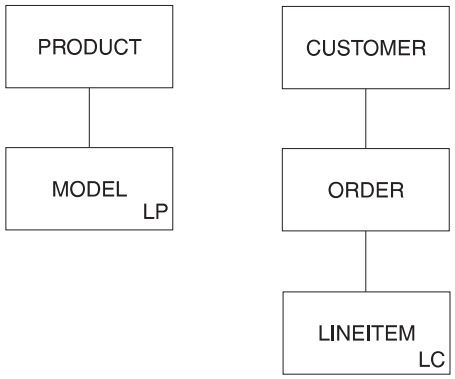
KEY/DATA Considerations

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:

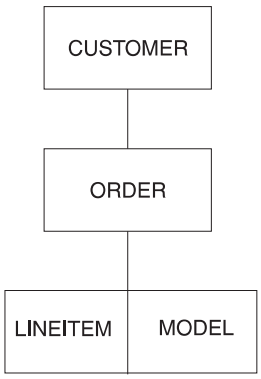
- The logical child part only
- The logical (or destination) parent part only
- Both parts

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 following unidirectional logical relationship:



- Assume you have the following 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.

Performance Considerations for Logical Relationships

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.

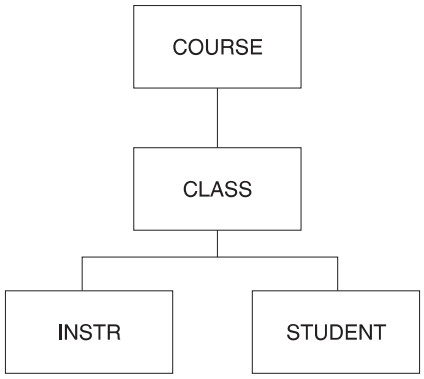
Using Secondary Indexes

Secondary indexes are indexes that allow you to process a segment type in a sequence other than the one defined by the segment's key. A secondary index can also be used to process a segment type based on a qualification in a dependent segment.

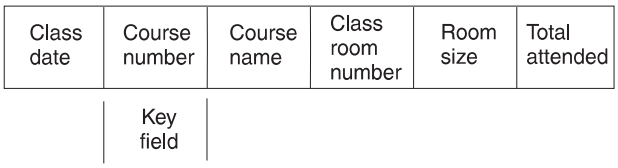
Why Secondary Indexes?

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* your application's processing requirements.

However, the choices you make might suit the processing requirements of some applications better than others. For example, suppose you have an educational database, and a database record in it looks like this:



Suppose the root segment, COURSE, has the following fields in it, and the course number field is a 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).

Secondary indexing is a solution to the different processing requirements of various applications. It allows you to have an index based on any *field* in the database, and not just the key field in the root segment.

Characteristics of Secondary Indexes

Secondary indexes can be used with HISAM, HDAM, and HIDAM databases. A secondary index is in its own separate database and must use VSAM as its access method. Because a secondary index is in its own database, it can be processed as a separate database.

Secondary indexes are invisible to the application program. When an application program needs to do processing 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 simply not coded. If the application program needs to do processing using both the regular processing sequence and the secondary index, the application program's PSB must contain two PCBs, one with PROCSEQ= coded and one without.

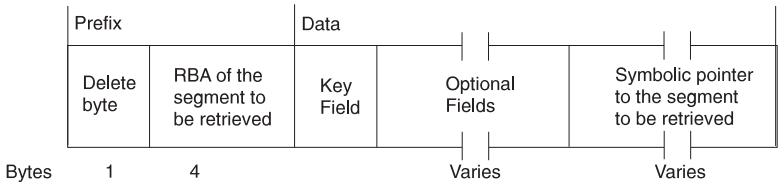
When two PCBs are used, it enables an application program to use two paths into the database and two sequence fields. One path and sequence field is provided by the regular processing sequence, and one is provided by the secondary index. The secondary index gives an application program both an alternative way to enter the database and an alternative way to sequentially process database records.

A final characteristic of secondary indexes is that there can be 32 secondary indexes for a segment type and a total of 1000 secondary indexes for a single database.

Segments Used for Secondary Indexes

Now that the concept and general characteristics of secondary indexes have been explored, the next section looks at how a secondary index works. As shown in Figure 73 on page 121, to set up a secondary index, three types of segments must be defined to IMS. The three types of segments are pointer, target, and source segments.

- *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 looks like this:
  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.)

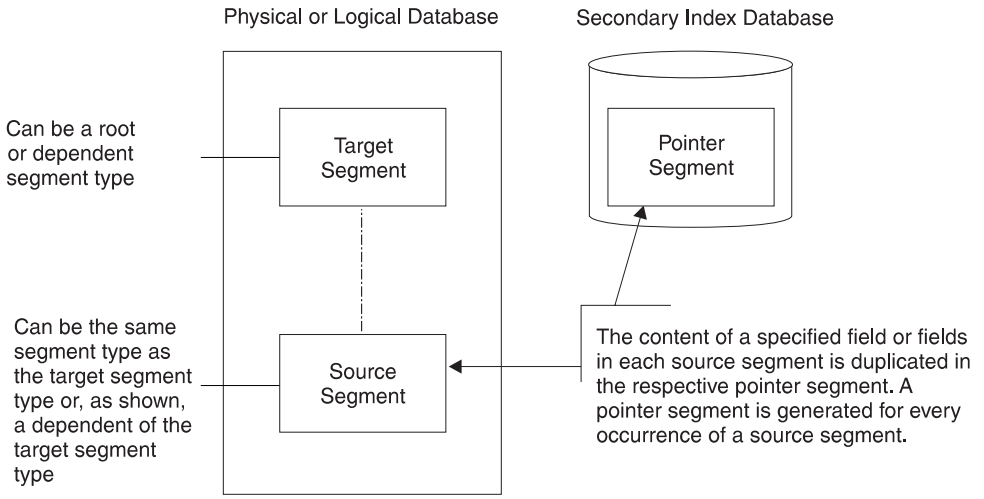


Figure 73. Segments Used for Secondary Indexes

The data portion of the segment contains the key field and other optional fields. (The optional fields are discussed later.) The key field contains the key. The key gets the application program to the correct pointer segment in the secondary index. The application program uses the address in the prefix of the pointer segment to retrieve the necessary segment from the database. If symbolic pointing is used, the key will get the application program to the correct pointer segment in the secondary index. The application program uses the symbolic pointer in the pointer segment to retrieve the necessary segment from the database.

- *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 later).
- *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.

Using the education database in Figure 74 on page 122, you can see how three segments work together. In this example, 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 invokes a search for a pointer segment with a key field that matches the student name. Once the correct pointer segment in the index is found, it contains the address of the *COURSE* segment the application program is trying to retrieve.

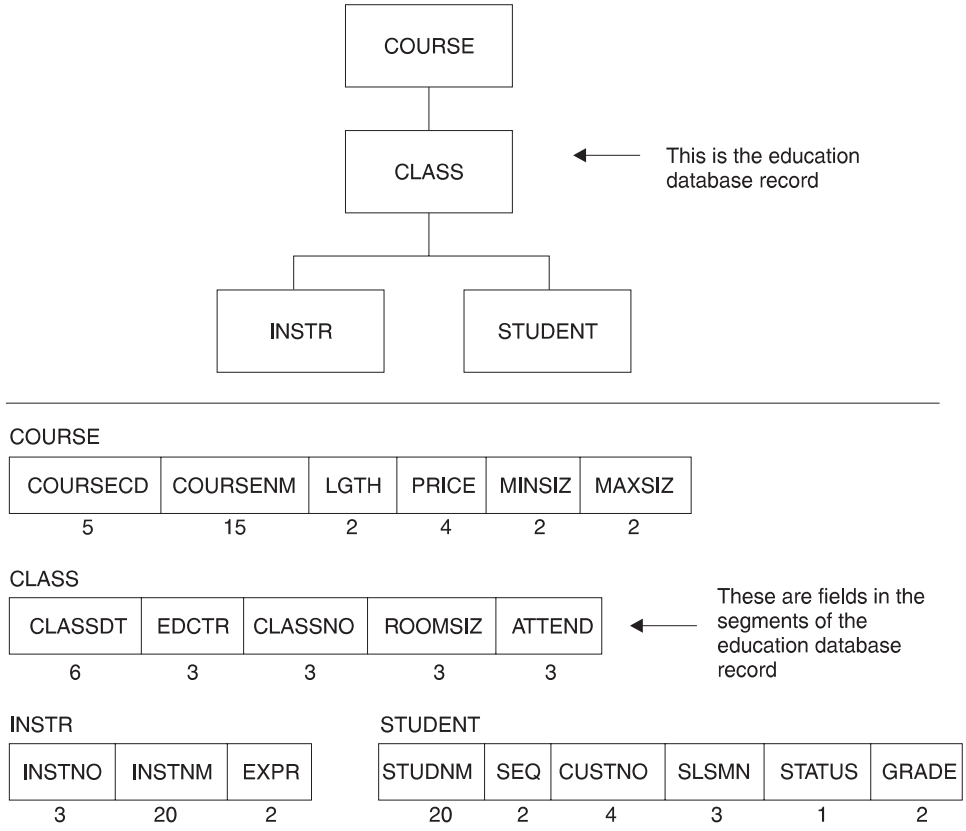


Figure 74. Education Database Record and the Fields in It

Figure 75 on page 123 shows how the pointer, target, and source segments work together.

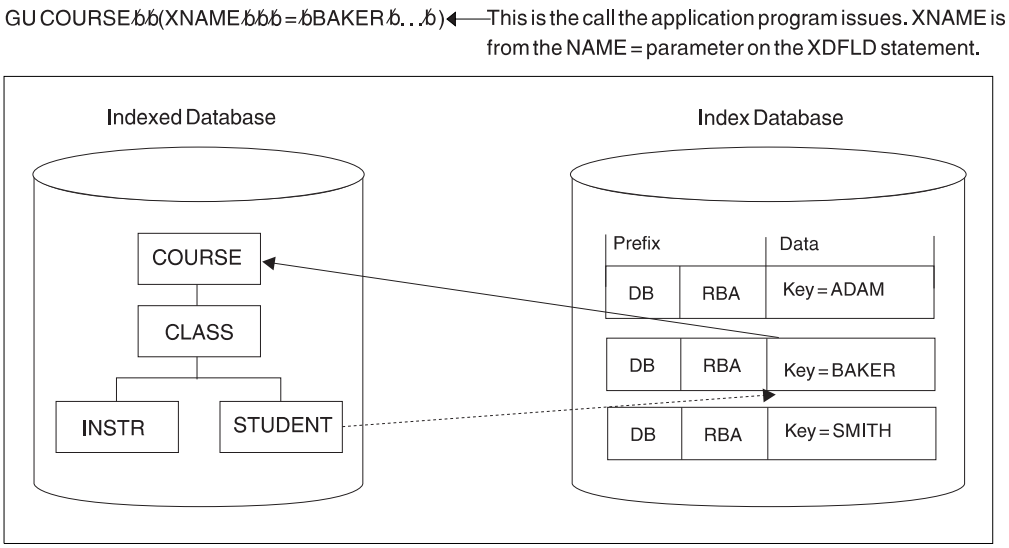


Figure 75. How a Segment Is Accessed Using a Secondary Index

COURSE is the target segment that the application program is trying to retrieve.

STUDENT is the source segment containing the 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 the Hierarchy Is Restructured

When the PROCSEQ= parameter in the PCB is coded (specifying 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 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. Figure 76 on page 124 illustrates this concept.

The target segment (as shown in the figure) is segment G. Target segment G becomes the root segment in the restructured hierarchy. 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.

Secondary Data Structure: 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 76 shows how the application program perceives the database record.

If the same segment is referenced more than once as shown in Figure 76, 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 will issue the message “SEG150” for the duplicate SENSEG names.

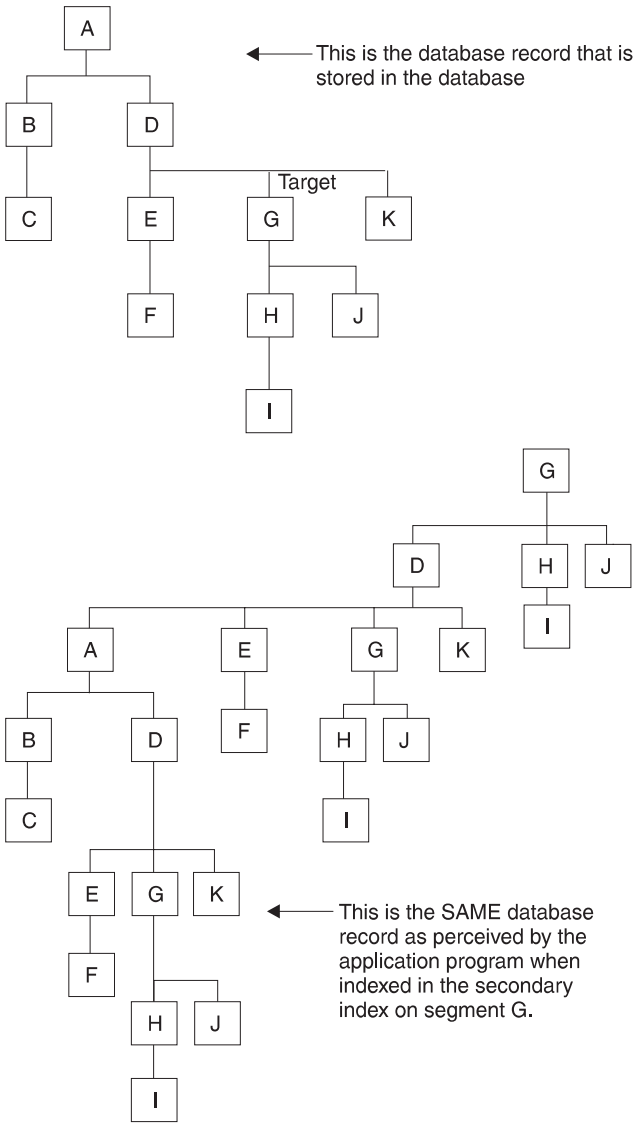
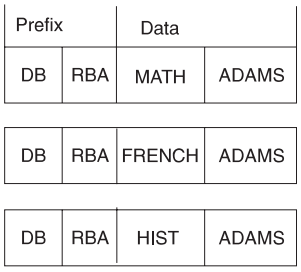


Figure 76. How the Hierarchy in a Database Record Is Restructured When Secondary Indexing Is Used

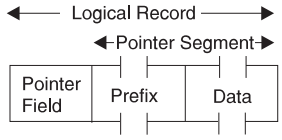
How a Secondary Index Is Stored

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:



Each pointer segment in a secondary index is stored in one logical record. A logical record containing a pointer segment looks like this:



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

This section contains diagnosis, modification, or tuning information.

Figure 77 on page 126 shows the fields in a pointer segment. Like all segments, the pointer segment has a prefix and data portion. The prefix portion has a delete byte, and when direct rather than symbolic pointing is used, it has the address of the target segment. 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).

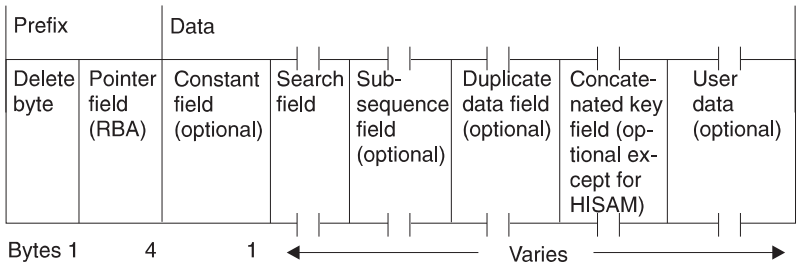


Figure 77. Fields in an Index Pointer Segment

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 “Concatenated Key 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. (This topic is discussed under “Sharing Secondary Index Databases” on page 132.) 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.

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 be possible to form a unique key using source data in the subsequence field. Therefore, you can use system related fields, explained later in the chapter, 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. (This topic is discussed under "Processing a Secondary Index as a Separate Database" on page 131.) 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.

Concatenated Key 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 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. This is 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, 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. (This topic is discussed under "Processing a Secondary Index as a Separate Database" on page 131.) 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.

Making Keys Unique Using System Related Fields

You have already seen why it is desirable to have unique keys in the secondary index. You have also seen one way to force unique keys using the subsequence field in the pointer segment. If use of the subsequence field to contain additional information from the source segment does not work for you, there are two other ways to force unique keys. Both are done using an operand in the FIELD statement of the source segment in the DBD. The FIELD statement defines fields within a segment type.

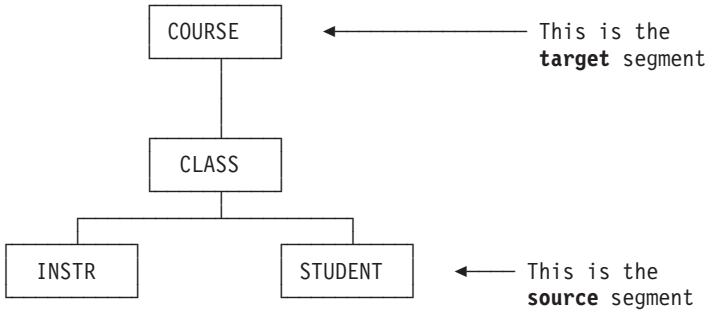
Using the /SX Operand: For HD databases, you can code a FIELD statement with a NAME field that starts with /SX. The /SX can be followed by any additional characters (up to five) that you need. When this is coded, the system generates (during segment insertion) the RBA of the source segment, which is always unique, and puts it in the subsequence field in the pointer segment. This makes the key 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.

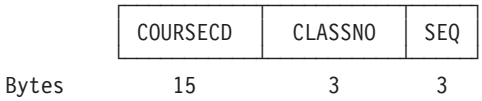
Using the /CK Operand: The other way to force unique keys is to code a FIELD statement with a NAME parameter that starts with /CK. When used as a subsequence field, /CK ensures unique keys for pointer segments. This can be done for HISAM, HDAM, or HIDAM databases. The /CK can be followed by up to five 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 following database record:



And the concatenated key of the STUDENT segment is as follows:



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).

Suppressing Index Entries (Sparse Indexing)

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 you do not need certain pointer segments built. An example of why you would not need a pointer segment built is addressed in the following scenario.

Suppose you have a secondary index for the education database at which you have been previously looking. 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.

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.

How to Specify a Sparse Index: Sparse indexing can be specified in two ways:

- 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.
- 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/ESA Customization Guide*.

How the Secondary Index Is Maintained

When a source segment is inserted, deleted, or replaced in the database, IMS keeps the index current. IMS does this whether or not 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 it might need to be deleted. After replacement or deletion, a new pointer segment is inserted. On the other hand, the pointer segment might need no changes. 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.

Sharing Secondary Index Databases

As many as 16 secondary indexes can be put in a single index database. When more than one secondary index is in the same database, the database is called a shared index database.

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. 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 (as long as 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.

Using the INDICES= Parameter

In the PCB on a SENSEG statement, you can specify an INDICES= parameter. This parameter is used to specify a secondary index that contains search fields used to qualify SSAs for an indexed segment type. Figure 78 illustrates use of the INDICES=parameter.

The use of INDICES= does not alter the processing sequence selected for the PCB by the presence or absence of the PROCSEQ= parameter.

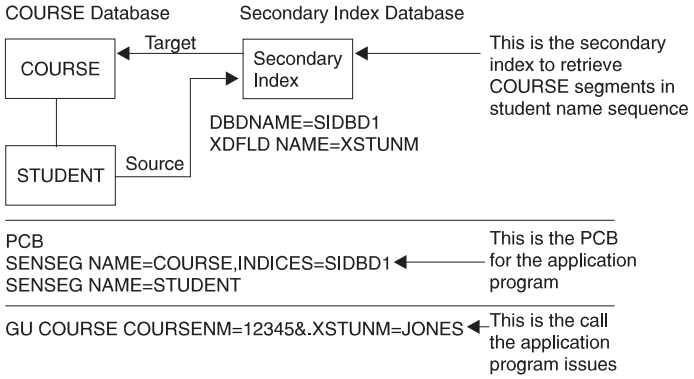


Figure 78. Use of the INDICES= Parameter (Example 1)

When the call shown in Figure 78 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.

Compare Process and Performance: Excluding COURSENM=12345 (in Figure 78 on page 133) from a GU call, impacts 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. Figure 79 on page 135 shows the use of INDICES=parameter.

In the figure, 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.

Using 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.

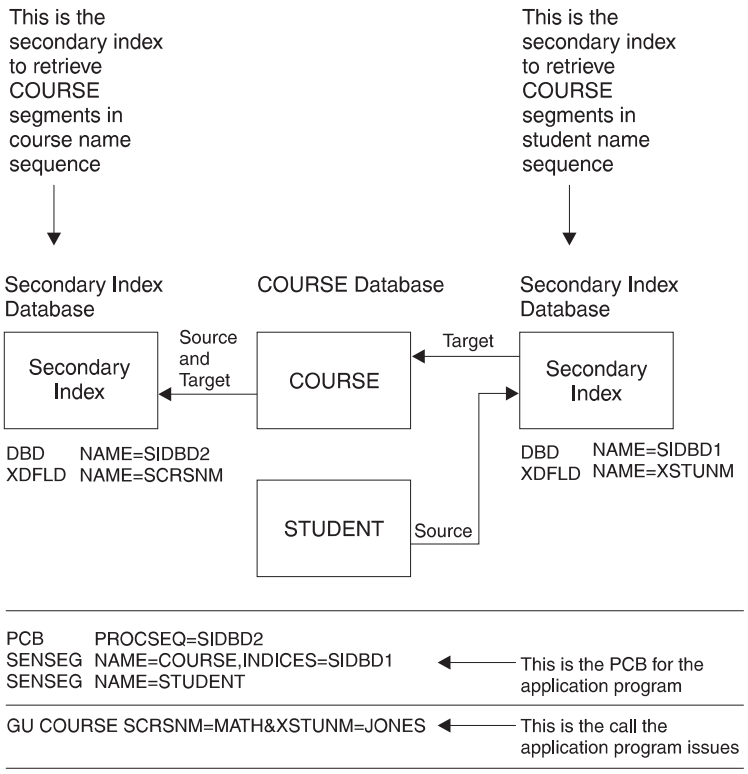


Figure 79. Use of the INDICES= Parameter (Example 2)

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 occurs.

- 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 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 steps 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 use of secondary indexing with other ways of achieving the same result. For example, to produce a report from an HDAM database in root key sequence, you can use a secondary index. However, in many cases, access to each root sequentially will be 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.

How to Specify Use of Secondary Indexing in the DBD

Figure 80 on page 138 shows the EDUC database, its secondary index, and the two DBDs required for the databases. The secondary index in this example is used to retrieve COURSE segments based on student names. The example uses direct, rather than symbolic, pointers. 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 at the bottom of the figure 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.) The following information provides a summary of how statements and parameters in the DBD in the figure are 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. The meaning of the parameters in the XDFLD statement are as follows:

  NAME= parameter.
  This parameter specifies the name that can be used in the SSA to qualify a DL/I call on the secondary processing sequence.

  SEGMENT= parameter.
  This parameter identifies the source segment, which in this example is STUDENT. If this operand is omitted, the target segment is assumed to be the same segment as the source segment. The remaining parameters in the XDFLD statement describe information related to the source segment.

  CONSTANT= parameter.
  This parameter (not used in the example) specifies the unique constant required when a secondary index is part of a shared database.

  SRCH= parameter.
  This parameter specifies the one to five fields from the source segment that are to be copied into the pointer segment's search field. In this case, only one field is being copied, the STUDNM field, which contains student names.

  SUBSEQ= parameter.
  This parameter specifies the one to five fields from the source segment that are to be copied into the pointer segment's subsequence field. These extra fields can be used to make the key in the index unique. In this case, one field is being copied, the /SX1 field, which contains a system-related field. This parameter is optional.

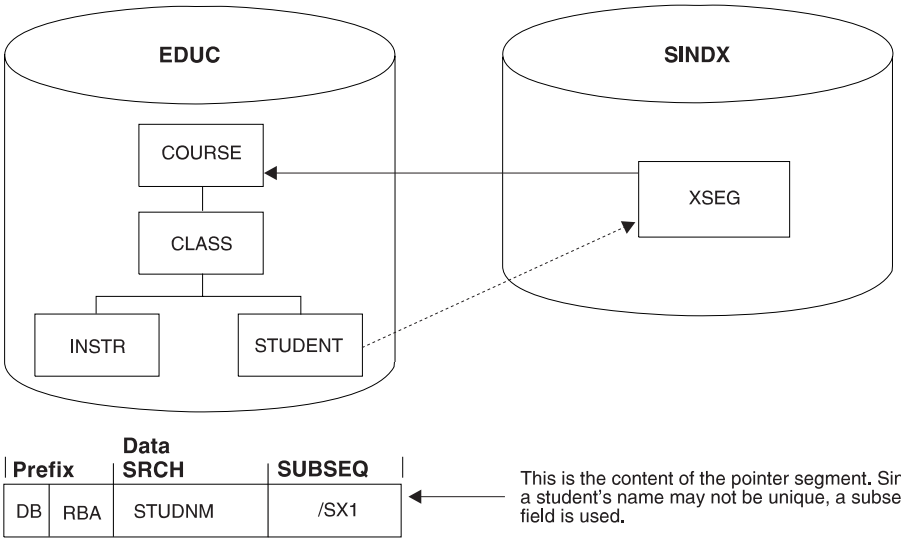
  DDATA= parameter.
  This parameter (not used in the example) specifies the one to five fields from the source segment that are to be copied into the pointer segment's duplicate data field. These fields can only be accessed when the secondary index is processed as a separate database. This parameter is optional.

  NULLVAL= parameter.
  This parameter (not used in the example) contains a 1-byte value used to suppress entries in the secondary index database. This parameter is optional.

  EXTRTN= parameter.
  This parameter (not used in the example) specifies a user-exit routine. The user routine gets control after a source segment is built. The routine is used to suppress entries in the secondary index database when you cannot use the values that can be specified in the NULLVAL= parameter. This parameter is optional.

In the example, 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. For more details, refer to "Making Keys Unique Using System-Related Fields".

Figure 80 on page 138 shows DBDs for secondary indexing.



DBD for EDUC Database

```
DBD          NAME=EDUC,ACCESS=HDAM,...
SEGM      NAME=COURSE...
FIELD      NAME=(COURSECD,...
LCHILD   NAME=(XSEG,SINDX),PTR=INDX
XDFLD     NAME=XSTUDNT,SEGMENT=STUDENT,
        SRCH=STUDNM,SUBSEQ=/SX1
SEGM      NAME=CLASS...

FIELD     NAME=(EDCTR,...
SEGM    NAME=INSTR...
FIELD     NAME=(INSTNO,...
SEGM    NAME=STUDENT...
FIELD     NAME=SEQ,...
FIELD     NAME=STUDNM,BYTES=20,START=1
FIELD     NAME=/SX1
DBDGEN
FINISH
END
```

DBD for SINDX Database

```
DBD          NAME=SINDX,ACCESS=INDEX
SEGM     NAME=XSEG,...
FIELD      NAME=(XSEG,SEQ,U),BYTES=24,
        START=1
LCHILD   NAME=(COURSE,EDUC),
        INDEX=XSTUDNT,PTR=SNGL
DBDGEN
FINISH
END
```

Figure 80. DBDs for Secondary Indexing

DBD for the SINDX Database:

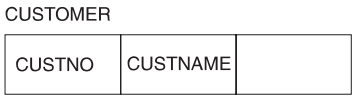
- DBD statement. The DBD statement specifies the name of the secondary index database in the NAME= parameter. The ACCESS= parameter is always ACCESS=INDEX for the secondary index DBD.
- SEGM statement. You choose what is used in the NAME= parameter. This value is used when processing the secondary index 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.

Choosing Secondary Indexes Versus Logical Relationships

While learning about secondary indexes and logical relationships, you might have noted that both options give you logical data structures. A logical data structure is a hierarchic data structure different from the data structure represented by the physical DBD. How, then, do you decide when to use a logical relationship and when to use a secondary index? This decision is based primarily on how your applications need to process the data.

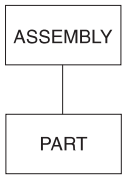
When to Use a Secondary Index

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 example, 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.



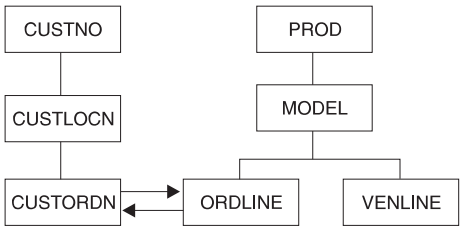
When to Use a Logical Relationship

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 that follow, 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.



The concept of a recursive structure was explained in “Establishing Logical Relationships Between Segments in the Same Database (Recursive Structures)” on page 97.

Finally, you can have application requirements that result in a segment that appears to have two parents. In the following example, 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.



Using 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. Variable-length segments can be used for HISAM, HDAM, and HIDAM databases.

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 Figure 81, 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.

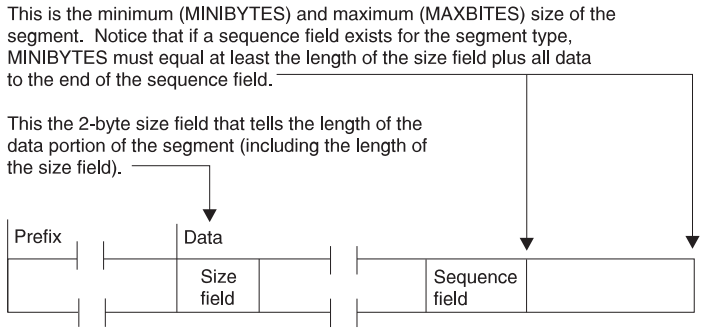


Figure 81. How Variable-Length Segments Are Specified

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.

Figure 82 shows the format of variable-length segments. The prefix and data portion of HDAM and HIDAM 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.

This is the format of a HISAM variable-length segment. It is also the format of an HDAM or HIDAM variable-length segment when the prefix and data portion of the segment have *not* been separated in storage.

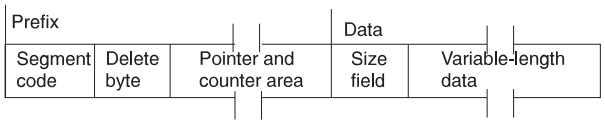


Figure 82. Format of HISAM Variable-Length Segments

This is the format of an HDAM or HIDAM variable-length segment when the prefix and data portion of the segment have been separated in storage.

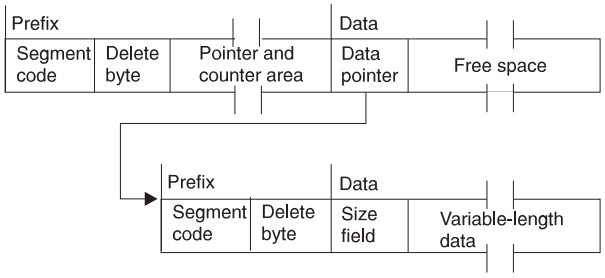


Figure 83. Format of HDAM OR HIDAM Variable-Length Segments

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 or HIDAM 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 82). 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 or HIDAM 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 or HIDAM 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. You should devise a scheme because if variable-length fields and fixed-length fields in a segment are mixed, the application program has no way of knowing where specific fields begin. One way to solve this problem is to put the size of a variable-length field at the beginning of the variable-length field. If a segment has only one variable-length field, it can be made the last field in the segment. If it is at all possible, the simplest scheme is to have only one field in a variable-length segment.

Adding or Converting to Variable-Length Segments

Information on how to add variable-length segments to an existing database and convert an entire database to variable-length segments is in “Chapter 15. Modifying Your Database” on page 365.

Using the Segment Edit/Compression Facility

Detailed information on how the segment edit/compression facility works and how you use it is in *IMS/ESA Customization Guide*. This section introduces you to the facility.

The segment edit/compression facility 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 segment edit/compression facility 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.

Two types of segment manipulation are possible using the segment edit/compression facility:

- *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.

Use of the segment edit/compression facility is specified by segment type. Any segment type can be edited or compressed (using either data or key compression) as long as the segment is:

- Variable length in the database (however, it can be defined to the application program as either fixed or variable length)
- Not a logical child or in a logical database
- Not in an HSAM, SHISAM, or index database

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

Things to Consider Before Using the Segment Edit/Compression Facility:

- You must provide storage for your edit routine for both batch and online systems. IMS adds 10 bytes to the maximum segment length when compression is specified.
- 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. See *IMS/ESA Customization Guide*for additional detail.
- Editing and compressing of each segment on its way to or from an application program requires additional processor time.

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.

How to Specify the Segment Edit/Compression Facility: Use of the segment edit/compression facility is specified by segment type on the SEGM statement in the DBD:

```
SEGM NAME=SEGNAME[,COMPRTN=(ROUTINE NAME[,DATA][,INIT])]
```

where:

- The routine name is the name of your edit routine.
- DATA specifies data compression. DATA is the default.
- KEY specifies key compression. Specifying KEY on segments in a DEDB is not allowed and causes DBDGEN to fail.
- INIT specifies that your edit routine gets control when the database is opened and closed. When specified on DEDB segments, INIT causes your edit routine to get control after the first area in the database is opened and before the last area in the database is closed. If INIT is not coded, your edit routine gets control when a DL/I call processes a segment for which segment edit / compression is specified. To do other edit processing, such as processing on the table appended to the DBD, use the edit routine to get control when the database is opened.

Note:

1. The COMPRTN= keyword is prohibited on DEDB segments containing a unique key field located at the end of the segment. If COMPRTN= is attempted against these types of segments, DBDGEN fails and message DGEN440 is issued. See *IMS/ESA Messages and Codes* for a description of this message.
2. Your routine must not modify or alter the relative position of a key field in a DEDB segment. If the key field in a DEDB segment changes or moves during a compress or expand call, abend 799, subcode 1 is issued. See *IMS/ESA Messages and Codes* for a description of this abend.
3. The Database Scan utility returns only *compressed* segments when run on SDEPs (sequential dependents) with the segment edit/compression facility applied.

Converting to the Segment Edit/Compression Facility: Information on how to convert an existing database so it can use the segment edit / compression facility is discussed in “Chapter 15. Modifying Your Database” on page 365.

Using Data Capture Exit Routines

This section contains general-use programming interface information.

The Data Capture exit routine is an installation-written exit routine. Data Capture exit routines promote and enhance database coexistence. Data Capture exit routines capture segment-level data from a DL/I database for propagation to DB2 databases. Installations running IMS and DB2 databases can use Data Capture exit routines to exchange data across the two database types.

Data Capture exit routines can be written in assembler language, C, VS COBOL II, or PL/I. *IMS/ESA Customization Guide* describes Data Capture exit routines in detail.

Data Capture exit routines are supported by IMS Transaction Manager and Database Manager. DBCTL support is for BMPs only.

Data Capture exit routines are compatible with the following physical database structures:

HDAM

HIDAM

HISAM

SHISAM

DEDB

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.

DBD Parameters for Data Capture Exit Routines: This section contains programming interface information.

Using Data Capture exit routines requires specification of one or two DBD parameters and subsequent DBDGEN. 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 a Data Capture exit routine, you must use the optional EXIT= parameter. 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))
```

exit-name is the only required operand for EXIT=. The Data Capture exit routine, exit-name, allows you to run against segments in a database. The routine, exit-name, 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 might be coded:

```
SEGM   EXIT=(EXITA,KEY,DATA,NOPATH,(CASCADE,KEY,DATA,NOPATH))
```

KEY, NOPATH, DATA, CASCADE, KEY, DATA, and NOPATH are default operands. These defaults define what data is captured by the exit routine when a segment is updated by an application program. For a full description of the default operands and other optional operands, see *IMS/ESA Utilities Reference: Database Manager*.

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.

Call Sequence of Data Capture Exit Routines: This section contains programming interface information.

A Data Capture exit routine is invoked once per segment update for each segment with that Data Capture exit routine specified. Data Capture exit routines are invoked multiple times for a single call under certain conditions. These conditions 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 Capture Exit Routine: This section contains programming interface information.

Data is passed to Data Capture exit routines when an application program updates IMS with a DL/I insert, delete, or replace call. 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 with compression/edit routines, the data passed is expanded data.

When an application replaces a segment, both before and after physical data is 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 logical child/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.

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.

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, it is recommended that the SQL update request not be issued until after the before and after replace data for those fields is compared to see if the fields were changed.

Data Capture Call Functions: This section contains programming interface information.

Data capture exit routines are called when segment data is updated by an application program insert, replace, or delete call. Optionally, Data Capture exit routines are called when DL/I deletes a dependent segment because the application program deleted its parent segment, a process known as cascade delete. Data Capture exit routines are passed to two functions to identify the following:

1. The action performed by the application program and
2. The action performed by IMS:
   - **Call function.** The DL/I call function issued by the application program for the segment: ISRT, REPL, or DLET.
   - **Physical function.** The physical action, ISRT, REPL, or DLET, performed by IMS as a result of the call. The physical function is used to determine the type of SQL request to issue when doing data propagation.

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 passed is ISRT and the physical function 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: This section contains programming interface information.

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 in order to issue the SQL delete for the parent of the physical structure in DB2. 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: This section contains programming interface information.

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. Table 4 summarizes which delete rules you can and cannot use in logically related databases with Data Capture exit routines specified on their segments.

Table 4. 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.

Converting to Data Capture Exit Routines: This section contains programming interface information.

Information on how to convert an existing database for Data Capture exit routines is discussed in “Converting Databases for Data Capture Exit Routines and Asynchronous Data Capture” on page 389. See *IMS/ESA Utilities Reference: Database Manager*for detailed information on coding the EXIT= and VERSION= parameters.

Using Field-Level Sensitivity

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.

(SENFLD statements are described later, but basically they determine the order of fields in a segment as seen by an application program.)

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 doesn't 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 MVS 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.

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 has been defined in the segment through the DBDGEN utility.

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 rightmost 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.

Figure 84 is an example of field-level sensitivity. Following 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.

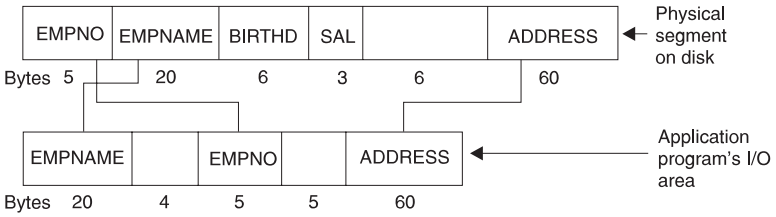


Figure 84. DBD and PSB Coding for Field-Level Sensitivity

This is the DBD for the figure shown 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
```

This is the PSB for the figure shown above:

```
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.

Figure 85 shows an example of a retrieve call based on the DBD and PSB in Figure 84.

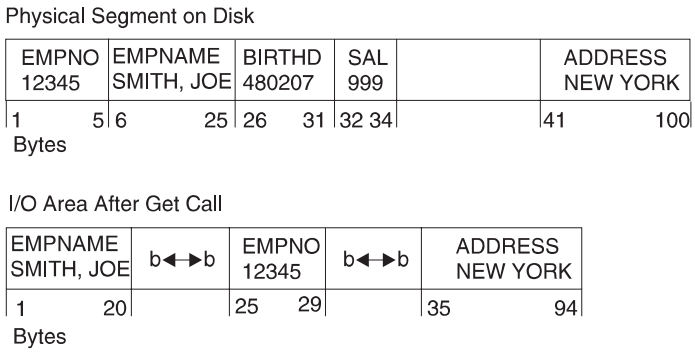


Figure 85. Example of a Retrieve Call

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 Figure 84 on page 150, 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. Figure 86 on page 152 shows an example of a REPL call based on the DBD and PSB in Figure 84.

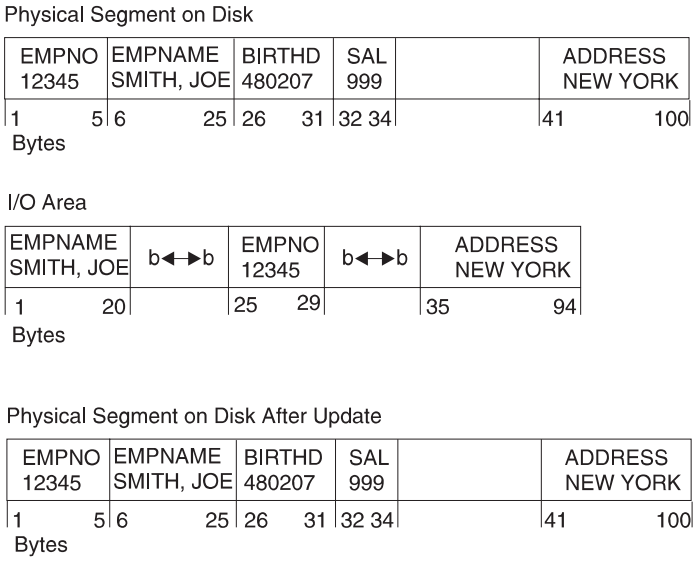


Figure 86. Example of a REPL Call

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

Figure 87 shows an example of an insert operation based on the DBD and PCB in Figure 84 on page 150.

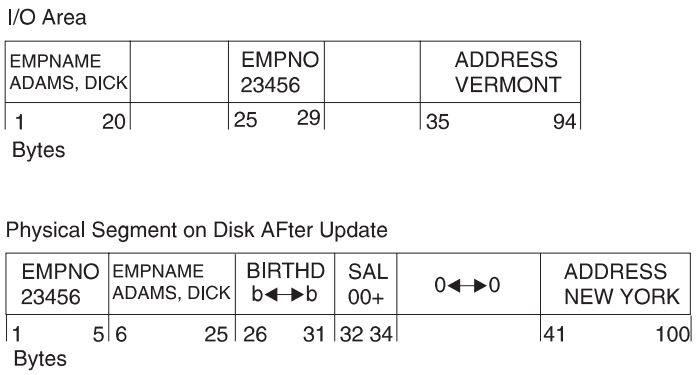


Figure 87. Example of an ISRT Call

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, 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: Here are the rules you must follow when using field-level sensitivity with segments involved in a logical relationship:

- 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. The following section addresses 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.

Performance Considerations for Logical Relationships

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 section 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. Figure 88 is an example of a missing field on a retrieve call based on the DBD and PSB in Figure 89.

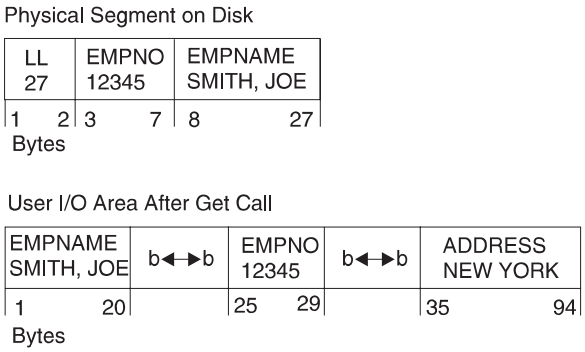


Figure 88. Example of a Missing Field on a Retrieve Call

```
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

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
```

Figure 89. DBD and PSB When Using Field-Level Sensitivity with Variable-Length Segments

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.

Replacing Missing Fields: A missing field that is not replaced does not affect the physical variable-length segment. Figure 90 is an example of a missing field on a replace call based on the DBD and PSB in Figure 89.

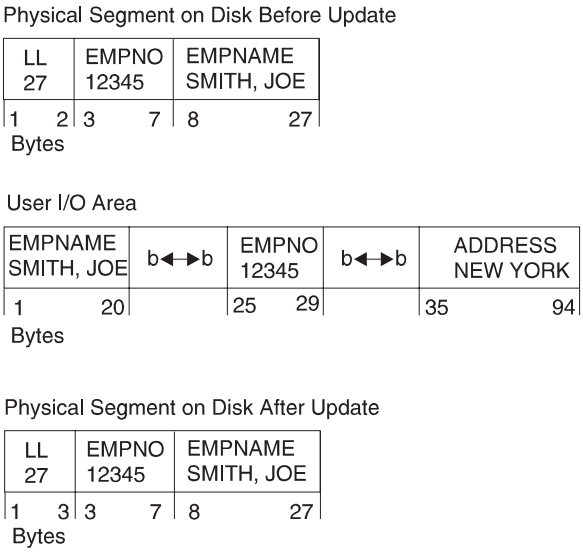


Figure 90. First Example of a Missing Field on a Replace Call

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 (see Figure 89 on page 154) 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.

Figure 91 is an example of a missing field on a replace call based on the DBD and PSB in Figure 89 on page 154.

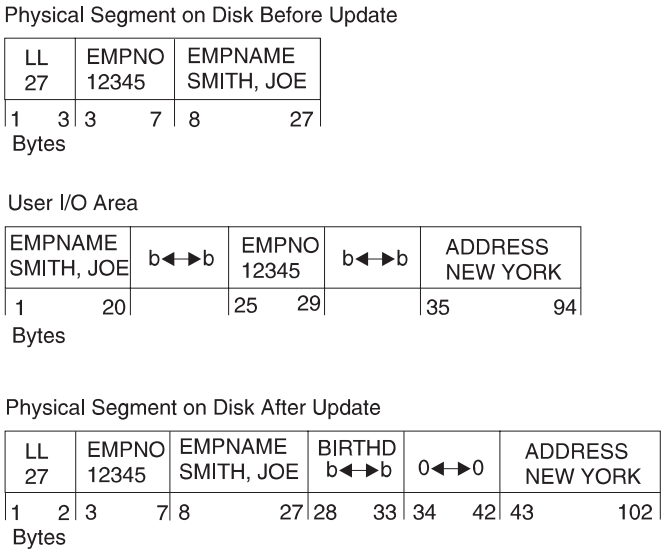


Figure 91. Second Example of a Missing Field on a Replace Call

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.

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 Figure 89 on page 154) 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.

Figure 92 is an example of a missing field on an insert call using the DBD and PSB in Figure 89 on page 154.

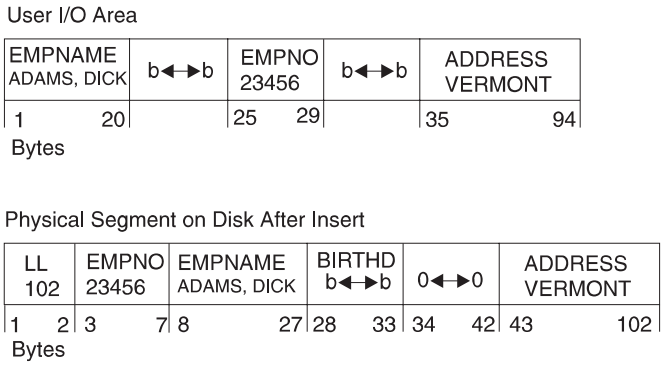


Figure 92. Example of a Missing Field on an Insert Call

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. Figure 93 is an example of a partially present field on a retrieval call based on the DBD and PSB in Figure 89 on page 154.

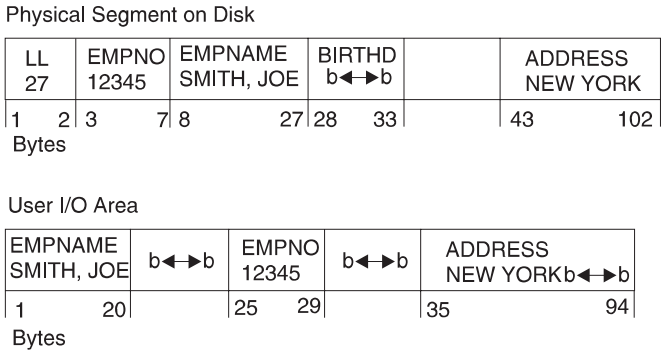


Figure 93. Example of a Partially Present Field on a Retrieval Call

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: You should know the following information about replacing partially present fields:

- 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.

Figure 94 on page 158 is an example of a partially present field on a REPL call based on the DBD and PSB in Figure 89 on page 154.

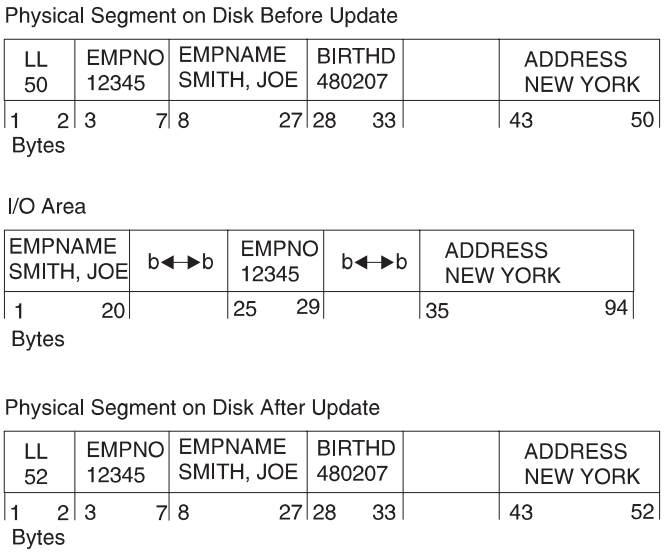


Figure 94. Example of a Partially Present Field on a REPL Call

The 'LL' field is changed from 50 to 52 by DL/I to accommodate the change in the field length of ADDRESS.

General Considerations for Using Field-Level Sensitivity:

- 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.

Using Multiple Data Set Groups

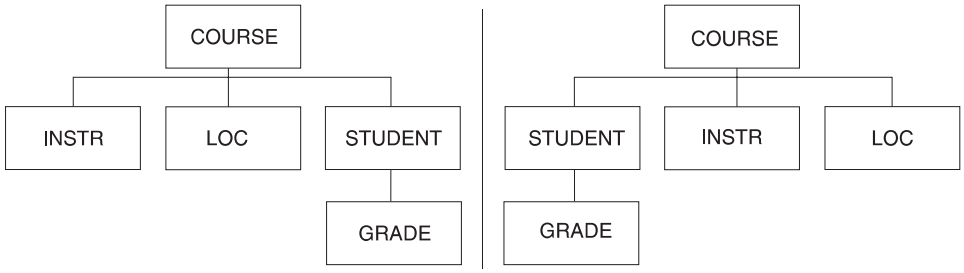
Although this book has explored storing a database on a single or a single pair of data sets, HD databases can be stored on more than the one or two data sets required for database storage. You have seen that an HD database is stored on an ESDS, if VSAM is being used, or an OSAM data set, if OSAM is being used.

HD databases can be stored on multiple data sets. 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.

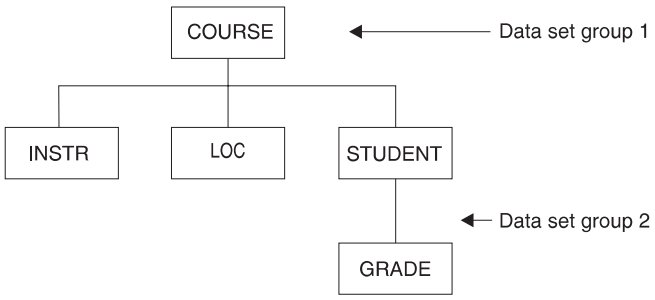
As many as ten data set groups can be used in HISAM and HD databases, that is, one primary data set group and a maximum of nine secondary data set groups.

Why 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 hierarchic 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 following database records. Both of them contain the same segments, but the hierarchic sequence of segments is different.



The hierarchy on the left favors applications that need to access INSTR and LOC segments. The hierarchy on the right 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 do this, you would use two data set groups. As shown in the following figure, the first data set group contains the COURSE, INSTR, and LOC segments. The second data set group contains the STUDENT and GRADE segments.



Other uses of multiple data set groups include:

- Separating infrequently-used segments from high-use segments.

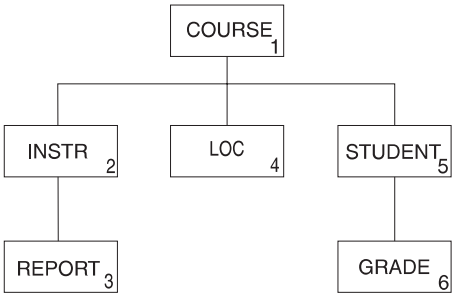
- Separating segments that frequently have information added to them from those that do not. For the former segments, you might specify additional free space so conditions are optimum for additions.
- Separating segments that are added or deleted frequently from those that are not. This can keep space 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 bit map 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 bit map does not track it.

HD Databases Using Multiple Data Set Groups: The following rules must be followed when using a multiple data set group in an HD database:

- As many as ten data set groups can be defined.
- The root segment in a database record must be in the primary data set group.

In the database record shown below, segments 1, 2, 4, and 5 could go in one data set group, while segments 3 and 6 could go in a second data set group. Other examples of how this HD database record could be divided are as follows (list not exhaustive):

Primary Data Set Groups	Secondary Data Set #1 Groups	Secondary Data Set #2 Groups
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 separated into different data set groups must be connected by physical child first pointers. For example, in Figure 95 on page 161 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.

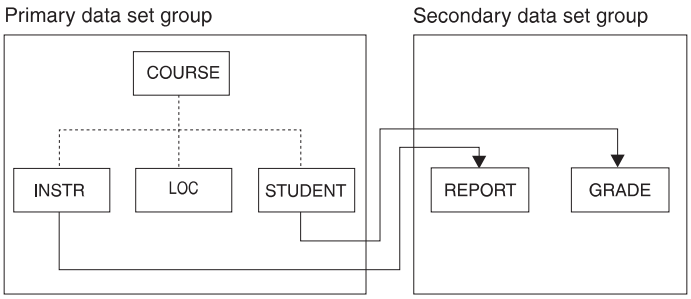


Figure 95. Connecting Segments in Multiple Data Set Groups Using Physical Child First Pointers

How HD Records Are Stored in Multiple Data Set Groups: Now that you have seen *what* segments can be stored in a single data set group in an HD database, this section looks at *how* segments are stored. Figure 96 on page 162 shows one database record:

- Stored in an HDAM database using two data set groups
- Stored in a HIDAM database using two data set groups

Specify in the DBD which segment types need to be put in a data set group. Based on that information, IMS automatically loads segments into the correct data set group. 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 database, note that only the primary data set group has a root addressable area. The secondary data set group is additional overflow storage.

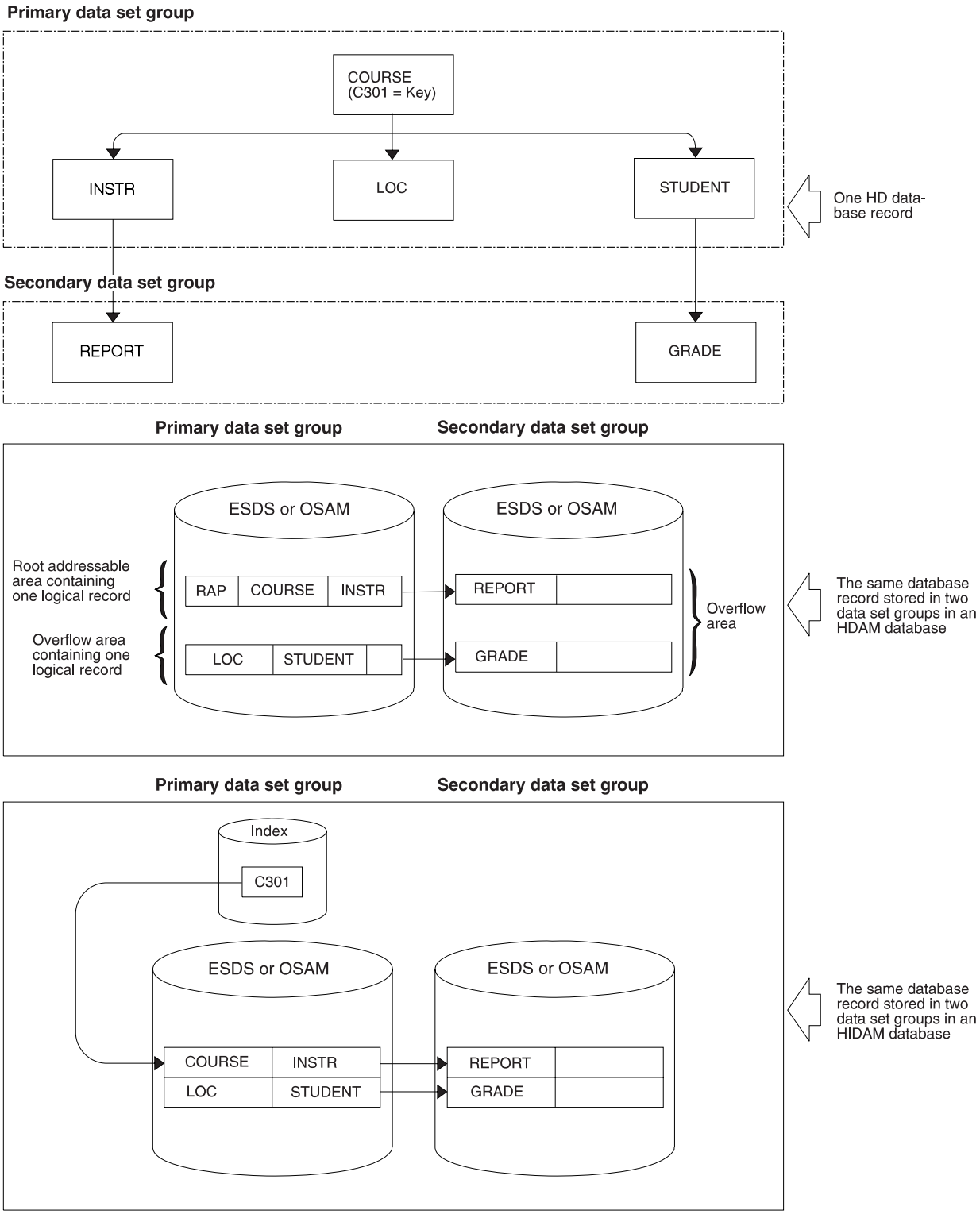


Figure 96. HD Database Record in Storage When Multiple Data Set Groups Are Used

Specifying Use of Multiple Data Set Groups in HD Databases: Use of multiple data set groups is specified to IMS in the DBD. Figure 97 shows how the DBD is coded for the database record at which you have been looking. In this example, the DBD is for an HDAM database.

This database record needs to be put in two data set groups, one containing segments 1 and 2; the other containing segments 3, 4, 5, and 6.

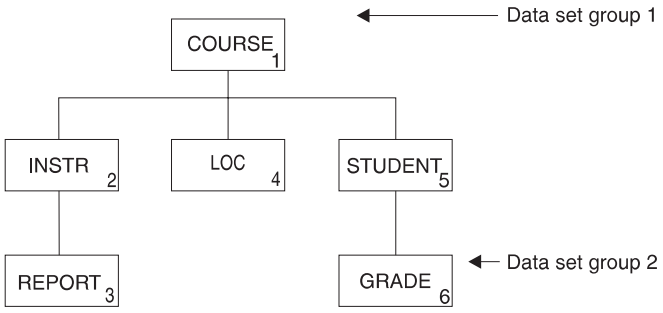


Figure 97. DBD For Multiple Data Set Groups in HDAM

Here is how these two data set groups are specified in the DBD. Segments are arranged in correct hierarchic sequence. Data set labels (DS1 and DS2) are used to keep segment types in the correct data set group.

```
DBD    NAME=HDMDSG,ACCESS=HDAM,RMNAME=(ROUT001)
DS1    DATASET  DD1=DS1DD,DEVICE=3330
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))
DS2    DATASET  DD1DS2DD,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
```

Note in Figure 97 that the DBD has two DATASET statements. The DATASET statements are followed by the appropriate SEGM statements arranged with segments in correct hierarchic sequence. In each DATASET statement, the DD1= parameter names the VSAM ESDS or OSAM data set that will be used. Also note that each data set group can have its own characteristics, such as device type (DEVICE= parameter).

When multiple data set groups are specified, segment types must be arranged in the DBD in correct hierarchic sequence. Figure 98 shows this concept. Segments 1 through 6 are put in the DBD in correct hierarchic sequence. The DATASET statement is used to keep segments in the correct data set group. In the figure, the third DATASET statement says put segment 6 in the same data set group as segments 1, 2, and 3.

This database record needs to be put in two data set groups, one containing segments 1, 2, 3, and 6; the other containing segments 4 and 5.

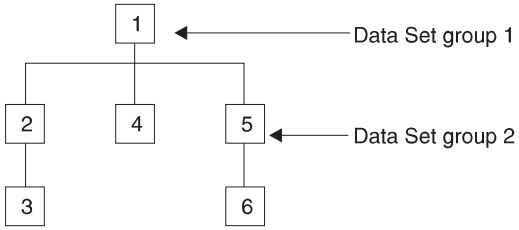


Figure 98. Use of the Data Set Label in the DBD When Specifying Multiple Data Set Groups

Here is how these two data set groups are specified in the DBD. Segments are arranged in correct hierarchic sequence. Data set labels (DS1 and DS2) are used to keep segment types in the correct data set group.

```
          DBD
DS1       DATASET . . .
          SEGM 1
          SEGM 2
          SEGM 3

DS2       DATASET . . .
          SEGM 4
          SEGM 5

DS1       DATASET  <-- This repetition of a data set label (DS1)
          SEGM 6       says to put the segments that follow this
          DBDGEN       statement in the data set (and, therefore, the
                       data set group) that first used this label.
                       In other words, store segment 6 with
                       segments 1, 2, and 3.
```

Chapter 6. Database Design Considerations for Full Function

About This Chapter

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

This chapter examines the following database design considerations:

- Specifying free space
- Estimating the size of root addressable areas
- Determining which randomizing modules to use
- Choosing HDAM options
- Choosing logical record length for HISAM and HD databases
- Determining size of CIs and blocks
- Determining which VSAM and OSAM options to use
- Deciding which FIELD statements to code into the DBD
- Planning for maintenance
- Using aids to help design your database

Specifying Free Space (HDAM and HIDAM Only)

As you have seen, dependent segments inserted after an HD database is loaded are put as close as possible to the segments to which they are related. (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.) 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.

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 databases. If you specify fbff for HDAM 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 macro 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.

  For details on the HD Space Search Algorithm, see “How the HD Space Search Algorithm Works” on page 78. For more information on the SEARCHA keyword, see “Database Description (DBD) Generation” in *IMS/ESA Utilities Reference: Database Manager*.
- 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 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 or HDAM. (This command is explained in detail in *MVS/DFP Access Method Services for VSAM Catalog*·)

Estimating the Size of the Root Addressable Area (HDAM Only)

To estimate the size of the root addressable area, use the following formula:

$$\frac{A \times B}{C} = D$$

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.

Sizing 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:

$$\frac{D \times E \times G}{F} = H$$

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 Only)

As you have seen, a randomizing module is required to store and access HDAM database records. This 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 root segments. An HDAM database uses only one randomizing module, but several databases 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. These modules, and the arithmetic techniques they use, are described in detail in *IMS/ESA Customization Guide*.

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 bit map locations, and keep packing density high. *IMS/ESA Customization Guide* tells you what the interface to your randomizing module should be.

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 System Utilities/Database Tools (DBT) HD tuning aid utility. This utility 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 DBT HD tuning aid utility 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.

"Adjusting HDAM Options" in "Chapter 14. Tuning Your Database" on page 323 discusses how you can adjust HDAM options to tune your database once it is running. One of these options is the randomizing module. Read that section *before* choosing a randomizing module.

The name of your randomizing module is specified in the RMNAME=mod operand in the DBD statement in the DBD.

Choosing HDAM Options

In an HDAM database, the options you choose can greatly affect performance. The options discussed here are those you specify in the RMNAME= keyword in the DBD statement:

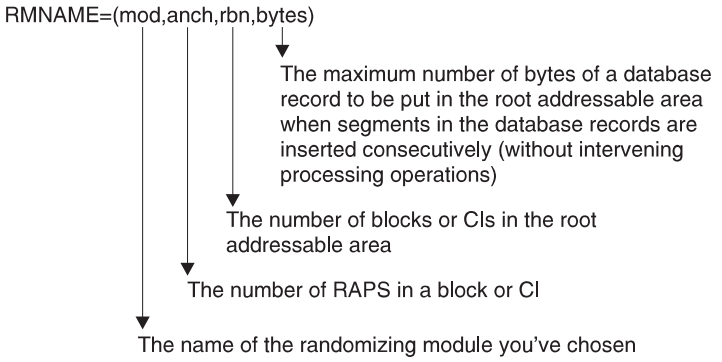


Figure 99. Specifying the RNAME keyword

Minimizing I/O Operations

In choosing these HDAM 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 (discussed in the preceding section) 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 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:

```
                      Number of roots  x  root bytes
Packing density   =   ____________________________________________

                      Number of CI or         Usable space
                      blocks in the root  x   in the CI or block
                      addressable area
```

where:

- Root bytes = the average number of bytes per root in the root addressable area
- Usable space in the CI or block = 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.

IMS System Utilities/Database Tools (DBT) has a tuning aid utility that helps you choose HDAM options. This utility uses a file of your root keys as input. One of the reports it produces shows the number of roots (synonyms) chained from a single RAP. This report is based on your randomizing module, the number of blocks or CIs in the root addressable area, the number of RAPs in a block or CI, and the access method you are using (VSAM or OSAM). Another report produced by the utility uses the same input file of root keys to show how many RAPs in the root addressable area are not used.

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. Figure 100 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.

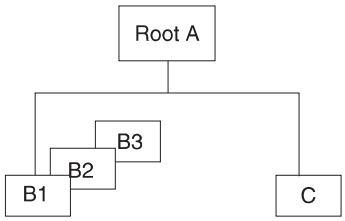


Figure 100. Why Short or Long Logical Records Increase Wasted Space #1

The database record shown above is stored on three short logical records. Note the three areas of unused space.

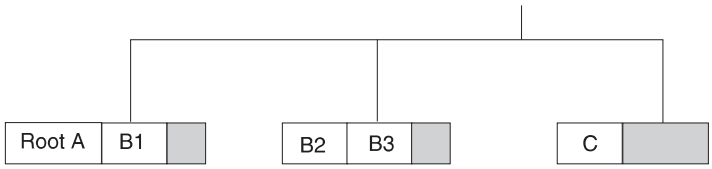


Figure 101. Why Short or Long Logical Records Increase Wasted Space #2

The same database record is stored on two longer logical records. Now there are only two areas of unused space, rather than three, but the total size of the areas is larger.

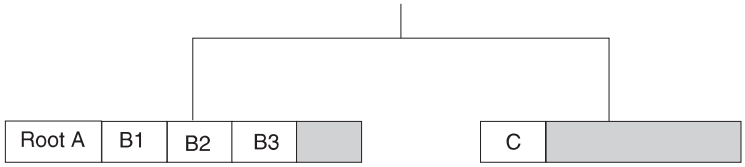


Figure 102. Why Short or Long Logical Records Increase Wasted Space #3

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 103, each CI contains two logical records, and two CIs are required to contain the database record. Consequently, it takes two read operations to get these four logical records into the buffer.

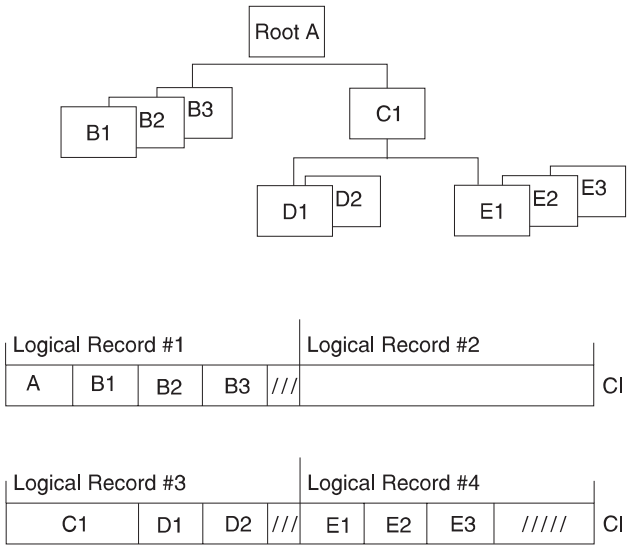


Figure 103. Two Read Operations to Get Four Logical Records

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). To determine the average size of your database records, see “Estimating the Minimum Size of the Database” on page 286 in “Chapter 12. Loading Your Database” on page 285.

Specifying Logical Record Length

Specify the length of the logical records in the RECORD= operand of the DATASET statement in the DBD.

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). See the next section (“Determining the Size of CIs and Blocks”) for information on determining CI or block size.

As with HISAM databases, specify the length of the logical records in the RECORD= operand of the DATASET statement in the DBD.

Determining the Size of CIs and Blocks

You can specify the DEDB CI resource size for your database. (If you do not specify it, the DBDGEN utility will calculate it for you.) 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 *MVS/DFP Access Method Services for VSAM Catalog*.

Choosing 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 following sections can help you with these decisions.

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.

"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. This option is discussed under "Determining Which VSAM Options to

Use" on page 184. Otherwise, you control and reduce wait time by carefully choosing of the number and size of buffers.

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. For a brief explanation of how to specify hiperspace buffering, see "Hiperspace Buffering Parameters" on page 354.

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.

Buffer Numbers

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. Monitoring is discussed in "Chapter 13. Monitoring Your Database" on page 309.

Buffer size and number of buffers are specified when the system is initialized. Both can be changed (tuned) for optimum performance at any time. Tuning is discussed in "Chapter 14. Tuning Your Database" on page 323.

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

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

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.

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 DC 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)
/*
```

Detailed information on how to code these control statements is located in the *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

OSAM buffers can be fixed in storage using the IOBF= parameter. In VSAM, buffers are fixed using the VSAMFIX= parameter in the OPTIONS statement. This parameter is described under “Determining Which VSAM Options to Use” on page 184. 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. The format of this parameter is:

```
IOBF= (length,number,fix1,fix2,id)
```

where:

- *length* is the size of buffers in a subpool.

- *number* is 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.
- *fix1* is whether the buffers and buffer prefixes in this subpool need to be fixed and is specified as Y or N (yes or no).
- *fix2* is whether the buffer prefixes in this subpool and the subpool header need to be fixed and is specified as Y or N (yes or no).

  The default for the fix1 parameter is that buffers and their prefixes are not fixed. The default for the fix2 parameter is that buffer prefixes and the subpool header are not fixed.
- *id* is a parameter that specifies an identifier to be assigned to the subpool. It 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, as described in *IMS/ESA Installation Volume 1: Installation and Verification*.) The id parameter 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

Using 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.

About SB

As you learned in the previous section, the normal OSAM buffering method reads only one block with each I/O operation. This is known as a *random read*. 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 device 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. This is called a *sequential read*. 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 Using SB

By using 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 and utilities that might benefit from the use of SB are:

- IMS batch programs that sequentially process your databases.
- BMPs that sequentially process your databases.
- Generalized software packages that sequentially process your databases, for example, Data Extract Facility (DXT)
- IMS utilities, including:
  - Online Database Image Copy
  - HD Reorganization Unload
  - Partial Database Reorganization
  - Surveyor
  - Database Scan
  - Database Prefix Update
  - Batch Backout
- Those few 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.

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.

Using OSAM Sequential Buffering

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.

These methods of controlling the use of SB are discussed in the section “How to Request the Use of SB” on page 181.

How SB Buffers Data

The next few sections describe what happens when you request SB. You will learn what SB buffers, how and when SB is activated, and what happens to the data that SB buffers.

What SB Buffers

As discussed in “Chapter 5. Choosing Additional Database Functions” on page 83, 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 referenced by a specific database PCB is referred to in the following discussion 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.

Conditional Activation and Periodical Evaluation 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 used by the program. After awhile, IMS performs the first of a series of *periodical evaluations* of the buffering process. IMS performs these periodic evaluation for each DB-PCB/DSB pair. This periodical evaluation determines if the use of SB would be beneficial for the DB-PCB/DSG pair. If the use of SB would be 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:

- 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 has been 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 periodical 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. (SB buffers are also discussed in the section “Virtual Storage Considerations for SB”.) 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 called the SB buffer handler to hold the sets of 10 consecutive blocks 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 processing a request from an online program, the SB buffer handler only searches the SB buffer pools allocated to that online program.

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 periodical 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. For details on the SB Initialization User Exit Routine see *IMS/ESA Customization Guide*.

How to Request the Use of SB

IMS provides two methods for specifying which of your programs and databases should use SB.

1. You can explicitly specify which programs and utilities should use SB. During PSBGEN or by using SB control statements.
2. 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 MVS/ESA environment, use the first method. If you are running IMS in a non storage-constrained MVS/ESA environment, use the second method.

Requesting SB During PSBGEN

You can code the SB= keyword in the PCB macro instruction of your application's PSB. (This is not possible for IMS utilities that do not use a PSB during execution.) You code this keyword for each database PCB buffered with SB.

The format of the SB= keyword is:

```
PCB     TYPE=DB,...,SB={COND/NO}
```

where:

COND This option specifies that SB should be conditionally activated for this PCB.

NO This option 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. This option is discussed in the next section.

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
```

Detailed instructions for coding PSB statements are contained in *IMS/ESA Utilities Reference: System*.

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. (For details on the SB buffer pools see “Virtual Storage Considerations for SB” on page 181.) 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. For details on these reports, see *IMS/ESA Utilities Reference: System*. For information on tuning the number of buffer sets, see “Chapter 14. Tuning Your Database” on page 323.

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
```

Detailed instructions for coding the SBPARM control statement are contained in *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

Requesting SB with an SB Initialization Exit Routine

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.RESLIB. An SB exit routine allows you to dynamically control the use of SB at application scheduling time.

IMS supplies five sample SB exit routines in IMS.DBSOURCE and IMS.RESLIB. 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 the *IMS/ESA Customization Guide* .

SB Options or Parameters Provided by Several Sources

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 nth control statement overrides the mth 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/ESA Installation Volume 2: System Definition and Tailoring*.

Disallowing the Use of SB

This section describes how 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. For details on these commands see *IMS/ESA Operator's Reference*.
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.

Determining Which VSAM Options to Use

Several types of options can be chosen for databases using VSAM. Specifying options such as free space for the ESDS data set, logical record size, and CI size are discussed in preceding sections of this chapter. This section describes these optional functions:

1. Functions specified in the OPTIONS control statement when IMS is initialized.
2. Functions specified in the POOLID, VSRBF, and DBD control statements when IMS is initialized.
3. 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 section describes each option and the implications of using it.

The OPTIONS statement is described in detail in the *IMS/ESA Installation Volume 2: System Definition and Tailoring*. The OPTIONS statement and all its parameters are optional.

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. (The DEFINE CLUSTER command is explained in the following section, “Specifying Free Space for a KSDS (FREESPACE Parameter)” on page 188. 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.

OFF is the default for all the traces. 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. For documentation on the trace parameters, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

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/or 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.

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

POOLID is the required control statement for defining multiple VSAM shared resource pools. Each POOLID statement defines one shared resource pool. The format of the POOLID control statement is:

```
POOLID=id,FIXDATA=YES | NO,FIXINDEX=YES | NO,FIXBLOCK=YES | NO,STRINGNM=n
```

The only required parameter for POOLID is id. The id parameter is coded as a user-defined, one-to-four character alphanumeric field, and is used in conjunction with the DBD subpool definition statement to assign a given data set to a specific shared resource pool. The id parameter assigned to a POOLID must match the id parameter on the DBD subpool definition statement. The format of the DBD statement is:

```
DBD=dbdname(data-set-number,id,ERASE=YES | NO,FREESPACE=NO | YES)
```

The dbdname is a name specified on the DBDGEN DBD macro statement NAME=. The data-set-number is an IMS-assigned number that identifies a specific data set of a data set group within a database. The data-set-number is the link that assigns a specific shared pool and its subpools to a specific data set.

POOLID is followed by one or more VSRBF subpool definition statements. Each VSRBF statement defines a subpool within that shared pool. The format of the VSRBF statement is:

```
VSRBF=buffer-size,number-of-buffers,type,HSO | HSR,HSn
```

In the VSRBF statement, type specifies the type of shared resource subpool to create, data subpool (D) or index subpool (I). You can create data subpools without creating index subpools, but you cannot create index subpools without corresponding data subpools. If the parameter is invalid or not coded, data (D) is the default.

You can use VSRBF statements *without* the POOLID shared pool definition statement to define subpools within a single default shared pool. If VSRBF is used without the POOLID statement, you can still define separate data and index subpools. Implementing the POOLID, VSRBF, and DBD control statements and their corresponding parameters is described in detail in *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

Optional Functions Specified in the Access Method Services DEFINE CLUSTER Command

There are several optional functions that affect performance that can be chosen when you define your VSAM data sets. These functions are specified in the Access Method Services `DEFINE CLUSTER` command. This command and all its parameters are described in detail in *MVS/DFP Access Method Services for VSAM Catalog*.

Specifying that 'Fuzzy' Image Copies Can be Taken with the Database Image Copy 2 (DFSUDMT0)

To establish that 'fuzzy' image copies of KSDSs can be taken with the Database Image Copy 2 (DFSUDMT0), specify the BWO(TYPEIMS) parameter. For this option to take effect the following conditions must exist:

- The KSDS must be SMS-managed.
- All access to the KSDS, once this option is specified, is done under DFSMS 1.3 or later version (once the KSDS has been opened under DFSMS 1.3, attempts to open it under an earlier version will fail).

Specifying Free Space for a KSDS (FREESPACE Parameter)

Get free space in a CI in a KSDS is by specifying it in the FREESPACE parameter in the `DEFINE CLUSTER` command. Free space for a KSDS can *not* 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. INSERT=SEQ is explained in a previous section called "Choosing an Insert Strategy (INSERT Parameter)".

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.

Specifying Whether Data Set Space Is Pre-formatted for Initial Load (SPEED | RECOVERY Parameter)

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/ESA Utilities Reference: Database Manager*.

RECOVERY is the default for this parameter.

Specifying Whether Sequence Set Records Are Embedded and Index Set Records Are Replicated

A VSAM KSDS cluster has a data component (where segments are stored in HISAM and HIDAM databases) and an index component (called the *VSAM index* in this discussion.) The VSAM index contains pointers to CIs in the KSDS data component. When a specific key in a KSDS is requested, the VSAM index is used to limit the search for the CI that contains the correct root segment. Without the VSAM index, the entire KSDS data component could be searched to find the correct CI. The VSAM index can be on either the same volume as the data component or on another volume. It is the VSAM index whose options are of concern here. You need to know some things about the VSAM index before the options are described.

The VSAM index consists of one or more levels, as shown in Figure 104. The first (lowest) level is called the sequence set level. All other levels are called index set levels. The sequence set level has a sequence set record for each CA in the database. Each sequence set record contains a pointer to each CI in a specific CA and the highest root segment's key in that CI.
Index set records on the first index set level contain pointers to sequence set

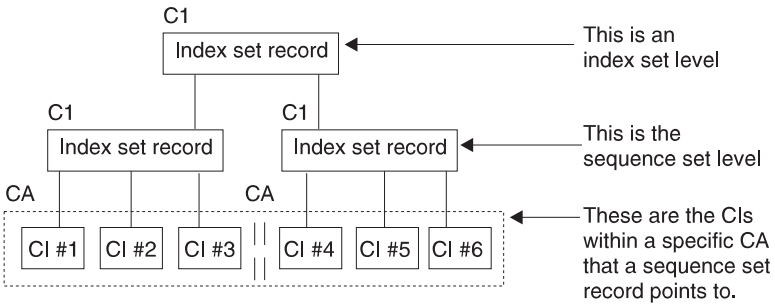


Figure 104. Levels in a VSAM Index

records. Each pointer on the first index set level contains the address of a sequence set record and the highest root segment key in the sequence set record pointed to.

If no more room exists for new pointers in an index set record, a new index set record is started on the same level. As soon as there are two index set records on a level, a new index set record is started on the next higher level.

At the second and higher levels of the index set, the pointers are to index set records at the next lowest level. Each pointer contains the address of an index set record at the next lower level along with the highest key in the index set record pointed to.

Two options exist that can be specified for the VSAM index that especially affect performance. The first option is specified in the IMBED | NOIMBED parameter in the `DEFINE CLUSTER` command. If you specify IMBED, the sequence set level of the

Determining Which VSAM Options to Use

VSAM index is stored in the data cluster. This improves performance, because, when index and data are close, fewer seek operations are needed to process the data.

If IMBED is specified with the REPLICATE parameter (discussed in the following paragraph), the sequence set level is stored in the data cluster. Each sequence set record for each CA is written as many times as will fit on the first track of the appropriate CA. The reason a sequence set record is repeated is to reduce the delay caused when the disk rotates.

The repetition of sequence set records means the read/write head will almost always be close to or over a sequence set record so very little disk rotation is necessary. Repeating records also improves performance. The track sequence set records are put on is the first track allocated for the CA. Although specification of IMBED improves performance, available space for data in a CA decreases by one track for each CA. Also, track recovery cannot be performed if IMBED is specified.

If you specify NOIMBED, the sequence set level is not stored with the database. NOIMBED is the default for this parameter.

The other option you can specify for the VSAM index that especially affects performance is the REPLICATE | NOREPLICATE parameter in the `DEFINE CLUSTER` command. If you specify REPLICATE, each record in the sequence set and the index set is written as many times as it will fit on the track. Repeat records to reduce the delay caused when the disk rotates. The repetition of records means the arm is almost always close or over a record so very little disk rotation is necessary. Repeating records also improves performance. Note, however, that the VSAM index, because of the repetition, will probably require more direct-access space.

If you specify NOREPLICATE, records in the VSAM index are not repeated. NOREPLICATE is the default for this parameter.

There is a new option that you must specify for KSDSs in order to take 'fuzzy' image copies using the Database Image Copy 2 utility. BWO(TYPEIMS) is the specification. The KSDS must be SMS-managed for BWO(TYPEIMS) to mean anything. And, you should ensure that all access to the KSDS (once the BWO(TYPEIMS) option has been specified) is under DFSMS 1.3 or higher.

Determining Which OSAM Options to Use

Two types of options are available for databases using OSAM:

1. Options specified in the DBD (free space, logical record size, CI size).

   These options are covered in preceding sections of 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/ESA Installation Volume 2: System Definition and Tailoring*. The statement and all its parameters are optional.

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 abnormal termination to take place when abnormal termination occurs in the buffer handler (an internal component).

Deciding Which FIELD Statements to Code in the DBD

"Chapter 10. Establishing Standards and Procedures" on page 271 describes the statements that are coded in the DBD. One of those statements is the FIELD statement, which defines a field within a segment type. An important thing to note about the FIELD statement is that it has to be coded for sequence fields and for fields an application program can refer to in the SSA of a call. A FIELD statement also has to be coded if it is referenced by a SENFLD statement in any PSB. Because each FIELD statement takes up storage in the DMB control block, do not generate FIELD statements that are unnecessary.

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 (described in "Chapter 15. Modifying Your Database" on page 365).

In the online environment, the Image Copy utilities and Online Recovery allow you to do some maintenance without taking the database offline. These utilities let you take image copies of databases while they are allocated to and being used by an online IMS system.

Using Design Aids for Your Database

The DB/DC Data Dictionary is discussed in this section.

DB/DC Data Dictionary

If you have the DB/DC Data Dictionary, it can be an accessible collection of the definitions of your installation's data resources. Therefore, it can be useful as a starting point for designing a database. These definitions can be much more than descriptions of database elements. They can include information about the use of data in the system, the relationships among data elements, and the relationships between data elements and business processes. These might include business entities such as personnel, departments, data processing devices, and projects.

Specifically, to benefit most from a data dictionary during development of any system, the attributes and relationships of the following entities should be entered as they become known:

- Database elements
- Segments
- Fields
- PCBs and PSBs
- Transactions
- Programs
- Modules
- Report structures
- MFS control blocks

Chapter 7. Designing a Fast Path Database

Choosing a Database Type

IMS allows you to define nine different types of databases. Each type has a different organization or different processing characteristics. Only the two Fast Path database types, MSDB and DEDB, are discussed in this chapter.

Understanding the differences between the two databases types allows you to pick the type of database that best suits your application's processing requirements.

Each IMS database type has its own access method, and each database type is named after the access method it uses. Here are the two Fast Path database types and the access methods they use:

Type of Database	Access Method
MSDB	Main Storage Database
DEDB	Data Entry Database

Note: MSDB not supported for DBCTL.

Both database types use the direct method of storing data. With this method, the hierarchic sequence of segments is maintained by putting direct-address pointers in each segment's prefix.

For quick reference, see Table 9 on page 213 for a summary of DEDB and MSDB database characteristics.

Databases Supported With DBCTL

DBCTL supports data entry databases (DEDBs), but does *not* support main storage databases (MSDBs).

Databases Supported With DCCTL

DCCTL does not support MSDBs and DEDBs.

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 32000 bytes with a maximum key length of 240 bytes. Additional prefix data extends the maximum total record size to 32258 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

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.

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.

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.

How MSDBs Are Stored

The MSDB Maintenance utility (DBFDBMA0) creates the MSDBINIT sequential data set in physical ascending sequence (see Figure 106 on page 197). 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 Figure 105).

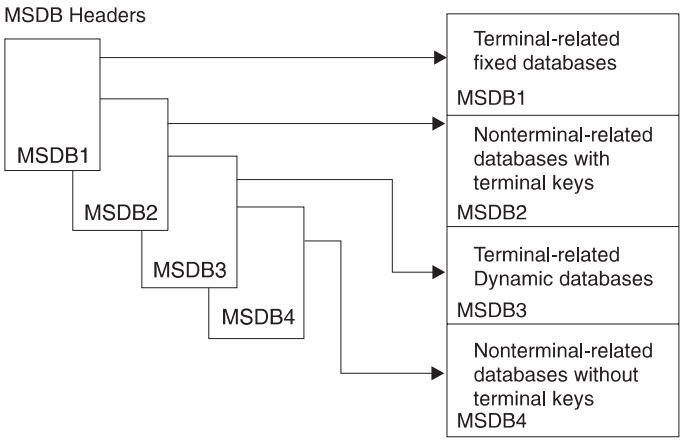


Figure 105. MSDB Pointers

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.

Diagnosis, Modification or Tuning Information

Figure 106 shows the MSDBINIT record format.

End of Diagnosis, Modification or Tuning Information

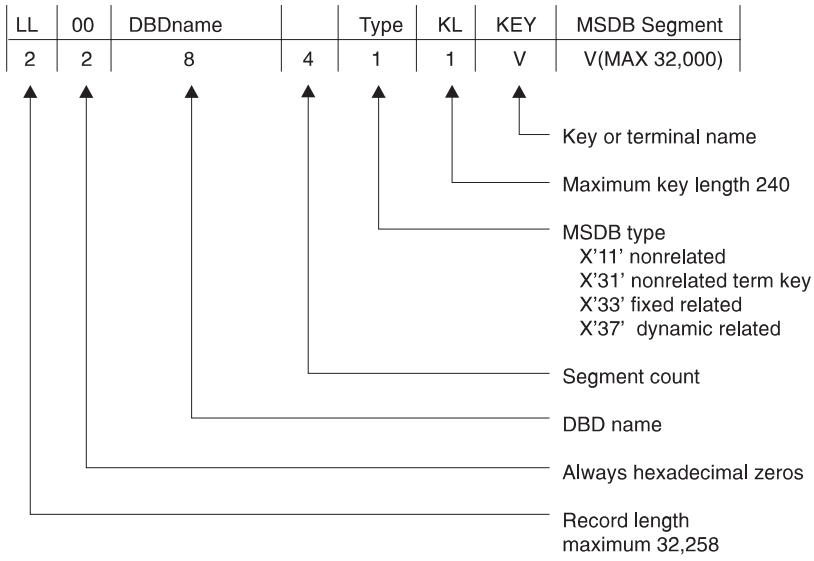


Figure 106. MSDBINIT Record Format

How an MSDB Record Is Stored

This section contains diagnosis, modification, or tuning information.

MSDB records contain no pointers except the forward chain pointer (FCP) connecting free segment records in the terminal-related dynamic database.

Figure 107 shows a high-level view of how MSDBs are arranged in priority sequence.

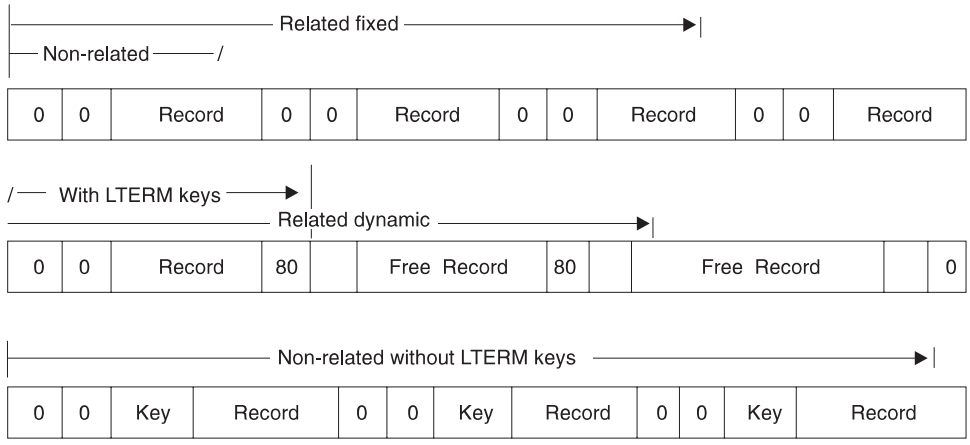


Figure 107. Sequence of the Four MSDB Organizations

How MSDBs Are Saved

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 above 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 table shows the type of compare or search technique used for a qualified SSA.

Insertion and Deletion of Segments

The terminal-related dynamic database accepts ISRT and DLET calls, and the other

	Sequence Field	Nonsequence Field	
		Arithmetic	Nonarithmetic
Type of Compare	Logical	Arithmetic	Logical
Type of Search	Binary if operator is ’=’, or ’>’, or ’>=’; otherwise, Sequential	Sequential	

Figure 108. Sequence and Nonsequence Types of Compares and Searches

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.

Figure 109 on page 199 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.

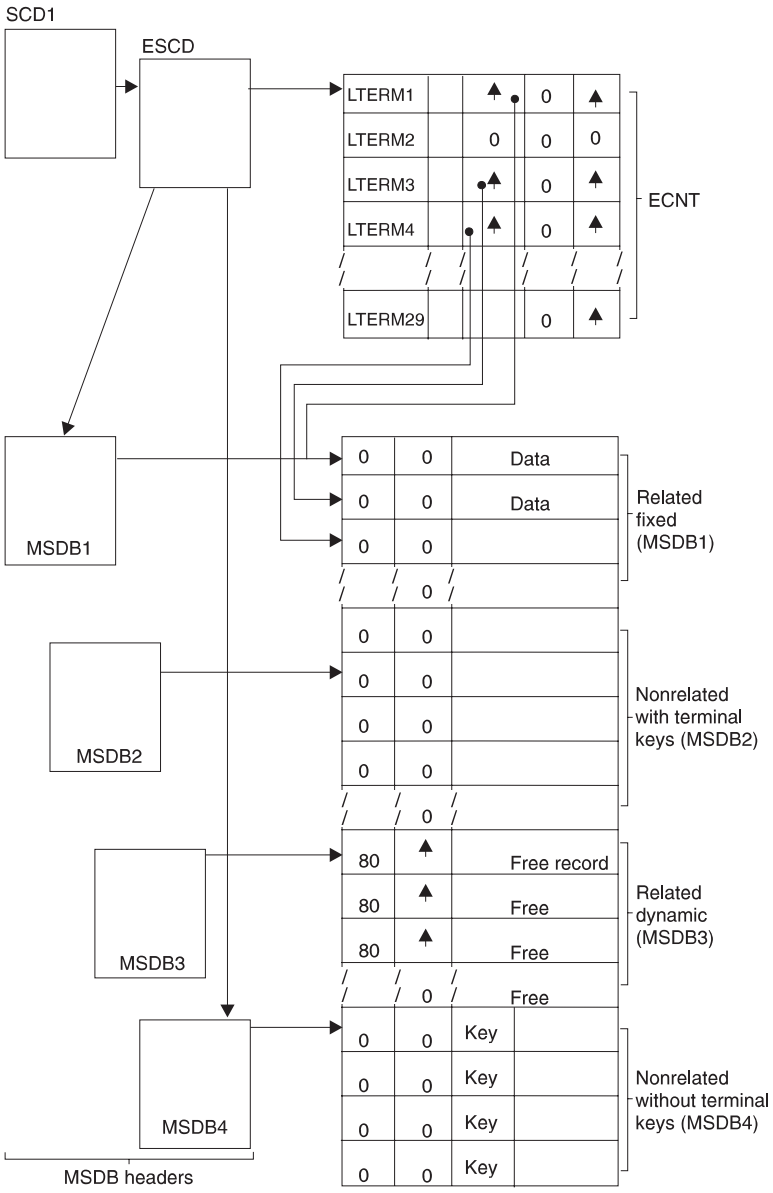


Figure 109. ECNT and MSDB Storage Layout

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. Figure 110 shows the relationship between the ECNT contents and the MSDBs.

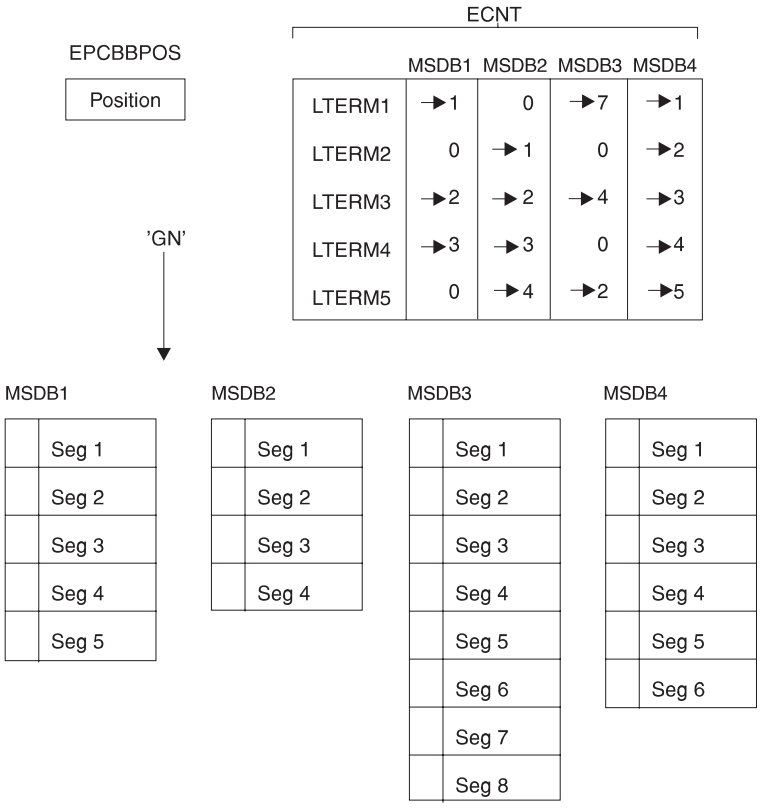


Figure 110. Position in an MSDB

Assume the current position for MSDB3 has been established for LTERM3. The table entry refers to segment 4 as the current segment. An unqualified GN retrieves segment 2. An additional GN call results in a “GB” status, indicating end of data.

The Field Call

The DL/I FLD call is available to MSDBs and DEDB. It allows for the operation on a field, rather than on an entire segment. Additionally, it 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. This call is described in “Processing Fast Path Databases” in *IMS/ESA Application Programming: Database Manager*.

Things to Know about 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 above differences become more critical when transactions update or refer to both full function DL/I and MSDB data. Updates to full function DL/I and DEDB databases 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

Data Entry Databases (DEDBs)

A data entry database (DEDB) is a hierarchic database containing up to 127 segment types (a root segment, an optional sequential dependent segment, and 0 to 126 direct dependent segments). If the optional sequential dependent segment type is defined, 125 direct dependent segment types can be defined. A DEDB structure can have as many as 15 hierarchic levels. Sequential dependent segment occurrences for an area are stored in chronological order, regardless of the root on which they are dependent. Direct dependent segments are stored in hierarchic fashion, allowing for rapid retrieval.

When to Use a DEDB

DEDBs are designed to provide efficient storage for and access to large volumes of data, as well as, a high level of availability for that data. Three characteristics that improve DEDB’s availability are:

area format

area data set replication

Record deactivation

These characteristics are useful when you must gather detailed and summary information.

Area Format

The physical format of this type of database makes the data more readily available. In a traditional hierarchic IMS database, 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. A DEDB can use multiple data sets, called areas, with each area containing the entire data structure (see Figure 118 on page 210). A DEDB record (a root and its dependent segments) does not span Areas. A DEDB can be divided into as many as 240 such Areas. This organization is transparent to the application program.

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. Areas of a DEDB can be allocated on different volume types.

Initialization, reorganization, and recovery are done on an area basis. Resource allocation is done at the CI level. Multiple programs, optionally together with one online utility, can concurrently access an area within a database, providing they are using different CIs. CI sizes of 512, 1K, 2K, 4K, up to 28K in 4K increments are allowed. The media manager and Integrated Catalog Facility catalog of Data Facility Product (DFP) are required.

Areas must be pre-formatted. See “Parts of a DEDB Area” on page 203 for a description of the independent overflow part of an Area.

Each area in a DEDB is a VSAM data set. An area is opened by the first call to it from a program that is eligible to access it. A single area in a DEDB can be stopped by the operator with the `/STOP AREA` command. A DEDB can be stopped with the `/STOP DATABASE` command. These commands do not affect programs currently scheduled against the DEDB. The `/STOP DATABASE` command prevents scheduling of any new programs needing access to the stopped database.

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 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.

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 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 database remain identical, IMS updates all copies when a change is made to only one copy.

If an area data set (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.

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. It only prevents them 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.

Parts of a DEDB Area

A DEDB area consists of three parts:

- Root addressable part
- Independent overflow part
- Sequential dependent part

Figure 111 shows DEDB area division.

Figure 111. DEDB Area Division

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.

Figure 112 shows the root addressable part of a DEDB.

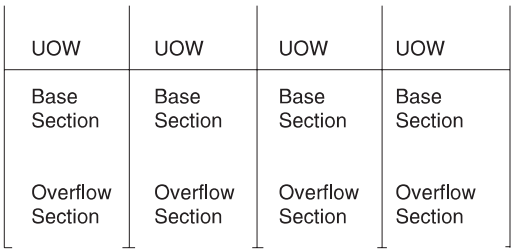


Figure 112. Root Addressable Part of a DEDB

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. Once 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. Figure 113 on page 205 shows a more detailed view of the independent overflow part in relation to the other parts in the DEDB. It shows how a CI fits into a DEDB Area.

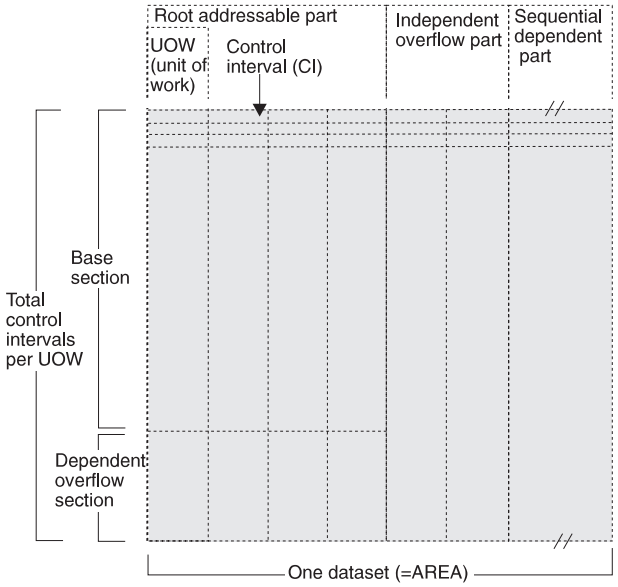


Figure 113. DEDB Composite Storage View

Sequential Dependent Part: The sequential dependent part holds sequential dependent segments from roots in all UOWs in the area (see Figure 118 on page 210). 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 Delete utility DBFUMDLO to delete a contiguous portion, or all the sequential dependent segments in that part.

CI and Segment Formats: This section contains diagnosis, modification, or tuning information.

The following four diagrams show:

- CI format
- Root segment format
- Sequential dependent segment format
- Direct dependent segment format

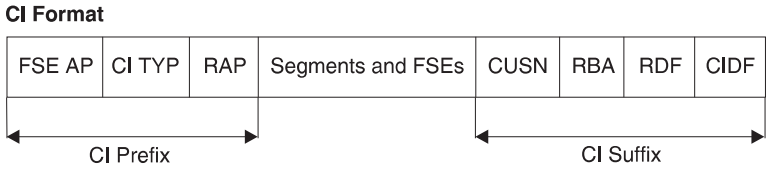


Figure 114. CI Format

Table 5. CI Format

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.

Table 5. CI Format (continued)

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).

Root Segment Format (with Sequential and Direct Dependent Segments with Subset Pointers)

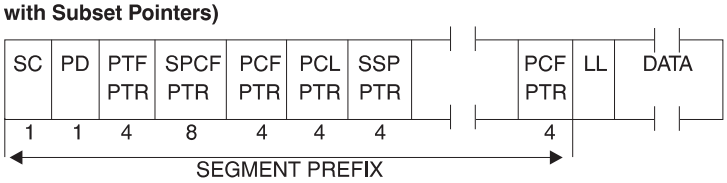


Figure 115. Root Segment Format (with Sequential and Direct Dependent Segments with Subset Pointers)

Table 6. Root Segment Format

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.
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.

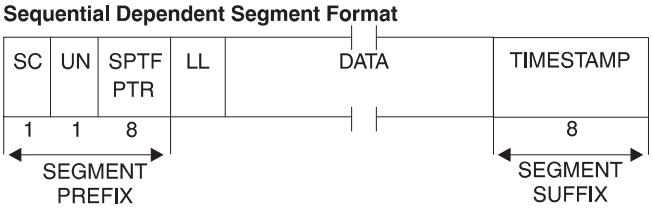


Figure 116. Sequential Dependent Segment Format

Table 7. Sequential Dependent Segment Format

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.

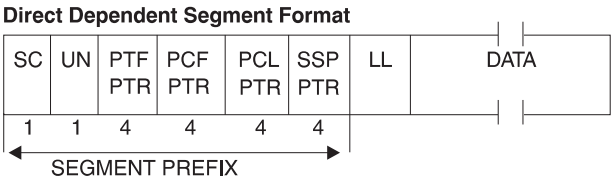


Figure 117. Direct Dependent Segment Format

Table 8. Direct Dependent Segment Format

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.
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.

How Root Segments Are Stored

DEDB root segments are stored as prescribed by the randomizing routine, and are chained in ascending key sequence from each anchor point. For information on the system-supplied or user-supplied randomizing module for DEDBs, see *IMS/ESA Customization Guide* . 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.

How Direct Dependent Segments Are Stored

The DEDB maintains processing efficiency while supporting a hierarchic 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. For more information on defining and using subset pointers, see *IMS/ESA Application Programming: Database Manager*.

How Sequential Dependent Segments Are Stored

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 within 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 sequential dependent (SDEP) scan utility, described in *IMS/ESA Utilities Reference: Database Manager*. SDEP segments are then processed by offline jobs.

SDEP segments are used for data collection, journaling, and auditing applications.

Enqueue Level of Segment CIs

This section contains diagnosis, modification, or tuning information.

Allocation of CIs involves three different enqueue levels.

- A NO ENQ level, which is typical of any SDEP CI. SDEP segments can never be updated, so 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're held at the SHARED level. Of course, there is the potential for deadlock.

The level of enqueue, just described, is reexamined each time the buffer stealing facility is invoked. (Refer to the section "Fast Path Buffer Allocation Algorithm" in "Chapter 6. Database Design Considerations for Full Function" on page 165for information about this facility.) 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.

Figure 118 on page 210 shows an example of DEDB structure.

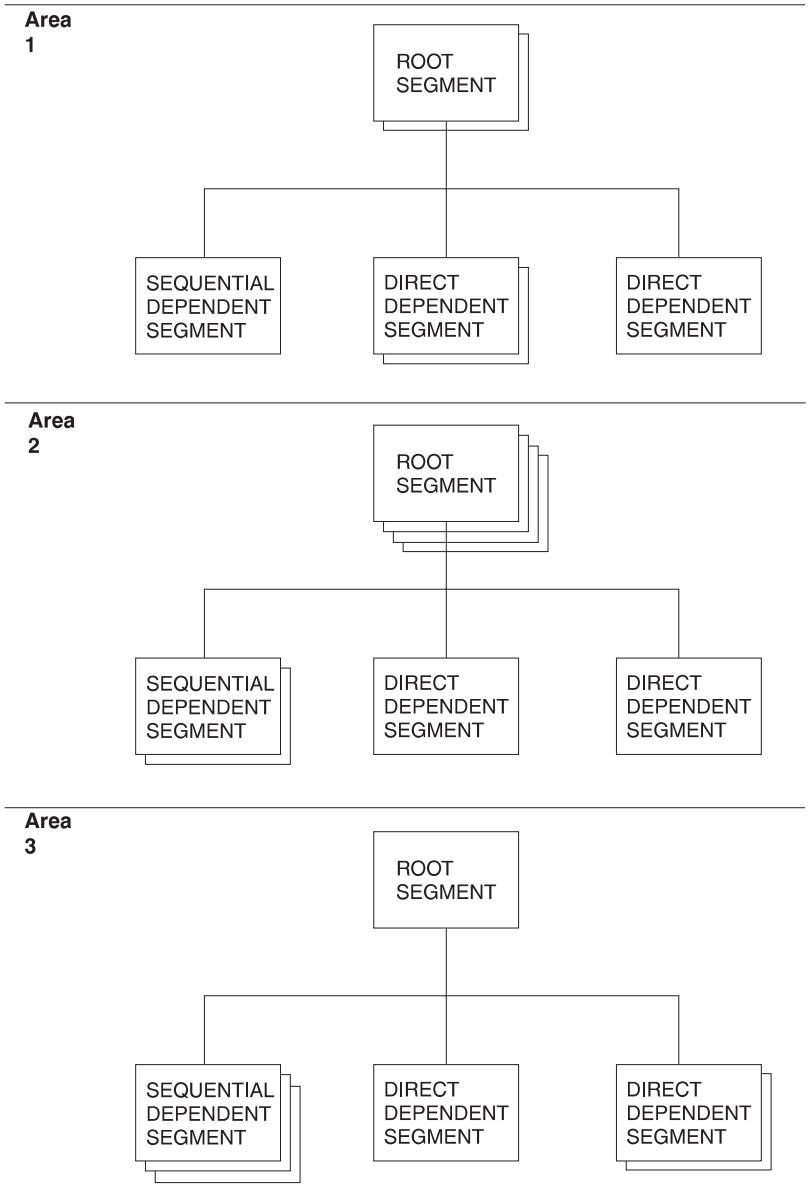


Figure 118. DEDB Structure Example

How the DEDB Space Search Algorithm Works

This section contains diagnosis, modification, or tuning information.

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.

This 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. The trade-off is improved performance for future database record access versus optimum space utilization.

DEDB Insert Algorithm

This section contains diagnosis, modification or tuning information.

The DEDB insert algorithm searches for additional space when space is not available in the most desirable block. 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. For information on DEDB CI format and allocation, see *IMS/ESA Diagnosis Guide and Reference*.

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.

Once 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 example describes 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 119 on page 212 shows extending a UOW into independent overflow.

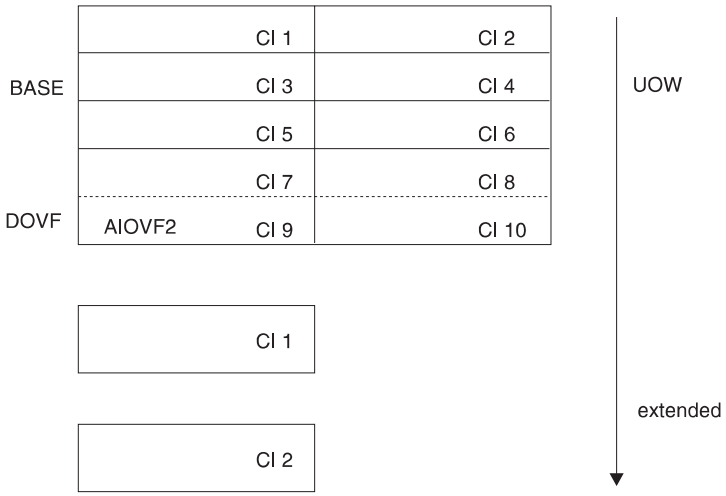


Figure 119. Extending a UOW to Use Independent Overflow

DEDB Free Space Algorithm

This section contains diagnosis, modification, or tuning information.

The DEDB free space algorithm is used to free dependent overflow and independent overflow CIs. 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 section.

A dependent overflow or an independent overflow CI can be freed in two ways:

- Reorganization
- Segment deletion

Reorganization: During online reorganization, the segments within a UOW are read in GN order and written to the reorganization UOW. This process inserts segments into the reorganization UOW, eliminating embedded free space. If all the segments do not fit into the reorganization UOW (RAP CI plus dependent overflow CIs), then new independent overflow CIs are allocated as needed. When the data in the reorganization UOW is copied back to the correct location, then:

- 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.

Considerations Related to Unusable Space

This section contains diagnosis, modification, or tuning information.

Space in a DEDB should be closely monitored to avoid out-of-space conditions for an Area. Products such as Database Tools (5685-093) DEDB Tuning Aid and Database Analyzer (5665-349) 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 Database Tools DEDB Tuning Aid and the Database Analyzer can assist in determining if the data distribution is reasonable.

DL/I Calls against a DEDB

This section contains diagnosis, modification, or tuning information.

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 database.

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. (See “Processing DEDBs with Subset Pointers” in *IMS/ESA Application Programming: Database Manager*.)

Table 9 compares the database types.

Table 9. Summary of Database Types

	HSAM	HISAM	HDAM	HIDAM	DEDB	MSDB
Hierarchical Structures	yes	yes	yes	yes	yes	no
Direct Access Storage	yes	yes	yes	yes	yes	no
Multiple Data Set Groups	no	no	yes	yes	no	no
Logical Relationships	no	yes	yes	yes	no	no
Variable-Length Segments	no	yes	yes	yes	yes	no
Segment Edit/Compression	no	yes	yes	yes	yes	no
Data Capture exit routines	no	yes	yes	yes	yes	no
Field-Level Sensitivity	yes	yes	yes	yes	no	no
Primary Index	no	yes	no	yes	no	no
Secondary Index	no	yes	yes	yes	no	no
Logging, Recovery, Reorg	no	yes	yes	yes	yes	yes
VSAM	no	yes	yes	yes	yes	N/A
OSAM	no	no	yes	yes	no	N/A
QSAM/BSAM	yes	no	no	no	no	N/A
Boolean Operators	yes	yes	yes	yes	yes	no
Command Codes	yes	yes	yes	yes	yes	no
Subset Pointers	no	no	no	no	yes	no
Use Main Storage	no	no	no	no	no	yes
High Parallelism (field call)	no	no	no	no	no	yes
Compaction	yes	yes	yes	yes	yes	no
DBRC Support	yes	yes	yes	yes	yes	N/A

Table 9. Summary of Database Types (continued)

	HSAM	HISAM	HDAM	HIDAM	DEDB	MSDB
Partitioning Support	no	no	no	no	yes	no
Data Sharing	yes	yes	yes	yes	yes	no
Partition Sharing	no	no	no	no	yes	no
Block Level Sharing	yes	yes	yes	yes	yes	no
Area Sharing	N/A	N/A	N/A	N/A	yes	N/A
Record Deactivation	no	no	no	no	yes	N/A
Database Size	med	med	med	med	large	small
Online Utilities	no	no	no	no	yes	no
Batch	yes	yes	yes	yes	no	no

DEDB Areas in Data Sharing

The concept of working with Areas has special meaning for the data sharing environment. Each area can be individually shared across multiple DB/DC environments. Data sharing considerations for Fast Path are discussed in *IMS/ESA Administration Guide: System*.

Additionally, using the SHARELVL parameter, you can specify the level of data sharing for which subsystems can share a database. If any subsystem has already authorized the database, changing the SHARELVL does not modify the database record. The SHARELVL parameter applies to all Areas in DEDB.

For more information on the SHARELVL parameter and data sharing, refer to *IMS/ESA Utilities Reference: System*.

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". The performance and resource use implications are discussed in "Fast Path Synchronization Points" on page 215.

Converting MSDBs to DEDBs

You can convert your MSDBs to DEDBs using the MSDB-to-DEDB Conversion utility; see *IMS/ESA Utilities Reference: Database Manager*. Once you have converted your MSDBs to DEDBs (a DEDB with one area), you can choose whether to use the area as Virtual Storage Option (VSO) or not. To help you convert your MSDB applications to DEDB applications, DEDBs can have the following MSDB characteristics:

- The field (`FLD`) call

  For more information on this, see *IMS/ESA Application Programming: Database Manager*.
- Fixed-length segments

  For more information on this, see "Using Fixed-Length Segments in DEDBs" on page 215.
- MSDB or DEDB commit view

For more information on this, see *IMS/ESA Application Programming: Database Manager* .

Using Fixed-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 existing MSDB applications for your DEDBs.

You define fixed-length segments during DBDGEN in the SEGM macro by specifying a single value for the `BYTES=` parameter. To define variable-length segments, specify two values for the `BYTES=` parameter.

Application programs for MSDBs do not see the length (LL) field at the beginning of each segment. Likewise, application programs for fixed-length-segment DEDBs 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

Figure 120 and Figure 121 show examples of how to use the `BYTES=` parameter to define variable-length or fixed-length segments.

```
ROOTSEG  SEGM  NAME=ROOTSEG1,                                  C
               PARENT=0,                                      C
               BYTES=(390,20)
```

Figure 120. Defining a Variable-Length Segment

```
ROOTSEG  SEGM  NAME=ROOTSEG1,                                  C
               PARENT=0,                                      C
               BYTES=(320)
```

Figure 121. Defining a Fixed-Length Segment

Fast Path Synchronization Points

This section contains diagnosis, modification, or tuning information.

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.

Phase 1 - Build Log Record

This section contains diagnosis, modification, or tuning information.

Fast Path Synchronization Points

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

This section contains diagnosis, modification, or tuning information.

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 I/O 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.

Monitoring and Tuning 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 IMS Monitor is not the prime tool for monitoring Fast Path applications.

Instead, you should 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:

1. A detailed listing of exception transactions
2. A summary of exception detail by transaction code for MPP (message-processing program) regions
3. A summary by transaction code for MPP regions
4. A summary of IFP, BMP, and CCTL transactions by PSB name or transaction code
5. A summary of the log analysis

Do not confuse this utility with the IMS Monitor or the IMS Log Transaction Analysis utility.

For more information on CCTL transactions, refer to *IMS/ESA Customization Guide*. This chapter introduces the Fast Path Log Analysis utility. For more detailed information on the utility, see *IMS/ESA Utilities Reference: System*.

As an administrator in the Fast Path environment, you should perform tasks, like establishing monitoring strategies, performance profiles, and analysis procedures. This section highlights how to use the Analysis utility to do these tasks, and suggests some Areas where tuning activities might be valuable.

Using the 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/ESA Utilities Reference: System* .

Fast Path Log Reduction

To reduce log volume you can use the LGNR parameter, which is specified during IMS startup. LGNR indicates the maximum number of DEDB buffer alterations that are held before the entire CI is logged. For more information on log reduction and the LGNR parameter, see *IMS/ESA Utilities Reference: System*.

Fast Path Transaction Timings

For each Fast Path transaction, four time intervals are separately calculated. Figure 122 shows the boundary events and intervals.

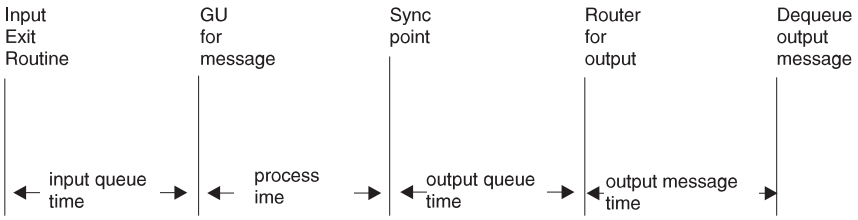


Figure 122. Fast Path Transaction Event Timings

- The first interval reflects the transaction input queuing within the balancing group to distribute the work.
- The second interval records the actual elapsed processing time for the individual transaction.
- The third interval shows the effect of sync point in delaying the output message release until after logging.
- The last interval 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. Table 10 shows the information that is made part of the system log

records for each Fast Path transaction.

Table 10. Monitor Data for Fast Path Transactions

Monitored Data	Message-Driven Region	Other Region
Transit and Output-msg 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
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/ESA Utilities Reference: System*, 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.

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

Factors Influencing Fast Path Performance

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. For information on this program, see *IMS/ESA Utilities Reference: System*.

The remainder of this chapter consists of sections that discuss each of the above factors.

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 an application's response time:

- Data replication
- Subset pointers
- Complexity of hierarchic structure
- Complexity of DL/I calls
- Use of sharing across IMS
- Last child pointers

The first two characteristics are unique to DEDBs; the last four 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 asynchronous 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 Buffer Pool

The Fast Path buffer pool is used by all Fast Path programs except the DEDB online utilities, 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 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. See *IMS/ESA Installation Volume 2: System Definition and Tailoring*for details of APPLCTN and TRANSACT class specifications.

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.)
- Limit the use of the overflow buffer interlock, 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 will 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 400.

Utilization of Available Real Storage

The amount of page-fixed storage defined will be a significant consideration in limited storage systems. The factors influencing real storage utilization are summarized in “Appendix B. Replace, Insert, and Delete Rules for Logical Relationships” on page 409.

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 and/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.

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:

  Either IMS was able to get the required IRLM latches, and there is no other interest on this resource.

  Or the request is compatible with other holders and/or waiters.
- **Asynchronously** because the request could not get the required IRLM latches and was suspended. The lock is granted when latches become available due to one of two conditions:

  Either no other holders exist.

  Or the request is compatible with other holders and/or waiters.
- **Asynchronously** because the request is not compatible with the holders and/or waiters and was granted after their interest was released.

In a Sysplex environment, global locks can be granted in one of three ways:

- **Locally by the IRLM** because:

  Either there is no other interest for this resource.

  Or this IRLM has the only interest, this request is compatible with the holders and/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 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).

Resource Name Hash Routine

The Fast Path Resource Name Hash routine generates the hash value used by the IRLM. You may specify the name of such a routine with the USRHASH parameter on the FPCTRL macro, but it is ignored. Using the UHASH parameter in the IMS procedure, you can override this name.

Because this routine can be user supplied, it is possible to provide your own hashing logic to satisfy any special requirements. However, from the performance viewpoint, it is highly desirable that a resource name be hashed to be distributed over a potentially wide number of GHT entries.

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.

Registering Databases

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, allowing recovery to be deferred to a

convenient time. I/O error retry is automatically performed at database close time. If the retry is successful, the error condition no longer exists and recovery is not needed.

Although databases need not be registered in DBRC in order for the error handling to work, it is highly recommended. If an error occurs on a non-registered database and the system terminates, the database could be damaged if the system is restarted and a /DBR command is not issued prior to accessing the database. The reason for this is that restart causes the error buffers to be restored as they were when the system terminated. If the same block had been updated during the batch run, the batch update would be overlaid.

Fast Path Virtual Storage Option

The Fast Path Virtual Storage Option (VSO) allows you to map data into virtual storage (into an MVS data space). You can map one or more data entry database (DEDB) Areas into virtual storage by defining them as VSO Areas.

For high-end performance applications with DEDBs, using DEDBs with VSO Areas instead allows you to realize the following performance improvements:

- Reduced read I/O

  Once a VSAM control interval (CI) has been brought into virtual storage, all subsequent I/O read requests read the data that is in virtual storage rather than on 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. You use DBRC commands to specify that you want to use a DEDB as a VSO DEDB.

Important: Terminal-related MSDBs and non-terminal-related MSDBs with terminal-related keys are **not supported** in IMS Version 5 and later releases. Non-terminal-related MSDBs without terminal-related keys are still supported in IMS Version 6. Therefore, you should consider converting all your existing MSDBs to VSO DEDBs or non-VSO DEDBs.

Enhancements to DEDBs

In order to help make the conversion from MSDBs to DEDBs as painless as possible, DEDBs have been given many of the functions of MSDBs. These include:

- Putting DEDB Areas into virtual storage (VSO)

- The field (FLD) call
- Fixed length segments
- MSDB or DEDB commit view

In addition, DEDBs have the following:

- Full DBRC support
- Block level sharing of Areas

  DEDB Areas are available to DBCTL and LU 6.2 applications, as well as DB/DC applications.
- DEDBs can be tracked in an RSR environment
- HSSP support for DEDBs
- Availability of DEDB utilities
- Online database maintenance
- Support for the full hierarchical model

  Calls which were not available to MSDBs, such as Insert and Delete, are available to DEDBs
- Improved search techniques

  MSDBs used the binary search technique; DEDBs use the randomizer search technique.
- MSDB checkpoint data sets are not required for DEDBs

In addition, the Fast Path Log Analysis utility (DBFULTA0) has been enhanced to provide more log information and VSO activity for SHARELVLs 0-3 option settings information; see *IMS/ESA Utilities Reference: System*.

Restrictions Using VSO DEDB Areas

VSO DEDB Areas have the following restrictions in their use:

- VSO Areas *must* be registered with DBRC
- The maximum allowable size for an MVS data space is two gigabytes (2 147 483 648 bytes)

  The actual size available for a VSO area is the maximum size (2 GB) minus an amount reserved by MVS (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 available for a particular Area.

  The/DISPLAY FPVIRTUAL command report displays the following VSO information for SHARELVLs 0/1; dataspace, maxsize, areaname, areasize, option (such as PREO and PREL). Then it shows coupling facility name, pool (private pool size), LKASID (Indicates if buffer lookaside is active for this pool), areaname, areasize, and option (such as PREO and PREL). The /DISPLAY FPVIRTUAL command report displays the following VSO information for SHARELVLs 2/3; areaname, structure, entries, changed, poolname, and options.
- You cannot use the DEDB Multiple area data set Compare utility (DBFUMMH0) for a VSO DEDB

Related Reading:

- See “Accessing a Data Space” on page 234 for more information on data space.
- See *IMS/ESA Operator’s Reference* for more information on the /DISPLAY commands.

Defining a VSO DEDB Area

All of the information which defines a DEDB as a DEDB using the Virtual Storage Option (VSO) is recorded in the RECON data set. You use the INIT.DBDS and CHANGE.DBDS commands to define your VSO DEDB Areas, using the following keywords:

VSO This defines the area as a VSO Area.

To define an area as a VSO Area implies that when a CI is read for the first time, it will be copied into an MVS data space. Data is read into a common buffer and is then copied into the data space. Once the data is in the data space, subsequent access to the data retrieves it from the data space rather than from DASD. Those CIs that are not accessed are not brought into the data space. All updates to the data are copied back to the data space and any locks held are released. Updated CIs are periodically written back to DASD.

NOVSO

This 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 when it is opened. This keyword implies the PREOPEN keyword, thus if PRELOAD is specified, then PREOPEN does not have to be specified.

Using PRELOAD implies that the root addressable portion and the independent overflow portion of an area are loaded into the data space at control region initialization or during /START AREA processing. Once the area is loaded into the data space, data is read from the data space to a common buffer. Updates are copied back to the data space, any locks are released, and updated CIs are periodically written back to DASD.

NOPREL

This defines the area as load-on-demand. For VSO DEDBs, as CIs are read from the data set, they are copied to the data space. This is the default.

To define an area with NOPREL gives you the ability to deactivate the preload processing, so that the area will not be preloaded into the data space next time it is opened.

If you specify NOPREL, and you want the area to be preopened, you must separately specify PREOPEN for the Area.

CFSTR1|2

These define the coupling facility cache structure names. These must follow MVS coupling facility naming conventions. These parameters are valid only for VSO Areas of DEDBs defined with SHARELVL(2|3). For detailed information see “Coupling Facility Structure Naming Convention” on page 229.

The following two keywords for the INIT.DBDS and CHANGE.DBDS commands apply both to VSO DEDB Areas and non-VSO DEDB Areas.

PREOPEN This opens the area after the first checkpoint following control

region initialization. Using this keyword, application programs do not incur the overhead of open processing.

PREOPEN can be specified for both VSO and non-VSO Areas. It removes the burden of Area-open processing from the application. At control region startup, all DEDB Areas defined with PREOPEN are opened. /START AREA processing preopens an area defined with PREOPEN, and starts it.

NOPREO This opens the area during the first call to the area. This is the default.

To define an area with NOPREO gives you the ability to deactivate the preopen processing, so that the area will not be preopened at the next startup or /START AREA command.

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. When you use a /STOP AREA command, any necessary VSO processing is performed. See *IMS/ESA Operator's Reference* for details on start and stop processing.

If the area needs to be preopened or preloaded, it will be done at this time.

Defining a VSO Cache Structure Name

The system programmer defines all coupling facility structures, including VSO cache structures, in the XES 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

The structure name is 16 characters long, padded on the right with blanks if necessary. It 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

Examples of Defining Coupling Facility Structures

The following example 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)
/*
```

Figure 123. Example of Updating a Policy with New Structures

In the example, the programmer defined one list structure (`LIST01`) and one cache structure (`CACHE01`)

Restriction: When defining this cache structure to DBRC, ensure that the name is identical (see “Registering a Cache Structure Name with DBRC”).

Registering a Cache Structure Name with DBRC

Structure names are maintained in the DBRC RECON data set. Register structure names on the INIT.DBDS or CHANGE.DBDS statement. The keywords for specifying structure names are CFSTR1 and CFSTR2. The following example registers structure name `TSTDEDBAR1`:

```
INIT.DBDS DBD(DEDBFR01) AREA(DEDBAR1) VSO PRELOAD CFSTR1(TSTDEDBAR1)
```

Figure 124. Defining a VSO area Coupling Facility Structure Name in DBRC

Defining a Private Buffer Pool Using the DFSVSMxx VSPEC Member

Define a private buffer pool using the following format:

Defining a Private Buffer Pool

```
DEDB=(poolname,size,pbuf,sbuf,maxbuf,lkasid,dbname)
```

where:

POOLNAME 8 character name of the pool. Used in displays and reports.

SIZE The buffer size of the pool. All the standard DEDB-supported buffer sizes are supported.

PBUF The primary buffer allocation. The first allocation will get this number of buffers. Maximum value is 99999.

SBUF The secondary buffer allocation. When the primary allocation starts to run low, another allocation of buffers is made. This amount indicates the secondary allocation amount. Maximum value is 99999.

MAXBUF The maximum number of buffers allowed for this pool. It is a combination of PBUF plus some iteration of SBUF. Maximum value is 99999.

LKASID Indicates whether this pool is to be used as a local cache with buffer lookaside capability. This value is cross-checked with the DBRC specification of LKASID to determine which pool the area will utilize. If there is an inconsistency between the DEDB statement and DBRC, the DBRC value takes precedence.

DBNAME Association of the pool to a specific area or DBD. If the DBNAME is an area name, then the pool will be used only by that Area. If the DBNAME specifies a DBD name, the pool will be used by all areas in that DBD. The DBNAME is first checked for an area name then for a DBD name.

The following example defines a private buffer pool:

```
DEDB=(POOL1,512,400,50,800,LKASID)
DEDB=(POOL2,8196,100,20,400,NOLKASID)
```

Figure 125. Examples of Defining Private Buffer Pools

In this example, 2 private buffer pools are defined:

1. The first pool 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 will be used as a local cache, and buffer lookaside will be performed for Areas that share this pool.
2. The second pool 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 will be 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 used:

Default Private Buffer Pool Statement

```
DEDB=(poolname,XXX,64,16,512)
```

where:

- `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.

Block-Level Sharing of VSO DEDB Areas

Block-Level Sharing of VSO DEDB Areas allows multiple IMS subsystems to concurrently read and update VSO DEDB data. The three main participants are the coupling facility hardware, the coupling facility policy software, and the XES and MVS services:

- The coupling facility hardware provides high-performance, random-access shared storage in which IMS subsystems 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 MVS data set, the *couple data set* (COUPLExx).
- The coupling facility policy software and its cache structure services provide interfaces and services to MVS 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 MVS services provide a way of manipulating the data within the cache structures. They provide high performance, data integrity, and data coherency for multiple IMS subsystems sharing data.

The Coupling Facility and Shared Storage

Each VSO DEDB area is represented in the coupling facility shared storage by one cache structure. These cache structures are **not persistent**. That is, they are deleted after the last IMS subsystem disconnects from the coupling facility.

Imagine one VSO DEDB area holding data that two or more IMS subsystems in a sysplex will share. Figure 126 on page 233 represents the coupling facility shared storage that provides structure storage of varying sizes (S1 through S9). The cache structure will be used for VSO DEDB data. The structure name `A1$0XXX111222333` represents our VSO Area.

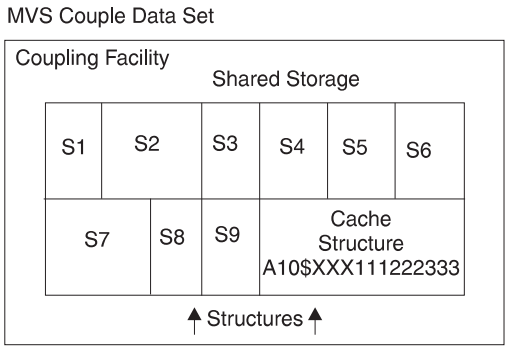


Figure 126. Coupling Facility Shared Storage

Each cache structure in the coupling facility is composed of a directory portion and a data portion.

The basic unit of interaction with the coupling facility is the **directory portion**. It contains a name consisting of the characters "VSO," the area name, and the relative byte address (RBA). For example, a directory entry name might be `vsoArea1rba1`. Only one directory can exist in a particular CI, so each CI within the area and within the cache structure is uniquely identified.

The detail of `CACHE STRUCTURE A10$XXX111222333` is represented in Figure 127.

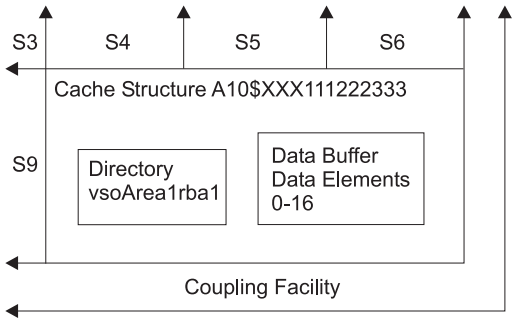


Figure 127. Detailed View of CACHE STRUCTURE A10$XXX111222333

Duplexing Structures

Duplexing structures are duplicate structures for the same Area.

Private Buffer Pools

IMS now 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 new keywords LKASID or NOLKASID, when specified on the DBRC commands INIT.DBDS or CHANGE.DBDS, indicate whether to use this lookaside capability or not.

How IMS Fast Path (VSO) Uses Data Spaces

Acquiring a Data Space

During control region initialization, IMS acquires two MVS data spaces, each of the maximum size of two gigabytes (2 147 483 648 bytes). One of the data spaces thus acquired is defined with the MVS `DREF` (disabled reference) option and the other data space is defined without the MVS `DREF` option. The difference between the two

data spaces is that with the MVS DREF option a combination of central storage and expanded storage is used while auxiliary storage is not used, whereas without the DREF option a combination of central storage, expanded storage (if available) and auxiliary storage is used.

Those VSO Areas that have been defined with the PRELOAD option are loaded into the data space which uses the DREF option. Those VSO Areas that have been defined with the NOPREL option are loaded into the data space which does not use the DREF option.

IMS acquires additional data spaces, both with DREF and without, as needed.

Accessing a Data Space

During IMS control region initialization, IMS calls DBRC to request a list of all the Areas that are defined as VSO (and those which need to be preopened [PREO] or preloaded [PREL]). All Areas defined with either PREOPEN or PRELOAD are opened at this time. Additionally, VSO Areas defined with PRELOAD are loaded into the 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 4KB.

Expressed in terms of the parameters of the DBDGEN AREA statement (see *IMS/ESA Utilities Reference: Database Manager*), this is:

```
(the value of the SIZE= parameter) ×
  (the value of the UOW=number1 parameter) ×
  (the value of the ROOT=number2 parameter)
```

rounded to the next 4KB.

The actual amount of space in a data space available for an area (or Areas) is 2 gigabytes (524,288 blocks, 4 KB each) minus an amount reserved by MVS (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. See *IMS/ESA Operator’s Reference* for sample output from this command.

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 aborted updates. Locking at the segment level is not supported for shared VSO areas. Only C1 locking is supported.

Preopen Areas and VSO Areas in a Data Sharing Environment

A VSO can be registered with any SHARELVL: SHARELVL(0) (exclusive access), SHARELVL(1) (one updater, many readers), SHARELVL(2), or SHARELVL(3) (block-level sharing).

SHARELVL(0)

In a data sharing environment, any SHARELVL(0) area with the PREOPEN option (including VSO PREOPEN and VSO PRELOAD) will be opened by the first IMS subsystem to complete its control region initialization. IMS will not attempt to preopen the area for any other IMS subsystem.

SHARELVL(1)

In a data sharing environment, a SHARELVL(1) area with the PREOPEN option will be preopened by all sharing IMS subsystems. The first IMS subsystem to complete its control region initialization will have update authorization; all others will have read authorization.

If the SHARELVL(1) area is a VSO area, it will be allocated to a data space by any IMS subsystem that opens the Area. If the area is defined as VSO PREOPEN or VSO PRELOAD, it will be allocated to a data space by all sharing IMS subsystems.

If the area is defined as VSO NOPREO NOPREL, it will be allocated to a data space by all IMS subsystems, as each opens the Area. The first IMS subsystem to access the area will have update authorization; all others will have read authorization.

SHARELVL(2)
A SHARELVL(2) area with at least one coupling facility structure name (CFSTR1) defined will be 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)
A SHARELVL(3) area with at least one coupling facility structure name (CFSTR1) defined will be shared at the block or control interval (CI) level within the scope of multiple IRLMs. Multiple IMS systems can be authorized for nonexclusive access.

You must take care when registering a VSO area as SHARELVL(1). Those subsystems that receive read-only authorization will never see the updates made by the read/write subsystem because all reads will come from the data space (not from DASD, where updates are eventually written).

Input / Output Processing

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 will be 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 utilizes 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 will go 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 will be 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.

I/O Error Processing

Using VSO increases the availability of data when write errors occur. Once 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 create 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 will receive 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, MVS abends.

For VSO Areas that have been defined with the `PRELOAD` options, the data is preloaded into the data space, so all read requests are satisfied from the data space. See "The PRELOAD Option" for a discussion of read error handling during the preload process.

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 will be used. When a write error to a structure occurs, the CI in error will be 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. If there is one structure defined and it reaches the maximum allowed, the Area is stopped and the structure is disconnected.

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 subsystem prior to the failure that caused the XRF takeover.

Emergency Restart Processing

Recovery of VSO Areas across IMS or MVS failures is similar to recovery of existing 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 uses data spaces to track VSO area updates: in addition to the data space resources used for VSO Areas, IMS 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 “recovery needed”.

At the end of emergency restart, IMS opens any Areas defined with the `PREOPEN` or `PRELOAD` options are opened, and Areas with the `PRELOAD` option are loaded into a data space. Preopening is done before dependent regions are enabled. Preloading is begun before dependent regions are enabled, but may run concurrently with the dependent regions. 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 VSO area removed from virtual storage by the /VUNLOAD command before the failure will be restored to the data space after the emergency restart.

VSO Options with XRF

During the tracking and takeover phases on the alternate subsystem, log records are processed in the same manner as during active subsystem emergency restart (from a previous active system checkpoint to the end of the log). The alternate subsystem uses the log records to determine which Areas have committed updates that were not written to DASD before the failure of the active subsystem. If any such committed updates are found, the alternate will REDO them, following the same process as for active subsystem emergency restart. See “Emergency Restart Processing” on page 238.

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 subsystem 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 will *not* be 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 will be able to access the area before preloading begins, but until preloading completes, some area read requests may have to be retrieved from DASD.

Chapter 8. Database Design Considerations for Fast Path

About This Chapter

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

This chapter examines the following design considerations:

- MSDB design considerations
- DEDB design considerations
- Using high-speed sequential processing
- Designing DEDB or MSDB buffer pools
- Designing a DEDB buffer pool in a DBCTL environment

MSDB Design Considerations

This section describes the choices you might need to make in designing an MSDB and proposes guidelines to help you make these choices.

You should 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 as follows:

```
(S * (L + 4)) + H + 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

```
H = C + (14 * F)
```

where:

C = 80 for non-related MSDBs without a terminal-related key
= 94 for the other types of MSDB

F = the number of fields defined in the DBD member

X = 2 if H is not a multiple of 4
= 0 if H is a multiple of 4

You need additional storage if the MSDB uses an LTERM name as a key:

```
4 * T * D
```

where:

T = the total number of LTERM names in the total online IMS system

D = the number of MSDBs using an LTERM name as a key (the sum of MSDBs that are terminal related and those that are non-terminal-related with terminal-related keys)

MSDBs reside in the MVS/ESA extended common storage area (ECSA).

Calculating Buffer Requirements

Details about calculating buffer requirements are in “Designing a DEDB or MSDB Buffer Pool” on page 257, along with other Fast Path buffer requirements. 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.

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. Since the MSDB is a key to good performance, it is important to understand it.

1. An MSDB record can be shared (S) by multiple users or be owned exclusively (E) by one user.

2. The same record can have both statuses (shared and exclusive) at the same time.
3. 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 are summarized in the following two tables.

Level	When	Duration
READ	GH with no update intent	VERIFY/get calls
	From call time until sync point (phase 1)[1]	Call processing
HOLD	GH with no update intent	At sync point, to reapply VERIFYs
	From call time until sync point (phase 1)[1]	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

Note:

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.

MSDB record status of shared (S) or owned exclusively (E):

		Enqueue level in program A		
		READ	HOLD	UPDATE
Enqueue level in program B	READ	S	S	E
	HOLD	S	E	E
	UPDATE	E	E	E

The above 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.

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 for 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 will be 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.

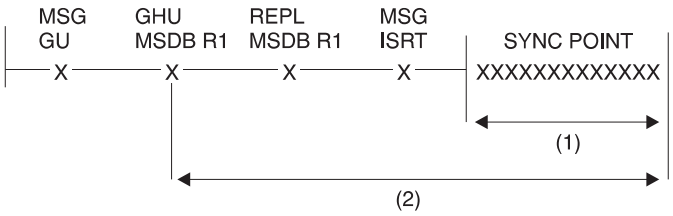
  In order to reduce the likelihood of deadlocks caused by resource contention, sync point processing enqueues (UPDATE level) MSDB resources in a defined sequence. This sequence is in ascending order of segment addresses. MSDB segments are acquired in ascending order of keys within ascending order of MSDB 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:



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

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.

It can represent a subset of the MSDBs currently on the MVS sequential data set identified by DD statement MSDBINIT. The opposite is not allowed and will result in an abend. The format is as follows:

```
DBD=dbname,NBRSEGS=xxxxxxxx[,F]

      where:
```

dbdname = the DBD name as specified during DBDGEN.

xxxxxxxx = 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 = the main storage required for the Fast Path extension of the
          CNTs (ECNTs).

This is derived from the following formula:

E = (20 + D * 4) * T bytes

      where:

      D = the number of MSDBs using logical terminal names as keys

      T = the total number of logical terminal names defined
          in the system

      P = the main storage required for all MSDBs as defined by
          the PROCLIB member DBFMSDBx.
```

DEDB Design Considerations

This section describes the choices you might need to make in designing a DEDB and proposes guidelines to help you make these choices.

Before designing the DEDB, you should understand several factors:

- How the application fits the limitations imposed by the DEDB itself
- How the application can make optimum use of the area concept of a DEDB
- Determining the size of the CI
- Determining the size of the UOW
- The DEDB randomizing routine
- Record deactivation
- Multiple copies of an area data set
- PCL (physical child last pointer)
- Subset pointers

In addition, DEDBs can be shared. For information on DEDB data sharing, see *IMS/ESA Administration Guide: System* and *IMS/ESA Utilities Reference: System*.

DEDB Design Guidelines

- Except for the relationship between a parent and its children, the logical structure (defined by the PCB) does not need to follow the hierarchic 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 a need later exists to relate SDEP segments to their roots, it is necessary to 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 will 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 will help differentiate them from the regular journal segments. This design prevents updates after the data has been converted into SDEP segments.

Considering the DEDB Area

The following are some reasons why DEDBs are divided into areas along with related design considerations:

- Database partitioning is required by the nature of the applications.

  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 could be given specific time slots. Their areas would then be 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, etc.) 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 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.

  Multiple copies of DEDB area data sets can be made to make data all the more available to application programs. See “Multiple Copies of an Area Data Set” on page 253.
- 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. To extend the independent overflow part of a DEDB, you must follow the procedures in “Extending DEDB Independent Overflow Online” on page 400.

  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 will change the area sequence number assigned to the areas. The subsequent log records written will 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 done 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.

- Because the area concept makes it possible to make an image copy of one area at a time, careful monitoring of the retention period of each log is necessary. Also, because the DEDB Direct Reorganization utility logs changes, no need exists to follow a reorganization run by an image copy run.
- 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.

Determining the Size of the CI

The choice of a CI size depends on the following factors:

- CI sizes of 512, 1K, 2K, 4K, and up to 28K bytes in 4K-byte 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, which 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 28K-byte 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 DEDB Direct Reorganization utility (DBFUMDR0) 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 used only by the utility.

   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, (explained later in this section). 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.

Although the following section pertains to programming, it is given here because it affects DEDB design, namely the size of the UOW.

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.

While 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 later 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.

DEDB Randomizing Routine Design

A DEDB randomizing module is required for placing root segments in a DEDB. The randomizing module is also required 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.

Refer to *IMS/ESA Customization Guide* for register usage and a sample randomizing program exit (DBFHDC40).

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.

Because of the area concept, some applications might decide to randomize in a particular area rather than through all the DEDB 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 abide by the same rules as described in *IMS/ESA Customization Guide*.

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 (multiple area data sets, MADS) of each ADS, making the data more available to application programs.

Each copy of an ADS 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 ADS. When an application program reads data from an area, Fast Path retrieves the requested data from any one of the available copies of the ADS. All copies of an ADS must have the same definition but can reside on different devices and on different device types. Using copies of ADS is also helpful in direct access device migration; for example, from a 3350 device to a 3380 device.

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.

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 will always be 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. For more information on defining and using subset pointers, see the section about Processing DEDBs with Subset Pointers in *IMS/ESA Application Programming: Database Manager*.

Restrictions: 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(s) 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.

High-Speed Sequential Processing (HSSP)

High-Speed Sequential Processing (HSSP) is a function of Fast Path that handles sequential processing of DEDBs.

Why HSSP?

Some reasons you may choose to use it are that, HSSP:

- Generally has a faster response time than regular batch processing.
- Optimizes sequential processing of DEDBs.
- Reduces program execution time.
- Typically produces less output than regular batch processing.
- Reduces DEDB updates and image copy operation times.
- Image copies can assist in database recovery.
- Locks at UOW level to ease ″bottle-necking″ of cross IRLM communication.
- Uses private buffer pools reducing impact on NBA/OBA buffers.
- Allows for execution in both a mixed mode environment, concurrently with other programs, and in an IRLM-using global sharing environment.
- Optimizes database maintenance by allowing the use of the image-copy option for an updated database.

More detailed information is included in the following subsections on HSSP.

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.
- HSSP processing and online utilities cannot process on the same area concurrently.
- Backward 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.
- 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.
- HSSP image copying is not allowed if PROCOPT ¬ =H.
- An ACBGEN must be done to activate the PROCOPT=H.
- H is compatible with all other PROCOPTs except for PROCOPT=O

Using HSSP

To use HSSP, you must specify a new PROCOPT option during PSBGEN, option 'H' see “HSSP Processing Option H (PROCOPT=H)” on page 256. 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 includes the image-copy option and the ability to set area ranges. To use these functions, you need one or more of the following:

- The SETR statement
- The SETO statement
- A DFSCTL data set for the dependent regions
- DBRC
- PROCOPT=H

High-Speed Sequential Processing (HSSP)

Related Reading: For more information about the SETR and SETO control statements, refer to *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

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. Its use 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 and/or SETR statements. H is compatible with any other Fast Path PROCOPT and PROCOPT=H can be used in one or more PCBs.

Related Reading: For information on PROCOPT=H rules see “Limitations and Restrictions When Using HSSP” on page 255.

H is compatible with any other Fast Path PROCOPT except for PROCOPT O. For more information on H processing, refer to *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

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 will 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 Reading: For information on DBRC databases and HSSP, and on created image copies, refer to the *IMS/ESA Operations Guide* and the *IMS/ESA DBRC Guide and Reference*.

For information on image copies and recovery, refer to *IMS/ESA Utilities Reference: System*.

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/ESA Administration Guide: System*.

Private Buffer Pools

Private buffer pools for the HSSP area are used for HSSP updates and image copies. HSSP does *not* impact NBA/OBA buffers. HSSP dynamically allocates up to three times the number of CIs per area in one UOW. Each buffer is a CI in size. The private buffer pools are located in ECSA/CSA.

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. The characteristics of the pool are defined at IMS definition time and can be overridden at IMS start-up time.

Three parameters characterize the Fast Path buffer pool:

- DBBF: Total number of buffers.

  The buffer pool is allocated at IMS start-up time in the common storage area (CSA) for System/390 environments and extended common storage area (ECSA) for MVS/ESA environments. During emergency restart processing, the entire buffer pool can be briefly page-fixed. You should consider the amount of available real storage when setting the DBBF value.
- DBFX: System buffer allocation.

  This is a set of buffers in the Fast Path buffer pool that is page fixed at start-up 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 28K bytes.

Buffer Requirements

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.

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.

Normal Buffer Allocation (NBA)

Fast path regions and IMS regions accessing Fast Path resources require this allocation to be specified in the region startup procedure. This allocation of buffers is used first and should be calculated to accommodate most of the transaction requirements. At the start of the region, the number of NBA buffers is page fixed in the Fast Path buffer pool.

Overflow Buffer Allocation (OBA)

This buffer allocation is optional and is used for exceptional buffer requirements when the normal buffer allocation (NBA) has been exhausted. Its use 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.

Fast Path Buffer Allocation Algorithm

Fast Path buffers are allocated on demand up to a limit specified at the start of the 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). 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, 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.

System Buffer Allocation (DBFX)

The system buffer allocation (DBFX) is needed, because DEDB writes are deferred until after sync point processing. 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 DBFX allocation of buffers is page fixed at the start of the first region specifying an NBA request.

Determining the Fast Path Buffer Pool Size

The number of fast path buffers (DBBFs) required is calculated using the following formula:

```
DBBF ≥ A + N + OBA + DBFX
```

where:

- 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

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 in accordance with the formula given in the above section 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.

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.

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. See “Processing Option P (PROCOPT=P)” on page 252 for a discussion of this subject.

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. Wait-for-buffer conditions could be alleviated by specifying a larger DBFX value.

Designing a DEDB or MSDB Buffer Pool

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

Buffers needed to fulfill requests from database calls are obtained from a global pool called the Fast Path buffer pool. The characteristics of the pool are defined at IMS definition time and can be overridden at IMS start-up time.

Three parameters characterize the Fast Path buffer pool:

1. DBBF: Total number of buffers.

   The buffer pool is allocated at IMS startup time in the common storage area (CSA) for System/390 environments and extended common storage area (ECSA) for MVS/ESA environments.
2. 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.
3. 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 28K bytes.

Buffer Requirements

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.

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.

Normal Buffer Allocation for BMPs

BMP regions accessing Fast Path resources require this allocation to be specified in the region start-up procedure. The start-up parameter is already specified as NBA. This allocation of buffers is used first and should be calculated to

accommodate most of the transaction requirements. At the start of the region, the number of NBA buffers is page fixed in the Fast Path buffer pool.

Normal Buffer Allocation for CCTL Regions and Threads

CCTL (coordinator control) regions, requiring fast path resources, need the following parameters specified in the DRA start-up table:

- CNBA
- FPB

CNBA is the normal buffer allocation of each active CCTL region. FPB is 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/ESA Administration Guide: System*.

Overflow Buffer Allocation for BMPs

This buffer allocation is optional and is used for exceptional buffer requirements when the NBA 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 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.

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 start-up 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. If FPB cannot be satisfied out of CNBA, the schedule request fails.

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.

System Buffer Allocation (SBA)

This allocation is needed 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 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 DBFX 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

The number of buffers required is calculated using the following formula:

```
DBBF ≥ A + N + LO + DBFX + CN
```

where:

- 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

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 in accordance with the formula given in the above section 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.

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.

The NBA/FPB Limit and Sync Point

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. See “Processing Option P (PROCOPT=P)” on page 252 for a discussion of this subject.

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. This could be alleviated by specifying a larger DBFX value.

Designing a DEDB Buffer Pool in the DBCTL Environment

Consider the special case: The BMP region loads or processes a DEDB and is 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.

A Note on Fast Path Buffer Allocation in IMS Regions

IMS regions that access Fast Path resources must have the NBA and OBA parameters specified in their start-up 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 9. Developing Your Test Database

About This Chapter

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 The following six IBM programs can help you develop your test database:

Data Extraction, Processing, and Restructuring System

Cross System Product/370 Application Development (CSP/370AD)

DL/I Test Program, DFSDDLT0

IMS System Utilities/Database Tools (DBT)

DB/DC Data Dictionary

You can find guidance information about application program testing in *IMS/ESA Application Programming: Design Guide* . For information about testing an online system, see *IMS/ESA Administration Guide: System*.

This chapter examines two areas of test development:

Understanding test requirements

Designing, creating, and loading a test database

Understanding Test Requirements

Depending on your system configuration, user requirements, and the design characteristics of your database and data communication systems, 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 Data Extraction, Processing, and Restructuring System.
- 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 Data Extraction, Processing, and Restructuring System or DBT.
- 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 DBT 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 following sections of 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.

Ways to Design, Create, and Load a Test Database

You can develop a test database just as you would develop a production database. With that approach, you perform the tasks described throughout the other chapters of this manual, 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

The IMS test system is discussed in *IMS/ESA Administration Guide: System* .

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.

Using the Data Extraction, Processing, and Restructuring System

You can use this system (Program Number: 5796-PLH) to access a wide variety of data and restructure it into a test database. By means of control statements, you define the source and target files and specify the structure of the target files.

The data restructuring phase of the system allows you to:

- Combine components of different files to form new files
- Restructure a file to form different files
- Rearrange data within a file
- Alter values according to your needs
- Form hierarchies
- Decrease or increase the number of levels in a hierarchy

Details about using this system are in *Data Extraction, Processing, and Restructuring System, Program Description/Operations Manual*·

Using the IMS Application Development Facility II

If your installation uses CSP/370AD to develop application programs, you can use it to create a simple test database. The interactive nature of ADF 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. For information on how to use CSP/370AD, see the *Cross System Product/370 Application Development Guide*·

CSP/370AD supports both IMS and CICS.

Using the DL/I Test Program, DFSDDLT0

If you need a test database with relatively few database records, for example, you can use DFSDDLTO 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. Information about this test program is in "Testing an Application Program," in *IMS/ESA Application Programming: Design Guide*.

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.

Using IMS System Utilities/Database Tools

The IMS System Utilities/Database Tools (DBT) is a collection of utility programs designed to help you manage and monitor your IMS databases. DBT supports both full-function and Fast Path databases.

In addition to aids that help you analyze the design of your production database, DBT includes an aid to help you create a test database. You can use the DB Segment Restructure utility to create a test database by structuring segments and hierarchies based on a subset of the actual database. You do this by supplying a beginning SSA (or it will begin at the first segment) and a key feedback compare string as an ending comparative value (or it will continue to the end of the database).

Information on the DB Segment Restructure utility is in *DBT DB Segment Restructure User's Guide*· For information on all of the utilities provided with DBT, see *IMS System Utilities/Database Tools (DBT) General Information Manual*·

Using the DB/DC Data Dictionary

The IBM DB/DC Data Dictionary is a development and administrative tool used to manage information about an installation's data processing resources. If you have your production database defined in the Data Dictionary, you have the means available to define your test database. You can search the definitions in the dictionary to determine which elements to include in your test database. Then, you can copy selected parts of the Data Dictionary definitions and assign the copy a test status. This step can become the definition of your test database.

You can also use the DBD and PSB generation facility of the Data Dictionary to provide DBDs and PSBs for the test database. You cannot use the Data Dictionary to define DEDBs or MSDBs, or PSBs naming either of these types of databases.

The DB/DC Data Dictionary (Program Number: 5740-XXF) is described in *DB/DC Data Dictionary General Information Manual*·

Using the DataAtlas for OS/2 Data Dictionary

DataAtlas for OS/2 is a development and administrative tool that manages your data processing information in a LAN-based environment. DataAtlas is a component

of the IBM VisualGen Team Suite set of products. It provides facilities for versioning, model extensibility, constraint checking, and an SQL query capability. DataAtlas also provides the following:

- Object-oriented architecture provides for efficient navigation of repository-controlled information and positions DataAtlas to take full advantage of emerging object-oriented methodologies.
- Easy-to-use GUI design optimizes productivity for data analysts, database administrators, and application programmers.
- Data and database modeling helps you optimize and tune your enterprise data and database designs.
- Application system documentation support facilitates a better understanding of your organization's application portfolio.

For managing your data definitions, DataAtlas allows you to do the following:

- Work from a central point of control and standardization on data definitions for the IBM family of hierarchical databases (IMS DB), relational databases (DB2 for MVS/ESA, DB2/2), and high-level languages (COBOL, PL/I). Oracle and Sybase databases are also supported.
- Populate data definitions into a centralized repository.
- Migrate definitions from the OS/VS DB/DC Data Dictionary.
- Create new data definitions and update existing data definitions.
- Easily identify and eliminate redundant data definitions.
- Optimize the reuse of existing data definitions to facilitate the development of new applications.
- Perform impact analysis to track change activities and to understand the impact of the changes.
- Validate data definitions prior to placing them into your production environment.
- Generate data definitions for your production environment.

Chapter 10. Establishing Standards and Procedures

About This Chapter

This chapter examines the following areas:

- Standards and procedures
- Establishing naming conventions
- Using the dictionary to enforce and control standards and procedures

Standards and Procedures

You should give careful thought to developing standards and procedures for your database system. Providing standards and procedures results in:

- Improved quality of application systems (because setting up and following standards and procedures gives you greater control over your entire application development process)
- Improved productivity in application and database design (because guidelines for design decisions exist)
- Improved productivity of application coding (because coding standards and procedures exist)
- Better communication between you and application developers (because both of you have clearly defined responsibilities)
- Improved 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.

In the area of *database design*, for example, you can establish standard practices for handling the following items:

- 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 index organization using VSAM
HIDAM pointer options at the root level
Sequencing twin chains
Use of HD free space
When to use HDAM
Processing an HDAM database sequentially
Use of the “byte limit count” for HDAM
Use of twin backward pointer for HDAM roots
Use of free space with HDAM
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

In the area of *application programs' use of the database*, establish standards for the following:

- Putting update and read functions 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 permitted 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 (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)

In the area of *application design*, establish procedures to govern the following:

- The interaction between you and the application designer
- Use of the dictionary or COPY or STRUCTURE libraries for data elements and structures
- The holding of design reviews and inspections

In the area of *batch operations*, you can 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/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 all problems are recorded and reported to the responsible parties
- Normal operating procedures. These operating procedures include operations schedules, cold start, warm start, shutdown procedures, 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. These procedures should include emergency restart, database backout, database recovery, log recovery, and batch program restart. For a more complete treatment of recovery procedures, see *IMS/ESA Operations Guide* and *IMS/ESA Sample Operating Procedures* .
- A master terminal operator’s guide for the installation. This guide should be supplemented by *IMS/ESA Operator’s Reference* .
- A master operations log. This log could contain a record of system availability, time and type of failure, and 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, status of APARs and date submitted, and bypass solutions.

Establishing Naming Conventions

Good naming conventions are mandatory in a data processing project, especially in an environment with multiple applications. Some general rules to follow in setting up naming conventions are:

- Each name should be unique.
- Each name should be meaningful and identifiable. You should be able to identify the type of thing being referred to by its name.

Table 11 on page 274 is an example of minimal naming conventions that are also compatible with naming conventions and requirements of the DB/DC Data Dictionary. They are presented only as an example, and you can establish your own

naming conventions.

Table 11. Example of Naming Conventions

CATEGORY	CONVENTION
SYSTEM	S as first letter
JOB	J as first letter
PROGRAM	P as first letter if IMS program (to match PSB) G as first letter otherwise
MODULE	M as first letter
COPY	C as first letter for member containing the segment structure A as first letter for member containing all the SSAs for the segment Remainder must be the same as the segment name
TRANSACTION	T as first letter
PSB	P as first letter
PCB	Same name as PSB **Note:** Occurrence number indicates position in PSB
DATABASE	Dtaaann **Where** **Indicates** **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** Makes the name unique if there are multiple logical or secondary index databases
SEGMENT	**Saaabbbb** **aaa** The 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** The user name **R** First letter for 'segments' that are non-DL/I file record definitions **O** First letter for any other data areas, for example, terminal I/O areas, control blocks, report lines etc.)
ELEMENT	E as first letter

Using the Dictionary to Enforce and Control Standards and Procedures

Making the dictionary an integral part of your application development process automatically helps you enforce standards and procedures. The dictionary acts as a control mechanism in the following ways:

- You require all data element names to be entered in the dictionary, and you can easily enforce naming conventions.
- You can use dictionary reports to tell whether certain database practices (such as program-to-database standards) are being followed.
- You ensure that standards relating to such things as PCB masks and segment structure declarations are followed by requiring that programmers get their data structures from the dictionary.

Setting up procedures involving use of the dictionary makes the entire application development process more disciplined, controlled, and efficient.

Chapter 11. Implementing Your Database Design

About This Chapter

After you have designed your databases and before application programs can use them, you must tell IMS their physical and logical characteristics by coding and generating a DBD (database description) for each database.

Before an application program can use the database, you must tell IMS the application program's characteristics and use of data and terminals. You tell IMS the application program characteristics by coding and generating a PSB (program specification block).

Finally, before an application program can be scheduled for execution, IMS needs the PSB and DBD information for the application program available in a special internal format called an ACB (application control block).

This chapter examines the following areas of implementing your database design:

- Coding database descriptions
- Coding program specification blocks
- Building application control blocks

Coding Database Descriptions as Input for DBDGEN the Utility

A DBD is 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.

Figure 128 illustrates the DBD generation process. Figure 129 shows the input to the DBDGEN utility. Separate input is required for each database being defined.

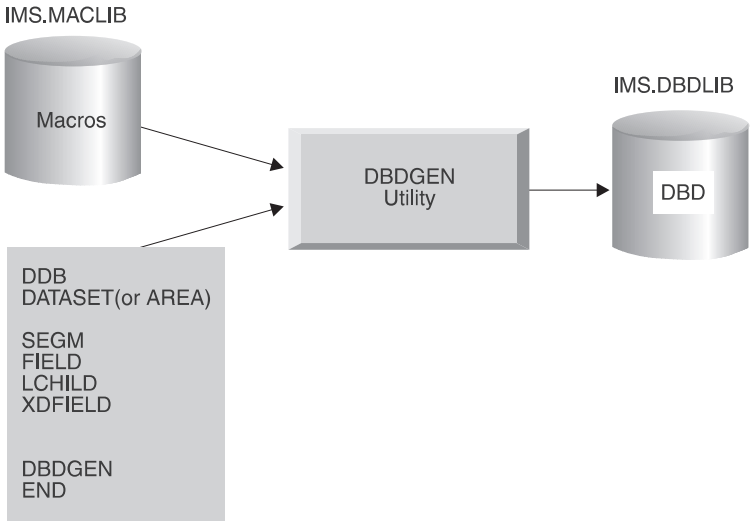


Figure 128. The DBD Generation Process

```
//DBDGEN    JOB  MSGLEVEL=1
//                 EXEC        DBDGEN,MBR=APPLPGM1
//C.SYSIN        DD     *

    DBD                 required for each DBD generation
    DATASET(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
    XDFIELD             required for each secondary index relationship
        .
        .
        .
    DBDGEN              required for each DBD generation
    END                 required for each DBD generation
/*
```

Figure 129. Structure of DBD Generation Input

The DBD Statement

In the input, the DBD statement names the database being described and specifies its organization. Only one DBD statement exists in the input deck.

The DATASET Statement

This 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 DATASET groups can be defined. Each DATASET statement is followed by the SEGM statements for all segments to be placed in that data set group.

If the database is a DEDB, the AREA statement is used instead of the DATASET statement. The AREA statement defines an area in the DEDB. Up to 240 AREA

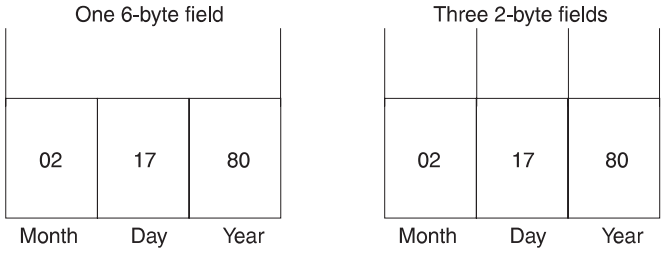
statements can be used to define multiple areas in the database. All AREA statements must be put between the DBD statement and the first SEGM statement.

The SEGM Statement

This statement defines a segment type in the database, its position in the hierarchy, its physical characteristics, and its relationship to other segments. SEGM statements are put in the input deck in hierarchic sequence, and a maximum of 15 hierarchic levels can be defined. The number of database statements allowed depends on the type of database. SEGM statements must immediately follow the DATASET or AREA statements to which they are related.

The FIELD Statement

This statement defines a field within a segment type. FIELD statements must immediately follow the SEGM statement to which they are related. A FIELD statement is required for all sequence fields in a segment and all fields the application program can refer to in the SSA of a DL/I call. A FIELD statement is also required for any fields referenced by a SENFLD statement in any PSB. To save space, do not generate FIELD statements except in these circumstances. FIELD statements can be put in the input deck in any order except that the sequence field, if one is defined, must always be first. Up to 255 fields can be defined for each segment type, and a maximum of 1000 fields can be defined for each database. 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.



This technique allows application programs to access the same piece of data in a variety of ways. To access the same piece of data in a variety of 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.

The LCHILD Statement

The LCHILD statement defines a secondary index or logical relationship between two segment types, or the relationship between a HIDAM index database and the root segment type in the HIDAM 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.

The XDFLD Statement

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 field used to create the secondary index from the source segment

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.

The DBDGEN and END Statements

One DBDGEN statement and one END statement is put at the end of each DBD generation input deck. These specify:

- The end of the statements used to define the DBD (DBDGEN)
- The end of input statements to the assembler (END)

Detailed instructions for coding DBD statements and examples of DBDs are contained in *IMS/ESA Utilities Reference: System*.

Using the DB/DC Data Dictionary

If you have the DB/DC Data Dictionary and have defined your databases in it, you can use the dictionary `DBD_OUT` command to produce the statements needed for DBD generation in card image format. Detailed information on how to do this is contained in *DB/DC Data Dictionary Terminal User's Guide and Command Reference*.

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.

Figure 130 shows the PSB generation process. Figure 131 on page 281 shows the structure of the deck used as input to the PSBGEN utility.

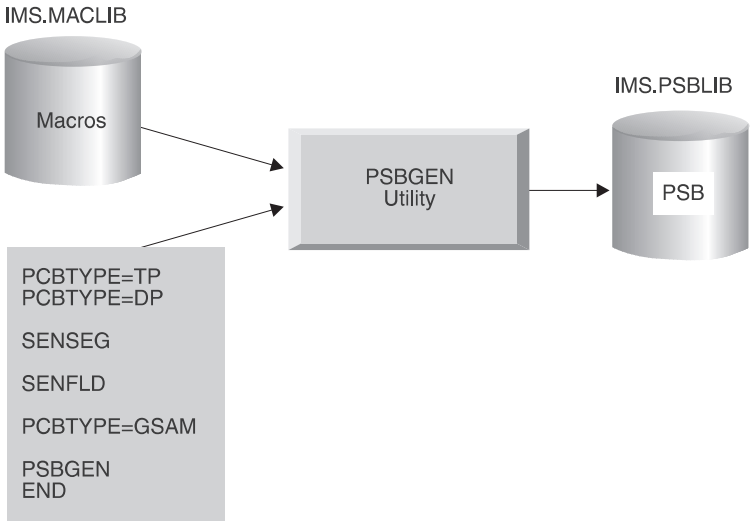


Figure 130. The PSB Generation Process

```
//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
/*
```

Figure 131. Structure of PSB Generation Input

The Alternate PCB

Two types of PCB statements can be placed in the input deck. The first type, called the *alternate* PCB, 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/ESA Administration Guide: System*.

The Database PCB Statement

The second type of PCB statement is called the *database* PCB statement. Database PCB statements define the DBD of the database the application program will access. The 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 the application program accesses. In each PSB generation, up to 255 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.

The SENSEG Statement

This 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 Figure 132 on page 282 if D is defined as a sensitive segment for an application program, B and A must also be defined as sensitive segments.

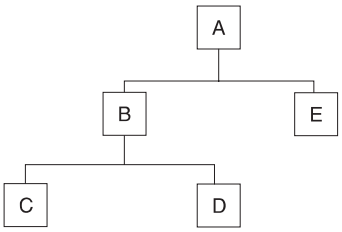


Figure 132. Example of a SENSEG Relationship

To keep an application program from becoming sensitive to a specific segment in the path to a lower level segment, code a K in the PROCOPT = keyword. Coding a K in the PROCOPT = keyword in the SENSEG key statement of the segment you do not need the application program to see, gives application program sensitivity to only that 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 3000 SENSEG statements can be defined for each PSB generation.

The SENFLD Statement

This statement is used only in parallel with field-level sensitivity. It defines the fields in a segment type to which the application program is sensitive. This 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/ESA Utilities Reference: System* .

Using the DB/DC Data Dictionary

If you have the DB/DC Dictionary and have put PSB information in it, you can use the dictionary PSB_OUT command to produce the statements needed for PSB generation in card image format. Detailed information on how to do this is contained in *DB/DC Data Dictionary Terminal User's Guide and Command Reference*·

Building the Application Control Blocks (ACBGEN)

IMS builds the ACB with the ACBGEN utility by merging information from the PSB and DBD. For execution in a batch environment, IMS can build ACBs either dynamically (PARM=DLI), or it can prebuild them using the ACB maintenance utility (PARM=DBB). ACBs *must* be prebuilt for use by online application programs. The ACB generation process is shown in Figure 133.

Coding Program Specification Blocks as Input to the PSBGEN Utility

ACBs cannot be prebuilt for GSAM DBDs. However, ACBs can be prebuilt for PSBs that reference GSAM databases.

The ACB maintenance utility (ACBGEN), shown in Figure 133, gets the PSB and DBD information it needs from IMS.PSBLIB and IMS.DBDLIB.

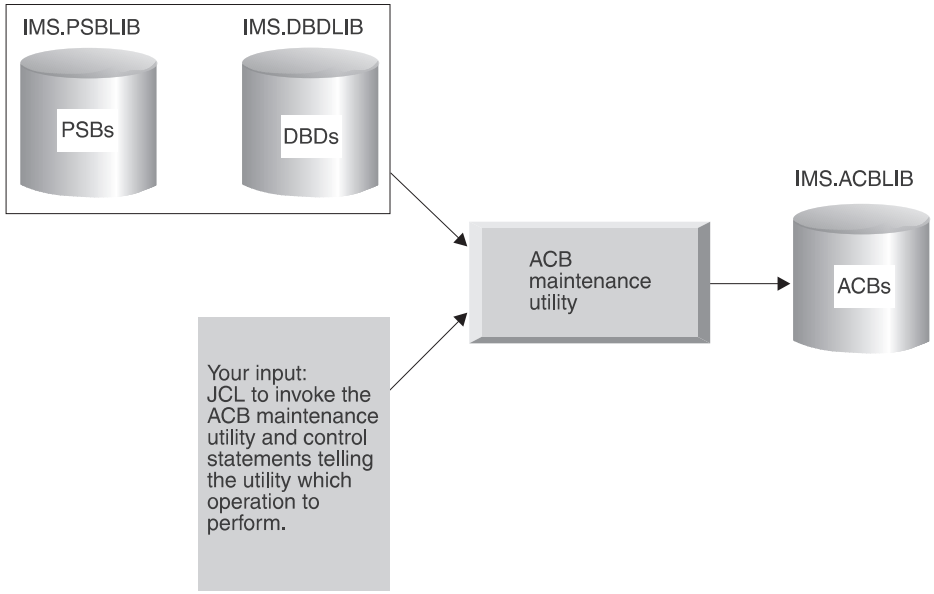


Figure 133. The ACB Generation Process

You can have the utility 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. (IMS.ACBLIB 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. Another advantage of using the ACB maintenance utility is the initial error checking it performs. It checks for errors in the names used in the PSB and the DBDs associated with the PSB and, if erroneous cross-references are found, prints appropriate error messages.

IMS.ACBLIB has to be used exclusively. Because of this, the ACB maintenance utility can only be executed 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 ACB maintenance utility.

You can change ACBs or add ACBs in an "inactive" copy of ACBLIB and then make the changed or new members available to an active IMS online system by using the online change function. "Using the Online Change Function" in "Chapter 15. Modifying Your Database" on page 365 describes how you effectively change ACBLIB for an online system.

Detailed instructions for running the ACB maintenance utility and examples of its use are contained in the *IMS/ESA Utilities Reference: System* .

Generated Program Specification Blocks

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 section of the *IMS/ESA Installation Volume 2: System Definition and Tailoring* .

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.

Chapter 12. Loading Your Database

About This Chapter

Once 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. This chapter examines the following areas of loading a database:

- Estimating the minimum size of the database
- Allocating data sets
- Writing the load program

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 section 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.

Step 1. Calculate the Size of an Average Database Record

First, 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.

The prefix portion of the segment depends on the segment type and on the options you are using. Table 12 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. (“Chapter 4. Designing a Fast Path Database” on page 33 under “Mixing Pointers” contains an illustration of a segment’s prefix.)

Table 12. Required Fields and Pointers in a Segment’s Prefix

Type of segment	Fields and pointers required 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 and HIDAM	PCF pointer	4
	PCL pointer	4
	PP pointer	4
	PTF pointer	4
	PTB pointer	4
	HF pointer	4
	HB pointer	4

Table 12. Required Fields and Pointers in a Segment's Prefix (continued)

Type of segment	Fields and pointers required in the segment's prefix	Size of the field or pointer (in bytes)
DEDB	PCF pointer	4
	PCL pointer	4
	Subset pointer	4
Logical parent	LCF pointer	4
	LCL pointer	4
	Logical child counter	4
Logical child	LTF pointer	4
	LTB pointer	4
	LP pointer	4
Secondary index	Direct-address pointer to the target segment	4

The following information shows an example of calculating prefix size:

If you have an HDAM database and the segment type whose prefix size you are calculating uses PCF and PCL pointers, then you need to add:

One byte for the segment code

One byte for the delete byte

Four bytes for each PCF pointer

Four bytes for each PCL pointer

After you calculate these numbers together, add the result to the size of the data portion of the segment type. This step gives you the total size of the segment type.

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 Figure 134 on page 288, 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.

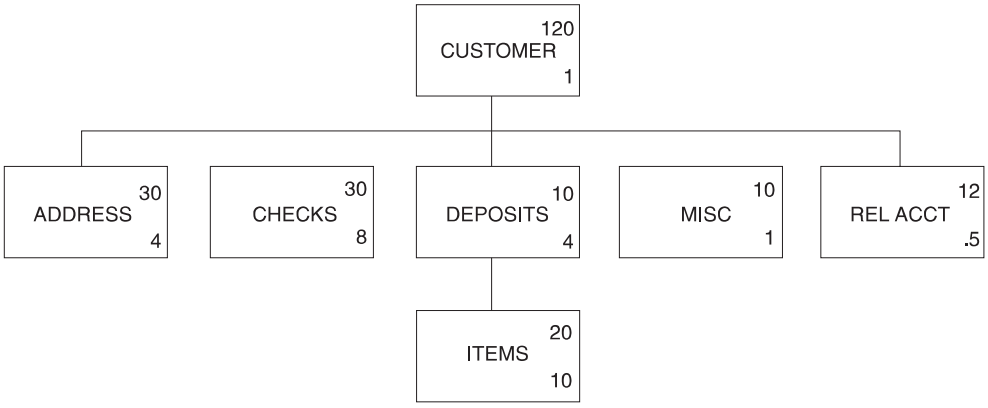


Figure 134. Segment Sizes and Average Segment Occurrences

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

Now that you have determined segment size and segment frequency, you can determine the average size of a database record. To determine average database record size, multiply segment size and segment frequency together for each segment type in the database record, then add the results. For example, in the database record you have been looking at, average database record size is calculated as follows:

Segment Type	Segment Size		Average Occurrences	Result
CUSTOMER	120	x	1	= 120
ADDRESS	30	x	4	= 120
CHECKS	30	x	8	= 240
DEPOSITS	10	x	4	= 40
ITEMS	20	x	40 (10x4)	= 800
MISC	10	x	1	= 10
REL ACCT	12	x	.5	= 6

Note: Average database record size = 1336

Step 2. Determine Overhead Needed for DEDB CI resources

If you are not using VSAM, you can skip this step. 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.

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 are as follows:

Fields	Size of field (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 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 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 section 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: Determining the Number of CIs or Blocks Needed

With HIDAM, 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, 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 in 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.
3. Continuing our example, because the average size of a database record is 1336 bytes, you need 668000 bytes (500 x 1336) to store data. To determine the number of logical records needed to hold the database, divide the available logical record space into the total data space needed: 668000/2037 = 328 logical records.

   Because one logical record exists per CI in HIDAM, you need a minimum of 328 CIs of 2048 bytes each for the ESDS.

HIDAM Index: Calculating the Space Needed

Calculating space for a HIDAM 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: Determining the Amount of Space Needed

Because of the many variables in HDAM, 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 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 bit map is in the second CI.

With HDAM, 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.) 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 and HIDAM databases, you can allocate free space when your database is initially loaded. Free space is explained in “Chapter 4. Designing a Fast Path Database” on page 33, “Specifying Free Space”. 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:

$$A = B \times \frac{fbff}{fbff-1} \times \frac{100}{100 - fspf}$$

where:

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 (what you specified in fbff operand in the DBD)

fspf = the minimum percentage of each block or CI you are leaving as free space (what you specified in the fspf operand in the DBD)

Step 5. Determine the Amount of Space Needed for Bit Maps

In HDAM and HIDAM databases, you need to add the amount of space required for bit maps to your calculations. Bit maps are explained in “Chapter 4. Designing a Fast Path Database” on page 33 under “General Format of HD Databases and Use of Special Fields”. To calculate the number of bytes needed for bit maps in your database, you can use the following formula:

$$A = \frac{D}{(B-C) \times 8}$$

where:

A = The number of bit map 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 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 bit map.

D = The number of CIs or blocks in your database.

You need to add the number of CIs or blocks needed for bit maps to your space calculations.

Allocating Data Sets

Once you have determined how much space you will need for your database, you can allocate data sets and then load your database. VSAM data sets can be allocated using the `DEFINE CLUSTER` command. Use of this command is described in *MVS/DFP Access Method Services for VSAM Catalog.*·

Attention: If you plan to use the Database Image Copy 2 utility to take image copies of your database, the data sets must be allocated on hardware that supports the DFSMS concurrent copy function.

When loading databases that contain logical relationships and/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 MVS JCL. You can use the MVS program IEFBR14 to preallocate data sets, except when the database is an MSDB. For MSDBs, you should use the MVS program IEHPROGM.

Allocating OSAM Data Sets

At the time the data set is loaded, you should use JCL to allocate OSAM data sets. This mode of allocation can be for single or multiple volumes, using the SPACE parameter.

If the installation control of direct-access storage space and volumes require that the OSAM data sets be pre-allocated, or if a message queue data set requires more than one volume, the OSAM data sets might be pre-allocated.

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 pre-allocated 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 pre-allocated space quantity. However, queue data sets can have multivolume allocation.

- 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 *multiple-volume* data set is pre-allocated, 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. *Do not* use IEFBR14 and specify a DD card with a multivolume data set, because this allocates an extent on only the first volume.

Restriction: Do **not** use this technique to allocate multi-volume OSAM databases on which you intend to use the Image Copy 2 utility (DFSUDMT0). All multi-volume databases on which the Image Copy 2 utility is used **must** be allocated using the standard DFP techniques.

Example of Allocating an OSAM Data Set

```
//OSAMALLO   JOB  A,OSAMEXAMPLE
//S1        EXEC  PGM=IEFBR14
//SYSPRINT    DD  SYSOUT=A
//EXTENT1     DD  VOL=SER=AAAAAA,SPACE=(CYL,(20,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
//S2        EXEC  PGM=IEFBR14
//SYSPRINT    DD  SYSOUT=A
//EXTENT2     DD  VOL=SER=BBBBBB,SPACE=(CYL,(30,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
      .
      .
      .
//LAST      EXEC  PGM=IEFBR14
//SYSPRINT    DD  SYSOUT=A
//EXTENTL     DD  VOL=SER=LLLLLL,SPACE=(CYL,(30,5)),UNIT=3390,
//                DSN=OSAM.SPACE,DISP=(,KEEP)
```

Cautions When Allocating OSAM Data Sets

1. Pre-allocating more volumes for OSAM data set extents than are used during initial load or reload processing might cause the following condition to occur:

   If the initial load or reload step did not result in the data being written to the last volume of the pre-allocated data set, and secondary allocation was not specified during data set pre-allocation, then any attempt to extend the data set beyond the last volume written to at initial load or reload time causes an abend.

2. It is recommended that you not reuse multivolume OSAM data sets without first scratching the data set and then reallocating the space. Failure to do this 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.

3. When loading this database, the order of the DD cards determines the order in which the data is loaded
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 DFP techniques and **not**with the method recommended in “Allocating OSAM Data Sets” on page 293.

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 (see Figure 135 on page 297), the load program builds segments for a database record and inserts them into the database in hierarchic order. If the data to be loaded into the database already exists in one or more files (see Figure 136 on page 298), 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.

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.

Recommendation: If a user load program using PROCOPT=L|LS is running in a DLI or DBB region, DBRC authorization is required for all databases logically related to the one being loaded.

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, and HIDAM, 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 hierarchic 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 hierarchic sequence (children after their parents, database records in order of their key fields).

If you are loading an HDAM 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.

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 these 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 hierarchic sequence.

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 and HIDAM, 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 sections describe both types of load programs.

Figure 135 on page 297 shows the load process.

Figure 136 on page 298 illustrates loading a database using existing files.

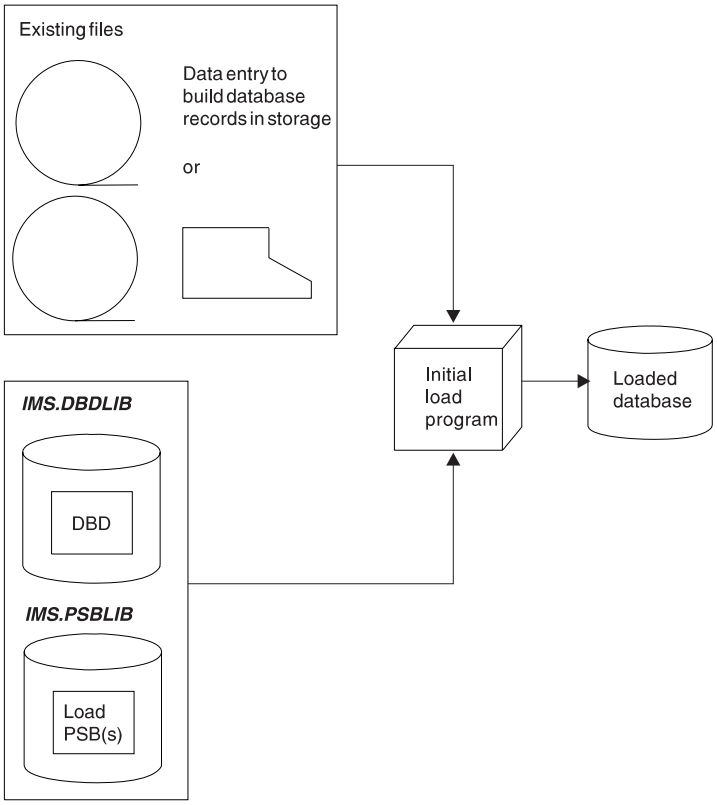


Figure 135. The Load Process

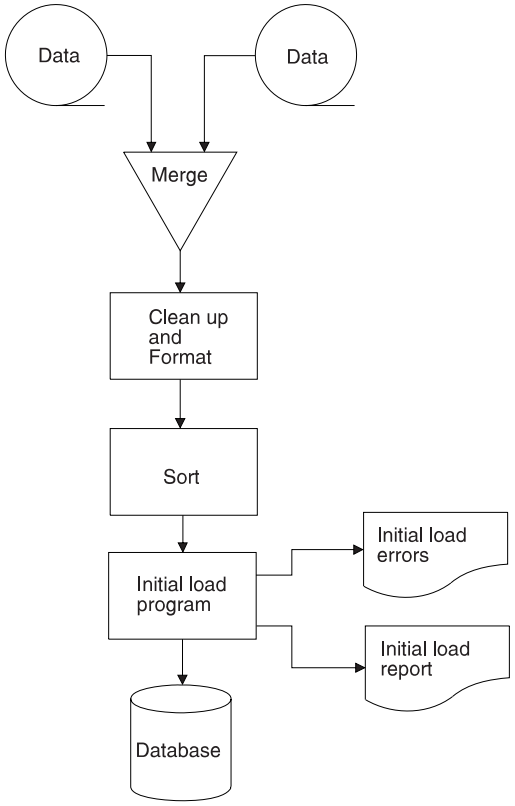


Figure 136. Loading a Database Using Existing Files

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.

Figure 137 on page 299 shows the logic for developing a basic initial load program. Following Figure 137 is a sample load program (Figure 138) 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.

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. See *IMS/ESA Utilities Reference: Database Manager*for a description of how DEDBs are loaded.

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. An example is

  ```
  DBD=MSDB0001,NBRSEGS=200[,F]
  ```

If the “F” is omitted, the MSDB can be paged.

For a description of the record format and the DBD keyword parameters, see the section “Member Name IMS” in *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

- 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. (For a description of the record format and information on how to use this utility, see *IMS/ESA Utilities Reference: Database Manager*.)

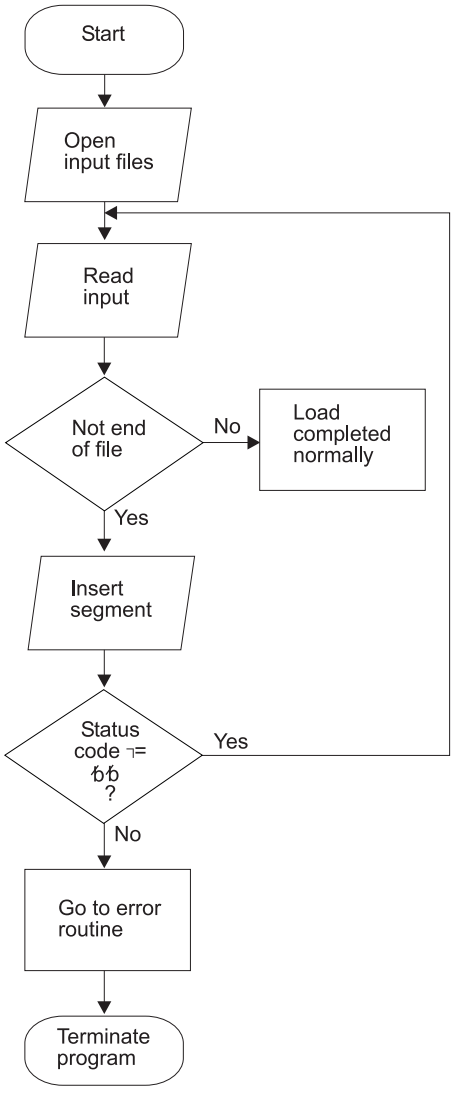


Figure 137. Basic Initial Load Program Logic

```
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
```

Figure 138. Sample Load Program

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.

Writing a Load Program

Restartable load programs differ from basic load programs in their logic. Figure 139 on page 302 shows the logic for developing a restartable initial load program. 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.

Detailed guidance information on what must be done to run a restartable load program under the control of UCF is contained in *IMS/ESA Utilities Reference: Database Manager*.

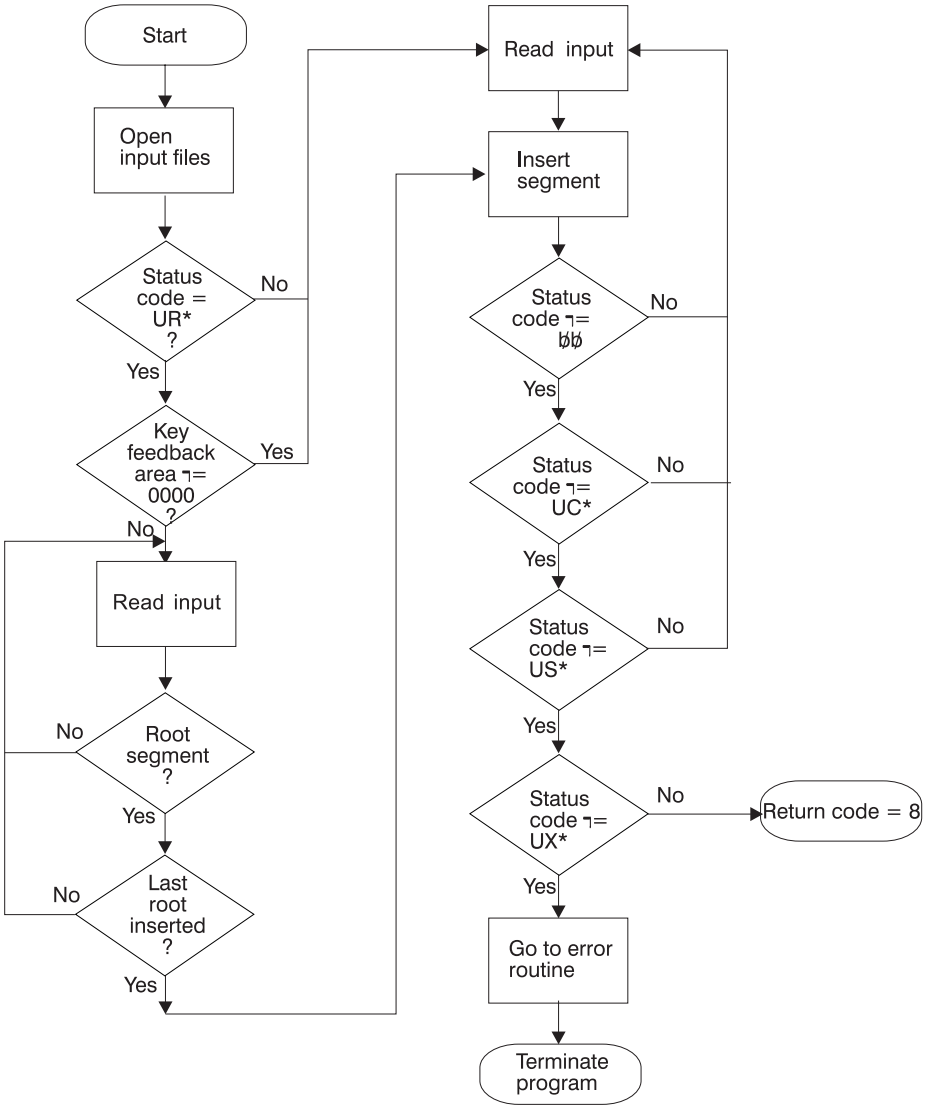
To make your initial database load program restartable under UCF, consider the following points when it is being planned and written.

- 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.



*UR = load program being restarted under control of UCF
UC = checkpoint record written to UCF journal data set
US = initial load program preparing to stop processing
UX = checkpoint record was written and processing stopped

Figure 139. Restartable Initial Load Program Logic

```
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
```

Figure 140. Sample Restartable Initial Load Program (Part 1 of 2)

```
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
```

Figure 140. Sample Restartable Initial Load Program (Part 2 of 2)

JCL for the Initial Load Program

Following is the JCL you will need to initially load your database. The //DFSURWF1 DD statement is present only if a logical relationship or secondary index exists.

```
//          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.RESLIB
//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 hierarchic 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 Database

In an HDAM 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 hierarchic sequence. PROCOPT=L (for load) or PROCOPT=LS (for load segments in ascending sequence) must be specified in the PCB.

Loading a HIDAM Database

To load a HIDAM database, you must present root segments in ascending key sequence and all dependents of a root should follow the root in hierarchic sequence. PROCOPT=LS (for load segments in ascending sequence) must be specified in the PCB.

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 them to put the correct pointer information in each segment's prefix. These reorganization utilities are described in "Chapter 14. Tuning Your Database" on page 323.

Loading Fast Path Databases

This section describes how to load MSDBs, DEDBs, and sequential dependent segments.

Loading an MSDB

Because MSDBs reside in main storage, you do not load them as you do other IMS databases, that is, by means of a load program that you provide. Rather, they are loaded during system initialization, when they are read from a data set. You first build this data set either by using a program you provide or by running the MSDB maintenance utility. See *IMS/ESA Utilities Reference: Database Manager* for information on how to use this utility. See Figure 106 on page 197 for the record format of the MSDBINIT data set.

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 240 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.

Chapter 13. Monitoring Your Database

About This Chapter

This chapter describes a number of tools and aids you can use to monitor performance and use of space in your database:

- DB Monitor
- //DFSSTAT Reports
- `AMS LISTCAT` command [2]
- IEHLIST [2]
- HD Reorganization Unload utility (DFSURGU0)
- HISAM Reorganization Unload utility (DFSURRL0)
- DL/I Test Program (DFSDDLT0) call
- Database Surveyor utility (DFSPRSUR)
- IMS/VS System Utilities/Database Tools (DBT)
- Batch Terminal Simulator
- IMS Monitor Summary and System Analysis Program II (IMSASAP II)

- STAT call

Information on using the IMS Monitor is found in *IMS/ESA Administration Guide: System*. (If you are sharing data, additional information about monitoring is found in *IMS/ESA Administration Guide: System* under “Administration of Systems That Share Data”.)

This chapter examines the following areas of monitoring a database:

- Using the database monitor
- Using database monitor aids.

Using the Database Monitor

The database (DB) 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 DB 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 Figure 141 on page 312.

- The DB Monitor program
- The DB Monitor Report print program (program name is DFSUTR30) (A similar set of reports describing online use can be generated by running the IMS Monitor Report print program, whose program name is DFSUTR20.)

The DB Monitor program 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.

2. These two functions are supported by Fast Path but the others are not.

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 DB Monitor Report print program is an offline program that produces reports summarizing information collected by the DB Monitor program. 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

Example output of each of these reports are in the *IMS/ESA Utilities Reference: System*. Each field in the reports is explained, followed by a summary of how you can use the report. Many of these reports are also provided by the IMS Monitor, which is described in *IMS/ESA Administration Guide: System*. Where the same report is produced by both the DB and IMS Monitor, the description of the report in the *IMS/ESA Utilities Reference: System* is applicable for both.

Information on operating the DB Monitor is contained in *IMS/ESA Operations Guide*.

When the DB 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 DB Monitor in a single-thread test environment rather than multi-thread application environments.

This ensures that the data gathered by the DB Monitor can be related to a particular program.

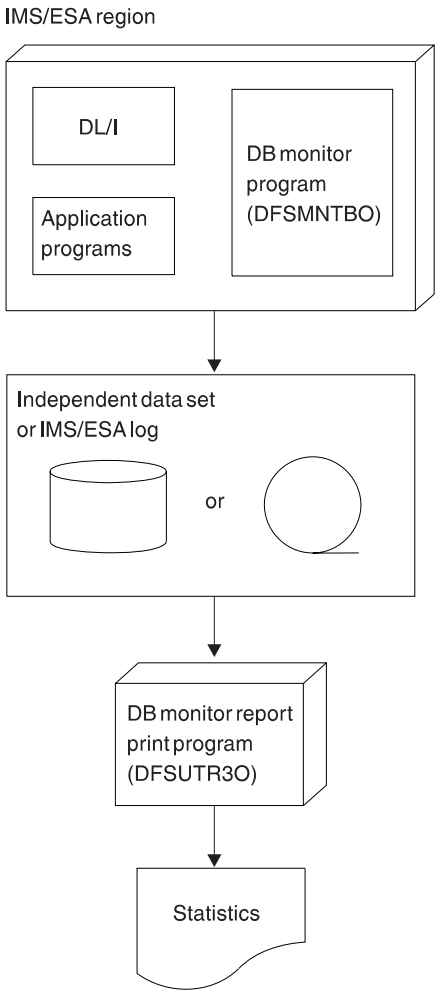


Figure 141. How the DB Monitor Works

Using Database Monitoring Aids

You can use IBM productivity aids to help monitor your database. Except for Batch Terminal Simulator (BTS), which cannot be used in the DBCTL environment, the productivity aids discussed in this section can be used in the batch, DBCTL, and DB/DC environments. For the Access Method Services `LISTCAT` command, explanations of the report fields are given. For other productivity aids, brief descriptions are given, and you are referred to sources of more detailed information on how to use them.

Access Method Services (LISTCAT Command)

For databases using VSAM, you can use the `LISTCAT` command to get VSAM cluster statistics from the VSAM catalog. This command produces an extensive report showing much information not pertinent to DL/I database monitoring. Therefore, unlike the information in this chapter for other monitoring aids, the following information includes interpretive details about some of the fields of the report. It describes the fields showing VSAM cluster attributes that affect DL/I database performance. Refer to Figure 142 on page 314 for a sample LISTCAT ALL report.

The first section of the LISTCAT ALL report is entitled CLUSTER. The first line of this section lists the cluster name, which is the DSNAME that will be used in a DD statement referencing the cluster. The components of the cluster can be defined or allowed to default to a VSAM internal name.

In general, it is preferable to define the name of the components with a name created by adding a suffix to the cluster name. Adding a suffix to the cluster name will result in the catalog record for it being maintained near the cluster record. By appending DATA or INDEX to the cluster name, it becomes more convenient for referencing in a `PRINT` command, and it is easier to identify in a LISTC VOLUME. These names are referenced in the ASSOCIATIONS section. The other fields of the cluster section do not have a direct influence on DL/I and are not discussed here.

The following information addresses the printouts concerned with the components of the cluster, first looking at the ESDS component of a HIDAM database. This is followed by the differences in the ESDS of an HDAM database. The ESDS for the HIDAM index and for secondary indexes is then discussed. The approach, in each case, is to describe those fields of each type that are different from those described for previous types.

HIDAM ESDS LISTCAT Report

Regarding the DATA section of the report for the ESDS (entry sequenced data set), the HISTORY or PROTECTION PSWD fields are not addressed. The ASSOCIATIONS field merely refers back to the cluster level.

Attributes Information in the LISTCAT Report

This section lists the attributes that have been either explicitly defined or have defaulted values. The first of these, KEYLEN and REP, have no meaning for a DL/I ESDS.

AVGLRECL and MAXLRECL are the defined average and maximum record lengths. For a DL/I database component, these are always identical, indicating fixed-length records. For an ESDS, this record length must be the CI length, less 4 bytes for the CIDF and less 3 more for the RDF.

The defined CISIZE is found at the end of this first line. As a general practice, it is advisable to have the DBD specifications for these values match the ones used in the definition.

```
CLUSTER ------------MJK.CLUSTER1
     HISTORY
          OWNER-IDENT-----OWNCLUST     CREATION----------76.320    RCVY-VOL----------333001    RCVY-CI--------X'00000E'
          RELEASE----------------2     EXPIRATION--------77.054    RCVY-DEVT----X'30502009'
     PROTECTION
          MASTERPW----------MASTCL     UPDATEPW----------UPDCL     CODE--------------CODECL
          CONTROLPW---------CNTLCL     READPW------------READCL    ATTEMPTS----------------3    USVR--------------(NULL)
          USAR--------------(NONE)
     ASSOCIATIONS
          DATA----MJK.CLUSTER1.DATA
          INDEX--MJK.CLUSTER1.INDEX
          AIX--MJK.ALT.INDEX1
   DATA---------MJK.CLUSTER1.DATA
     HISTORY
          OWNER-IDENT--------(NULL)    CREATION----------76.320    RCVY-VOL----------330001    RCVY-CI--------X'00000D'
          RELEASE----------------2     EXPIRATION--------00.000    RCVY-DEVT----X'30502009'
     PROTECTION------------(NULL)
     ASSOCIATIONS
          CLUSTER------MJK.CLUSTER1
     ATTRIBUTES
          KEYLEN-------------------5    AVGLRECL--------------80    BUFSPACE-----------25088    CISIZE-------------12288
          RKP----------------------5    MAXRECL--------------100    EXCPEXIT--------EXITCLUS    CI/CA------------------2
          SHROPTNS(1,3)     RECOVERY    SUBALLOC          NOERASE    INDEXED       NOWRITECHK    NOIMBED       NOREPLICAT
          UNORDERED          NOREUSE    NONSPANNED
     STATISTICS
          REC-TOTAL--------------20    SPLITS-CI--------------0    EXCPS-------------------4
          REC-DELETED------------0     SPLITS-CA--------------0    EXTENTS-----------------1
          REC-INSERTED-----------0     FREESPACE-%CI---------15    SYSTEM TIMESTAMP:
          REC-UPDATED------------0     FREESPACE-%CA---------20         X'89E5CE26C0623000'
          REC-RETRIEVED----------20    FREESPC-BYTES------12288
     ALLOCATION
          SPACE-TYPE----------TRACK    HI-ALLOC-RBA-------24576
          SPACE-PRI---------------2    HI-USED-RBA--------24576
          SPACE-SEC---------------2
     VOLUME
          VOLSER-------------333001    PHYREC--------------4096    HI-ALLOC-RBA-------24576    EXTENT-NUMBER----------1
          DEVTYPE-------X'30502009'    PHYRECS/TRK------------3    HI-USED-RBA--------24576    EXTENT-TYPE--------X'00'
          VOLFLAG-------------PRIME    TRACS/CA---------------2
          EXTENTS:
          LOW-CCH-------X"00040005'    LOW-RBA----------------0    TRACKS-----------------2
          HIGH-CCHH-----X'00040006'    HIGH-RBA-----------24575
   INDEX-------MJK.CLUSTER1.INDEX
     HISTORY
          OWNER-IDENT--------(NULL)    CREATION----------76.320    RCVY-VOL----------330001    RCVY-CI--------X'00000F'
          RELEASE----------------2     EXPIRATION--------00.000    RCVY-DEVT----X'30502009'
     PROTECTION------------(NULL)
     ASSOCIATIONS
          CLUSTER------MJK.CLUSTER1
     ATTRIBUTES
          KEYLEN-------------------5    AVGLRECL----------------0    BUFSPACE---------------0    CISIZE---------------512
          RKP----------------------5    MAXLRECL--------------505    EXCPEXIT--------EXITCLUS    CI/CA-----------------20
          SHROPTNS(1,3)     RECOVERY    SUBALLOC          NOERASE    MOWRITECHK       NOIMBED    NOREPLICAT     UNORDERED
          NOREUSE
     STATISTICS
          REC-TOTAL--------------1     SPLITS-CI--------------0    EXCPS-------------------4    INDEX:
          REC-DELETED------------0     SPLITS-CA--------------0    EXTENTS-----------------1    LEVELS-----------------1
          REC-INSERTED-----------0     FREESPACE-%CI----------0    SYSTEM-TIMESTAMP:            ENTRIES/SECT-----------1
          EC-UPDATED-------------0    FREESPACE-%CA----------0        X'89E5CE26C0623000'     SEQ-SET-RBA------------0
          REC-RETRIEVED----------0     FREESPC-BYTES--------9728                                HI-LEVEL-RBA-----------0
     ALLOCATION
          SPACE-TYPE----------TRACK
          SPACE-PRI---------------1    HI-ALLOC------------10240
          SPACE-SEC---------------1    HI-USED-RBA----------512
     VOLUME
          VOLSER-------------333001    PHYREC-SIZE----------512    HI-ALLOC-RBA-------10240    EXTENT-NUMBER----------1
          DEVTYPE-------X'30502009'    PHYRECS/TRK-----------20    HI-USED-RBA----------512    EXTENT-TYPE--------X'00'
          VOLFLAG-------------PRIME    TRACS/CA----------------1
          EXTENTS:
          LOW-CCHH------X'00040007'    LOW-RBA----------------0    TRACKS-----------------1
          HIGH-CCHH-----X'00040007'    HIGH-RBA-----------10239
```

Figure 142. An Example of LISTCAT ALL Output

The BUFSPACE field is not used because DL/I dynamically calls for buffer creation under shared resources and, based upon user control, input statements. The EXCPEXIT field is not currently used by DL/I.

CI/CA is of little importance with an ESDS except as an indicator of the frequency of DEB or for end-of-extent checking done while sequentially processing a VSAM data set.

Binary decision attributes are listed on the following lines. SHROPTNS are not used by DL/I and the only concern should be that they do not degrade performance. Option 4 is not allowed with shared resources and is therefore not valid for a DL/I database.

The SPEED option is usually the one to use instead of RECOVERY, because DL/I does not provide any facility for restarting a load operation.

UNIQUE or SUBALLOC specifies whether this component will stand alone or share VSAM space with others. The UNIQUE value results in the component having an entry in the VTOC, and it can increase the system overhead to open and close the database. It also results in a full cylinder CA being assigned, which can be inappropriate. SUBALLOC can also often ensure that secondary allocations are nearby in cases where VSAM space is small relative to volume space.

ERASE/NOERASE is an installation security decision. NONINDEXED instead of INDEXED says this cluster is an ESDS. WRITECHK/NOWRITECHK is an integrity decision, but WRITECHK results in one additional rotation for each write.

The IMBED and REPLICATE options have no meaning with NONINDEXED. The remaining attributes of NOREUSE and NONSPANNED must be specified for DL/I.

Statistics Information in the LISTCAT Report

This section of the report is potentially the most valuable in determining performance values. The numbers must be used with care, since they are much less complete than they imply, especially in the case of DL/I activity.

The first column under statistics lists record activity. Remember these are VSAM records, not DL/I records. The REC-TOTAL value is the number of VSAM records in the component which, in the case of a DL/I ESDS, always equals the number of CIs used in the database. For a HIDAM database, this provides the amount of storage required to hold the data.

An increase in this number over a period of time indicates additions to overflow in terms of VSAM records. This information can be used as an indication of the need to reorganize and as an indication of the effectiveness of free space.

The REC-DELETED statistics are not updated because records cannot be deleted from an ESDS. DL/I deletes are not VSAM deletes in a HIDAM ESDS.

The REC-INSERTED value is updated only when a VSAM record is inserted in a data set and does not include additions to the end. In an ESDS, additions are allowed only at the end, therefore, this field is never incremented.

The REC-UPDATED value does not reflect DL/I updating activity, because DL/I uses the shared resources macros rather than the usual VSAM macros.

REC-RETRIEVED is the number of times DL/I requested VSAM to locate a record. This number has little to do with DL/I retrievals, but it can indicate insert positioning, reacquiring a record during complex maintenance, or several other circumstances. It should not be confused with physical reads or DL/I calls. In general, it has little relevance for DL/I.

The second column of statistics is not used for ESDS except for the FREESPC-BYTES value at the bottom of the column, which is an indication of the space remaining in the allocation. It is the space between the end of the last VSAM record and the end of the current allocation. It should not be confused with DL/I free space, which is unknown to VSAM.

The third column of statistics starts off with the number of EXCPs issued against this component. This number reflects the number of actual I/Os issued for the component. This number can also be very helpful in determining adequacy of buffers and causes of performance problems.

The number of EXTENTS provides information about the degree of separation of data in the component and the amount of insert activity. More than one extent per volume means some performance impact can exist, caused by longer than necessary seeks. An increasing number of extents indicates the growth of the database and warns of the need for reorganization.

Remember all counts are maintained from the creation of the cluster (through DL/I load or reorganization, or by the Access Method Services REPRO or EXPORT/IMPORT function) and that to determine the effect of any one run against the database requires a before and after picture of the catalog statistics.

Allocation Parameters in the LISTCAT Report

This section describes the allocation parameters specified in the definition of the component and the current state of the space.

The first column shows the original allocation parameters of SPACE-PRI (primary allocation) and SPACE-SEC (secondary allocation). The second column shows the current state of the allocation in terms of the highest byte allocated and the highest byte used. Again, these are in terms of VSAM, not DL/I. For an ESDS, the HI-USED-RBA is the furthest point forward in the data set used by VSAM.

Volume Information in the LISTCAT Report

This section describes the physical information about the component as it resides on each volume. The first column contains volume device type information and is of little use in DL/I monitoring.

The second column displays the track information. The PHYREC-SIZE is the physical block size that underlies the CI size. Generally, the larger this value, the better track utilization is for the component. VSAM assigns the largest allowable value for the device that is an even smaller multiple of the CI size. A small value in this field probably means poor disk utilization. For instance, on a 3330-1, a 512-byte block stores 10240 bytes for each track, and a 2048-byte or 4096-byte block uses 12288 bytes for each track. The second line of the column shows the number of these blocks that fit on a track and the third line (TRACKS/CA) shows the number of tracks in a CA. Like the CI/CA field, this has little use in an ESDS.

The third column shows the allocation on this volume in terms of bytes and how much of it has been used for VSAM records.

The last column shows the number of extents on this volume. Usually, for an ESDS, this number should be 1 to ensure that seeks are minimum. This number can be used as an indication of the need for a reorganization as secondary extents are allocated. The EXTENT-TYPE field indicates whether this extent is data or index.

Extents Information in the LISTCAT Report

This subsection describes each of the extents on the volume for this component.

The first column of each of these subsections provides the physical cylinder and head of the allocation. With multiple extent components, it is possible to use this to determine how badly multiple allocations have impacted seek time.

The second column shows the VSAM relative bytes assigned to this extent. The information in this column is of little use for DL/I except for extreme debugging.

The third column, TRACKS, provides a convenient translation of cylinder and head information into tracks.

HDAM ESDS LISTCAT Report

Statistics Information in the LISTCAT Report

The REC_TOTAL figure for HDAM is the number of CIs used (as in HIDAM), but the interpretation is different. The number in HDAM is the sum of CIs in the root addressable area, plus the overflow CIs, plus one for record zero. Changes in this number will therefore reflect additions of data that could not be put into the root addressable area. At load time, the figure reflects data excluded from the root addressable area by the BYTES or the SCAN parameters of the DBD (or by a packing density greater than 100%).

Other fields in the statistics, allocation, and volume areas record the same type of information as for HIDAM ESDS data sets.

All Other Information in the LISTCAT Report

Except for the STATISTICS section, the interpretation of the LISTCAT output for an HDAM ESDS is the same as for a HIDAM ESDS.

HISAM or Index KSDS LISTCAT Report

Information recorded for a KSDS has a much different significance than for the HIDAM and HDAM ESDSs. In DL/I, a KSDS is used in three cases: the HISAM database, the primary index, and the secondary index. For primary or unique secondary index databases, the statistics listed in the data portion of the LISTCAT are accurate. For non-unique secondary index databases, however, use LISTCAT for the VSAM ESDS to get *all* the statistics for the database.

Data Portion of Cluster Information in the LISTCAT Report

For HISAM databases, the data portion of the cluster is the primary data portion of the database. In the case of the primary and secondary indexes, the data portion of the cluster is the index to the data of the database. This distinction must be kept in mind.

Attributes Information in the LISTCAT Report

The attributes fields are important in the KSDS. The KEYLEN attribute is the length of the root key for both the HISAM database and the primary index. In the case of the secondary index, it is the sum of the lengths of the SRCH and SUBSEQ fields (and CONST field, if used) specified in the XDFLD statement. It is the length of the key field for this database or index as VSAM will see it.

RKP is the relative position of this key within the VSAM record, not within the DL/I segment. Thus, for an index, the relative key position is usually 6, allowing for the segment code, delete byte, and target pointer (if PTR=SYMB has been specified, the RKP will be 2). For the HIDAM database, the RKP will be 6, plus the offset into the root of the key field. An additional prefix can exist if the root is a logical parent.

AVGLRECL and MAXLRECL are always equal, because DL/I uses fixed-length VSAM logical records. Unlike the ESDS, in the KSDS blocking usually exists. Ideally, the logical record length is a submultiple of the CI size, minus 10 for VSAM control information.

CISIZE is what you specify in the defines.

REPLICATE and IMBED options are reflected as specified.

Index Portion of Cluster Information in the LISTCAT Report

The catalog for the index portion of the KSDS contains information that should be checked. Particularly, fields exist that should be checked to ensure that the data set has the characteristics defined for it. Some performance characteristics can also be implied from some of the values.

Attributes Information in the LISTCAT Report

The KEYLEN and REP fields are the same as for the data portion. The AVGLRECL and MAXLRECL fields, however, do not relate to those of the data portion and, henceforth, are not important for database monitoring.

CISIZE, however, is important. Because allocation to a VSAM subpool is done based on CI size, the CISIZE value determines the contention with other data sets for pool space. The value can be different from the value specified. If the size specified is smaller than what Access Method Services considers adequate, the size is increased with no message to the user. Failure to check this value after allocation can result in contention between data and indexes when you had planned to isolate the two. The problem occurs when the index and data have been assigned small CI sizes and the CA is large. To get a small index CI, the data CI must be enlarged, the CA reduced, or both.

The attribute NOIMBED is always present in the index list. It is irrelevant in the index and its specification is reflected in the data portion. REPLICAT is present if specified.

Statistics Information in the LISTCAT Report

Statistics reflects activity at the index level only. EXCPs provide an indication of whether the index that was anticipated to be resident is resident. LEVELS provides an indication of buffering requirements in a storage constrained system. The pool should have at least enough buffers to hold one CI at each level.

Volume Information in the LISTCAT Report

By examining the EXTENT information, it is possible to determine the distance that the index set has been located from the data. VSAM makes no attempt to locate index and data adjacent to each other. Where the two coexist on the same pack, it is usually desirable to have them adjacent or have the index centered in the data. Centering the index in the data or getting the two adjacent to each other can only be done in VSAM by manipulating the available space at allocation time to force the space to occur where it is desired.

If IMBED has been specified, an extent for each CA exists in the data set in the index.

For information about how to use the `LISTCAT` command, see *MVS/DFP Access Method Services for VSAM Catalog* manual.

IEHLIST Utility (LISTVTOC Command)

For HDAM databases using OSAM, one way to assess the need for reorganization is to monitor the DASD volume table of contents (VTOC). The `LISTVTOC` command of the IEHLIST utility prints a report showing the VTOC of the DASD (direct access storage device) volume on which the OSAM database resides. For this purpose, the NO EXT field in the report helps you monitor the increase in the extent number.

For information about this MVS utility, see *IMS/ESA Utilities Reference: Transaction Manager*.

HD Reorganization Unload Utility

This utility unloads a specific database into a QSAM data set. You can also use it to gather statistics that will enable you to determine whether or not to reorganize.

For information about running this utility and to see a sample of the report, see *IMS/ESA Utilities Reference: System*.

HISAM Reorganization Unload Utility

This utility is designed primarily to unload a HISAM database and to create a reorganized output usable as input to either the Database Recovery utility or the HISAM Reload Resolution utility. In addition, you can use the printed Data Set Group Statistics and Segment Level Statistics to monitor the need for reorganization of the database.

For information about how to run this utility and to see a sample of the report, see *IMS/ESA Utilities Reference: System*.

DL/I Test Program

This program (DFSDDLT0) is designed primarily for testing DL/I call sequences. As a test program, it provides reports showing call execution statistics. To use it as a monitoring aid, you can provide control statements to specify a database, establish print options, get a log data set, and cause the programs to compare the actual results with anticipated results.

You can use the timing function, which you can turn on and off, to get performance information when various call sequences are run against your database. A DFSDDLT0 report shows the elapsed time taken to perform a DL/I task and the time-of-day of the completion of each DL/I call.

You can find more information about the DL/I Test Program (DFSDDLT0) in *IMS/ESA Application Programming: Database Manager* .

The DL/I Test Program cannot be used by CICS for online or shared batch applications but can be used for stand-alone batch programs. If used for stand-alone batch programs, it can be useful to interpret the database performance since it might be implemented for online or shared database programs.

Database Surveyor Utility

You can use the output of this utility to determine the need to partially reorganize your database. You can specify the database to be surveyed and the range of keys or blocks to be analyzed. DB Surveyor produces statistics about the blocks and records accessed for each partition of the specified range.

Using Database Monitoring Aids

To see sample reports and get information about how to run this utility (DFSPRSUR), see *IMS/ESA Utilities Reference: Database Manager*.

Fast Path Log Analysis Utility

The Fast Path Log Analysis utility (DBFULTA0) prepares statistical reports for Fast Path based on data recorded on the IMS system log. This is an offline utility 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.

For more information on CCTL transactions, refer to *IMS/ESA Customization Guide*. For more information of the Fast Path Log Analysis utility, refer to *IMS/ESA Utilities Reference: System*.

IMS System Utilities/Database Tools

The IMS System Utilities/Database Tools (DBT) can help you:

- Detect and repair broken databases
- Analyze detailed statistics about databases
- Model potential DBD changes without creating new databases
- Restructure databases
- Decrease reorganization time for some HDAM databases
- Reduce processor time and elapsed time required for sequential retrieval from HDAM, HIDAM, and HISAM databases
- Perform high-speed database unloading
- Tune buffer handlers
- Produce pictorial layouts of physical and logical databases
- Produce detailed reports about DBDs and PSBs
- Verify contents in a VSAM data set against data that you supply
- Replace contents in a VSAM data set with data you supply
- Detect and report broken DEDB areas
- Analyze detailed statistics about DEDB areas
- Model potential DEDB changes without creating new databases
- Perform high-speed unloading and reloading of DEDB areas

You can use four of the utilities to help monitor your database.

HD Pointer Checker Utility

In addition to validating pointers, the pointer checker programs produce detailed statistics about the segments and pointers in the database. This utility analyzes disk space used and gives you reports showing free space and pointer statistics. Also, output from this utility is used by the HD tuning aid utility to enable you to determine the effects of changing parameters, randomizing routines, or both.

HD Tuning Aid

This utility produces a map of how the HD data is actually stored throughout the database. You can use this information and the segment and pointer statistics from the pointer checker to optimize distribution of the data.

HDAM Physical Sequence Sort/Reload Utility

This utility sorts the sequential file that contains the unloaded database during the reorganization of an HDAM database. This sorted file is used to make the reload step in the reorganization process run faster. This utility can also print reports showing statistics about database record sizes and HD randomization.

DEDB Pointer Checker

This program detects DEDB database errors and produces reports that pinpoint the errors and locations within the database. It also creates numerous reports that aid in the tuning of databases.

You can find information about DBT (Program Number 5685-093) in *IMS System Utilities/Database Tools (DBT) General Information Manual*.

Batch Terminal Simulator

The primary purpose of the batch terminal simulator (BTS) program is to enable testing of IMS DB Batch, TM Batch, and online application programs in an IMS batch environment without use of teleprocessing hardware.

You can use BTS to monitor your database, because it tracks the interaction between the database and the application program. BTS has access to every IMS call made by the application program; therefore, it provides you detailed information about the PCBs, SSAs, SPAs, and key feedback areas.

For details about BTS (Program Number 5668-948), see *IMS/VS Batch Terminal Simulator Program Reference and Operations Manual*.

This program does not support CICS.

IMS Monitor Summary and System Analysis Program II

The IMS Monitor Summary and System Analysis Program II (IMSASAP II), a field developed program, is a performance analysis and tuning aid for IMS database and data communication systems.

IMSASAP II executes under the system for Generalized Performance Analysis Reporting (GPAR). IMSASAP II processes IMS DB and IMS monitor data to provide summary, system analysis, and program analysis level reports that assist in the analysis of an IMS system environment.

With IMSASAP II, you can select reports and reporting options to satisfy your requirements for database analysis, without running the DB Monitor Report print program (DFSUTR30).

IMSASAP II (Program Number 5798-CHJ) requires GPAR (Program Number 5798-CPR). For a complete description of IMSASAP II, see *IMSASAP II Program Description/Operations Manual*.

The DL/I System Service STAT Call

This DL/I system service call retrieves statistical information about the database buffer pools. Using this call, you can get the full OSAM buffer pool information, the full VSAM buffer subpool information, or a summary of either.

For more information about the STAT call, see *IMS/ESA Application Programming: Database Manager*.

Chapter 14. Tuning Your Database

About This Chapter

Tune your database either to improve performance or to better utilize database space. This chapter introduces the reorganization utilities, which you can use tune your database. The chapter also describes the various types of tuning changes you can make with the reorganization utilities, as well as, when and how to make them.

This chapter examines the following aspects of database tuning:

- Reorganizing the database
- Changing DL/I access methods
- Changing the hierarchic structure
- Changing direct-access storage devices
- Tuning OSAM Sequential Buffering
- Adjusting HDAM options
- Adjusting buffers
- Adjusting VSAM options
- Adjusting OSAM options
- Changing the amount of space allocated
- Changing operating system access methods
- Changing the number of data set groups

Note the following information: First, 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 chapter 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 flowchart on the last page of this chapter. When used in conjunction with the individual procedures in this chapter, the flowchart guides you in making some types of multiple changes to the database.

Second, 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.

If you are using data sharing, additional information about tuning is in *IMS/ESA Administration Guide: System* .

Reorganizing the Database

Reorganizing a database is changing either its *storage* or *structure*.

Change storage in a database when use of storage space becomes inefficient. Storage becomes inefficient as a database grows and parts of a database record get widely scattered or available space for adding new records is used up. This section emphasizes reorganizing storage in a database.

You need to change the structure of a database when changing user requirements necessitate changes in the database design. You also need to change a database structure when you need to use new or different options or when you have found a more efficient way to structure the database. Structural changes to a database can often be made using the reorganization utilities in this chapter. See “Chapter 15. Modifying Your Database” on page 365, for information on the structural changes.

When Should You Reorganize?

You should reorganize your database when performance is becoming unacceptable. 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.

The various aids that exist for monitoring a database help you determine when it is time to reorganize your database. These aids are discussed in “Chapter 13. Monitoring Your Database” on page 309.

Steps in Reorganizing

Three basic steps are required in reorganizing a database (when you are not making structural changes to the database):

1. Unloading the existing database.
2. Deleting 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. Reloading the database.

Protecting Your Database

When you reorganize your database, 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 make a copy of your database as soon as it is reloaded and before any application programs are run against it. You need a backup copy in case of system failure. You can copy your database using the Database Image Copy utility or the Database Image Copy 2 utility, which are described in detail in *IMS/ESA Utilities Reference: Database Manager*.

Using the Reorganization Utilities

Utilities are supplied with the system to help you reorganize your database. These utilities are documented in *IMS/ESA Utilities Reference: Database Manager*. This section is designed to introduce you to the utilities, tell you what they do, and how they work together.

You should know the following information about the utilities:

- 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. Information about when these utilities can be used to make structural changes to a database is contained in “Chapter 15. Modifying Your Database” on page 365.
- 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 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

Partial 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

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.

Reorganization Without UCF

When you do not use UCF, reorganization of the 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:

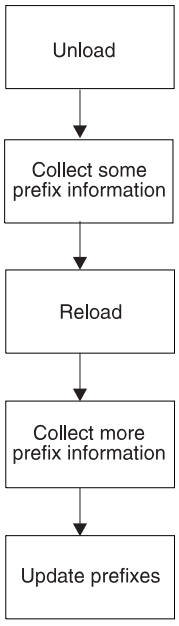
- For HISAM databases, the HISAM Reorganization Unload utility and the HISAM Reorganization Reload utility
- For HIDAM index databases (if reorganized separately from the HIDAM database), the HISAM Reorganization Unload utility and the HISAM Reorganization Reload utility
- For SHISAM, HDAM, and HIDAM 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 following discussion of the utilities, 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 occur looks similar to this:



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.

Figure 143 shows you the sequence in which utilities are executed if logical relationships or secondary indexes exist.

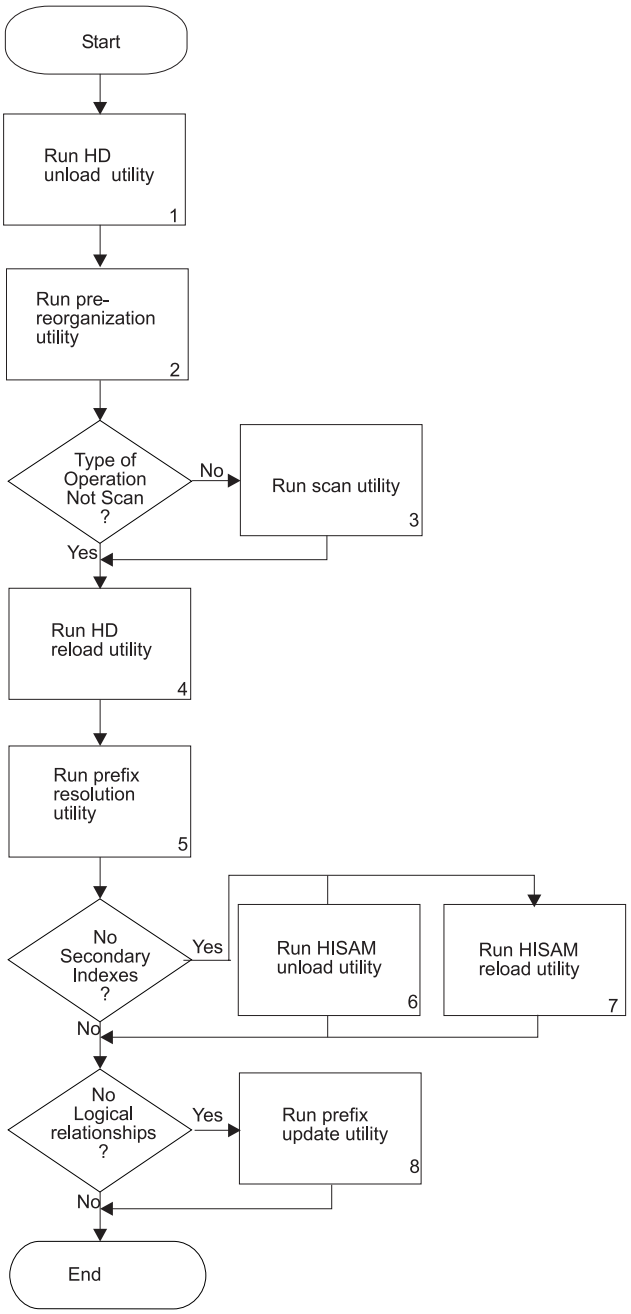


Figure 143. Steps in Reorganizing When Logical Relationships or Secondary Indexes Exist

HISAM Reorganization Unload Utility (DFSURUL0)

Figure 144 shows the input to and output from the HISAM Reorganization Unload utility.

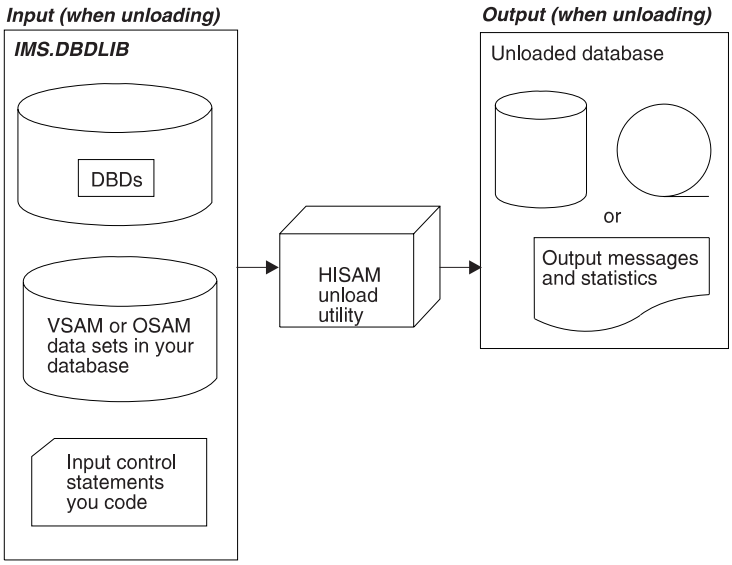


Figure 144. HISAM Reorganization Unload Utility (DFSURUL0)

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.

HISAM Reorganization Reload Utility (DFSURRL0)

Figure 145 shows the input to and output from the HISAM Reorganization Reload utility.

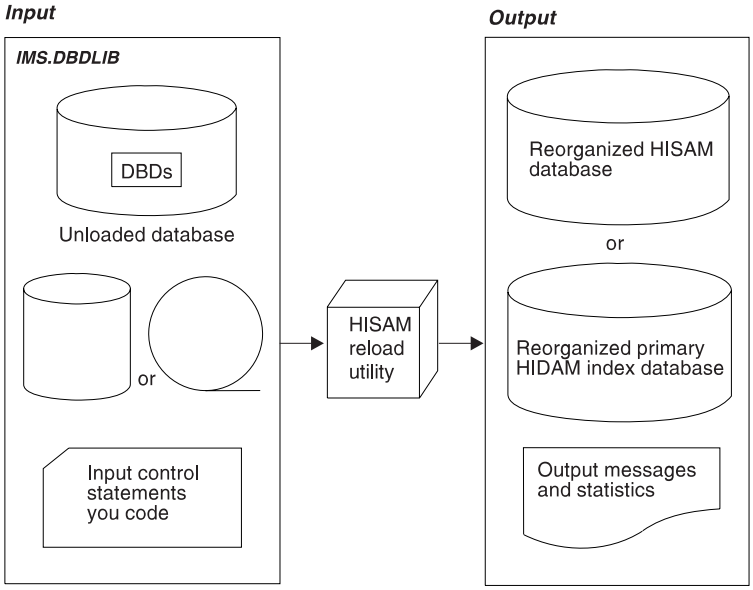


Figure 145. HISAM Reorganization Reload Utility (DFSURRL0)

You use the HISAM reload utility to reload a HISAM database. (SHISAM databases are reloaded using the HD Reorganization Reload utility.) You also use the HISAM reload utility to reload the primary index of a HIDAM database. 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.

HD Reorganization Unload Utility (DFSURGU0)

Figure 146 shows the input to and output from the HD Reorganization Unload utility.

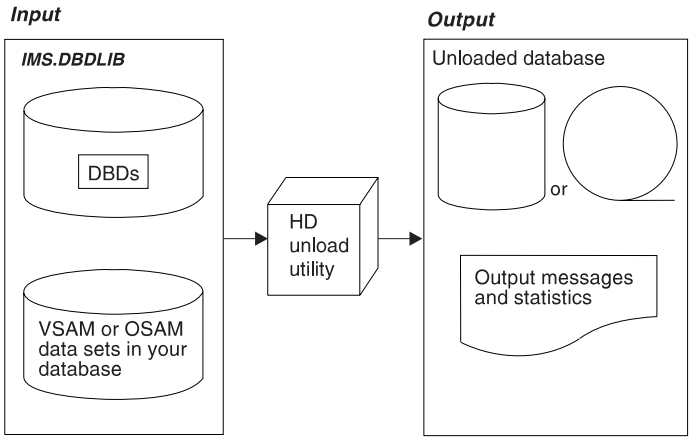


Figure 146. HD Reorganization Unload Utility (DFSURGU0)

You use the HD Unload utility to unload:

- 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.

HD Reorganization Reload Utility (DFSURGL0)

Figure 147 shows the input to and output from the HD Reorganization Reload utility.

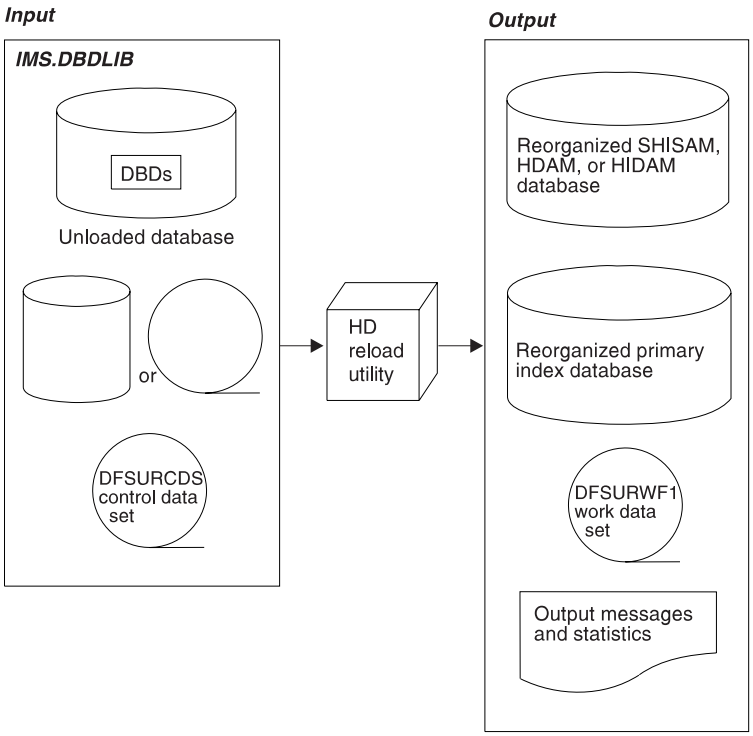


Figure 147. HD Reorganization Reload Utility (DFSURGL0)

You use the HD Reload utility to reload:

- HDAM, HIDAM, 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

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 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 Reload utility produces as output the DFSURWF1 work data set. DFSURCDS identifies the information that will be collected on DFSURWF1.

The DFSURWF1 work data set will become input to the Database Prefix Resolution utility. Note in Figure 147 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.

Database Prereorganization Utility (DFSURPR0)

Figure 148 shows the input to and output from the Database Prereorganization utility.

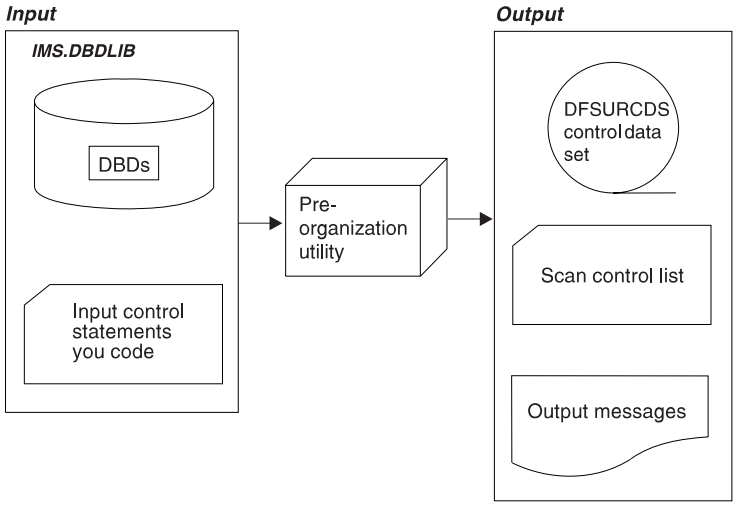


Figure 148. Database Prereorganization Utility (DFSURPR0)

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 Database Prereorganization 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 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)

Figure 149 shows the input to and output from the Database Scan utility.

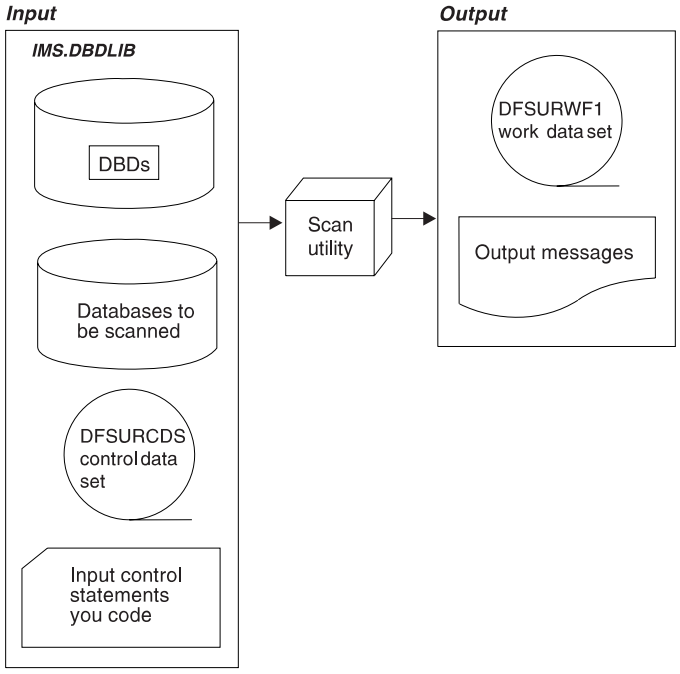


Figure 149. Database Scan Utility (DFSURGS0)

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. 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)

Figure 150 shows the input to and output from the Database Prefix Resolution utility.

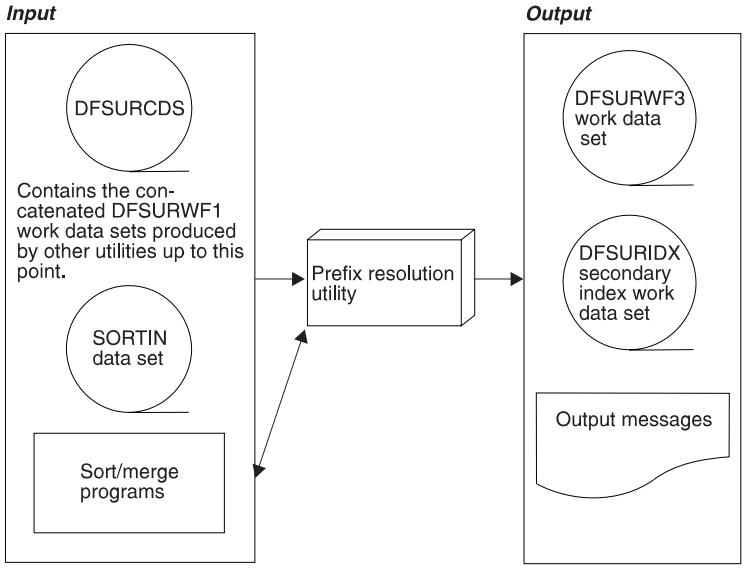


Figure 150. Database Prefix Resolution Utility (DFSURG10)

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 various work data sets that could be input to this 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 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 MVS 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)

Figure 151 shows the input to and output from the Database Prefix Update utility.

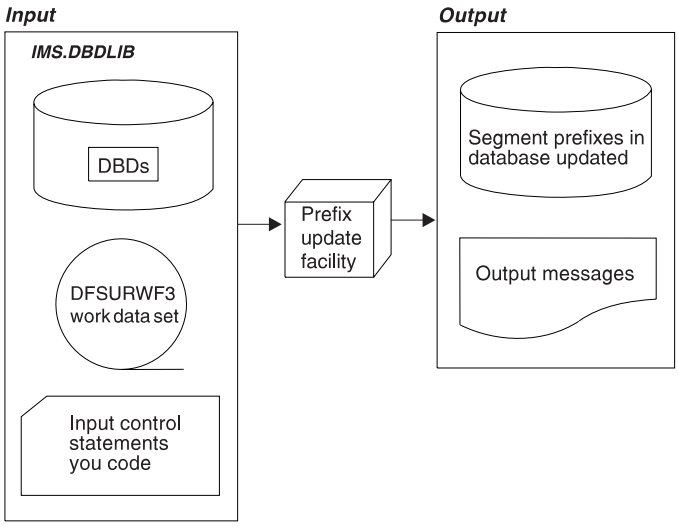


Figure 151. Database Prefix Update Utility (DFSURGP0)

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 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 using the HISAM unload and reload utilities to unload and reload a database, you can also use them 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.

Figure 152 on page 336 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 153 on page 337 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.

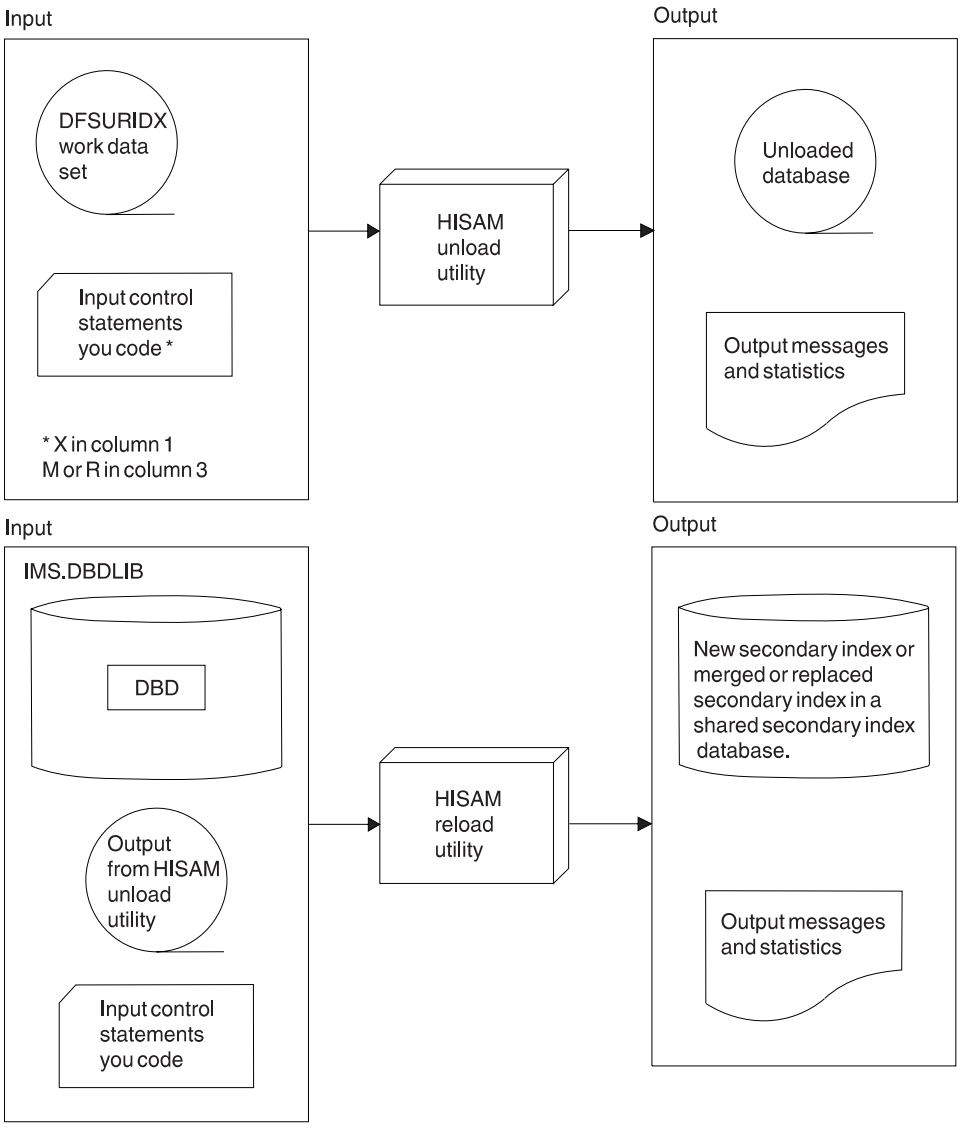


Figure 152. HISAM Reorganization Unload and Reload Utilities Used for Create, Merge, or Replace Secondary Indexing Operations

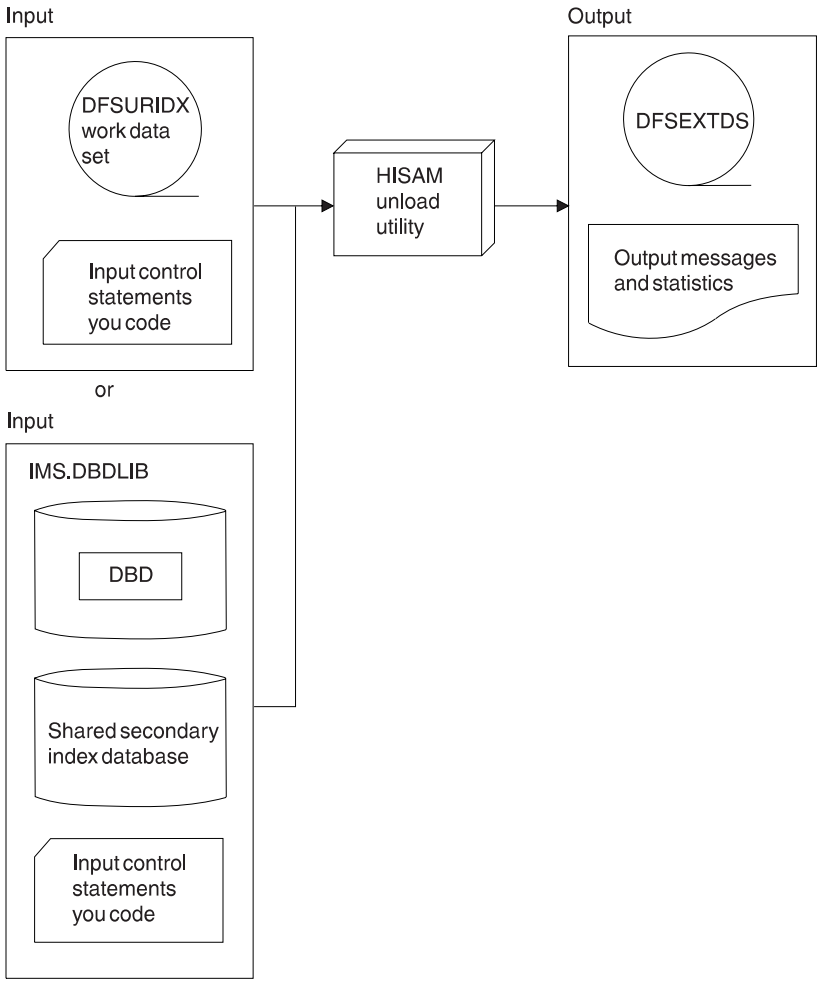


Figure 153. HISAM Reorganization Unload and Reload Utilities Used for Extract Secondary Indexing Operations

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)

Figure 154 on page 338 shows the input to and output from the Database Surveyor utility.

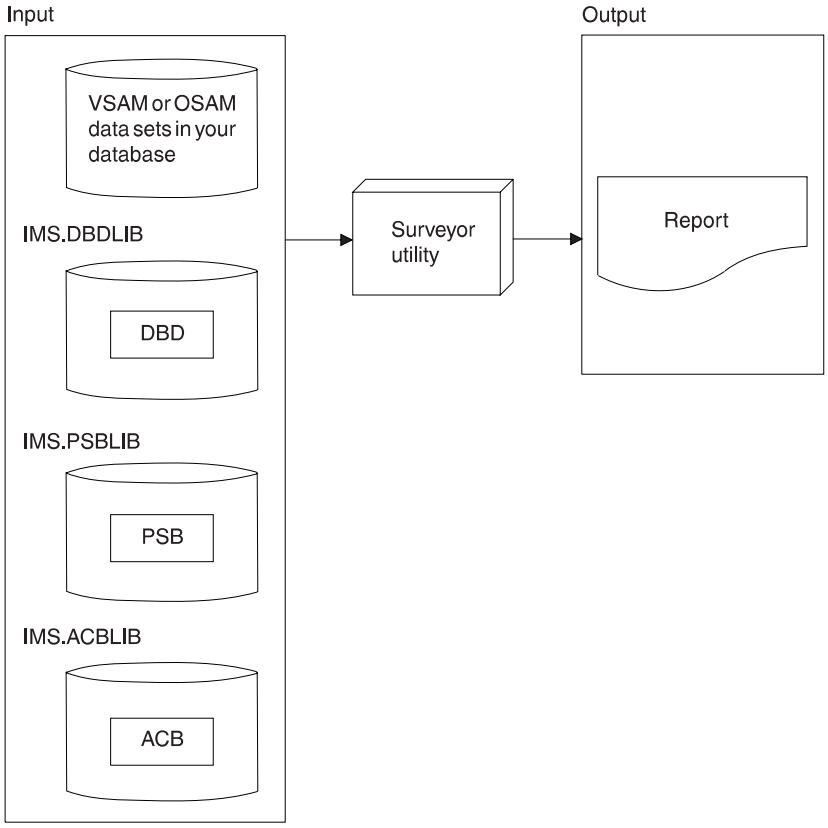


Figure 154. Database Surveyor Utility (DFSPRSUR)

You would need to use the Surveyor utility to scan all or part of an HDAM or HIDAM database to determine whether a reorganization is needed. 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.

Partial Database Reorganization Utility (DFSPRCT1)

Figure 155 on page 339 shows the input to and output from the Partial Database Reorganization utility.

You would 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 (described in the preceding section). 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.

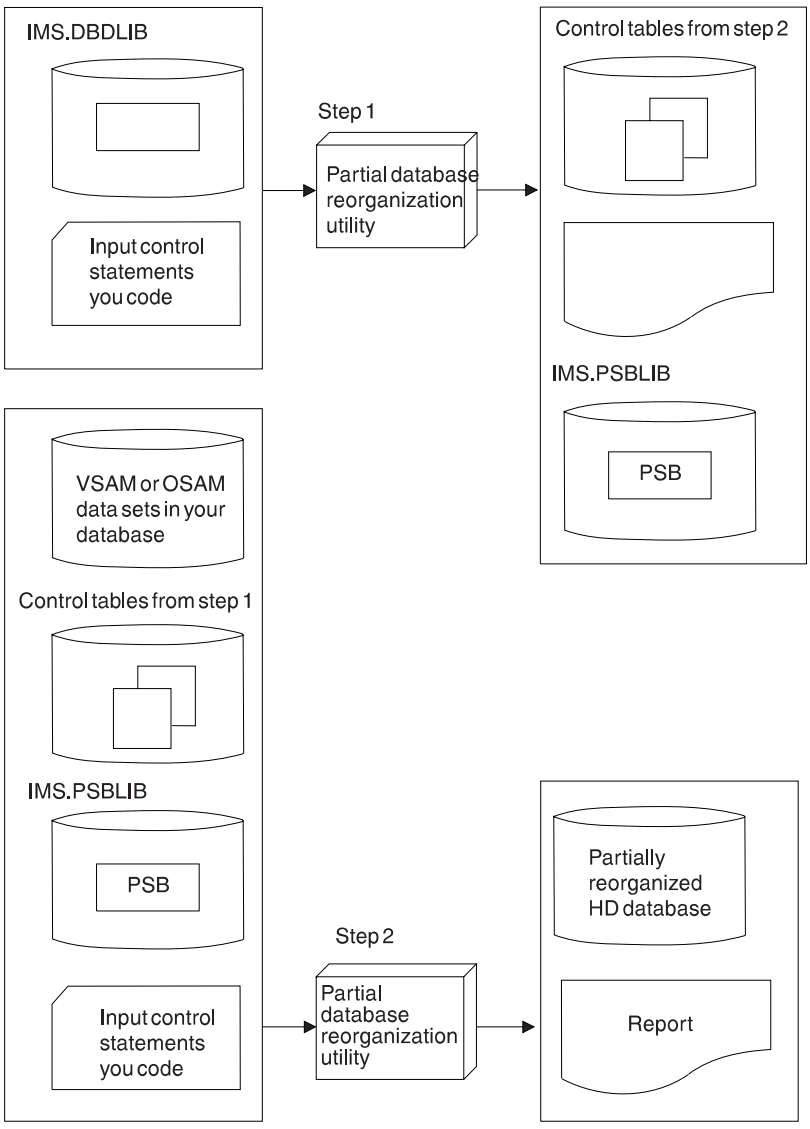


Figure 155. Partial Database Reorganization Utility (DFSPRCT1)

Procedure for Reorganizing a HISAM Database (No Logical Relationships or Secondary Indexes)

To reorganize a HISAM database when it does not use logical relationships or secondary indexes:

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 index database, using the HISAM Reorganization Reload utility. Make an image copy of your database once it is reloaded.

Procedure for Reorganizing an HD (HIDAM or HDAM) Database (No Logical Relationships or Secondary Indexes)

To reorganize an HD database when it does not use logical relationships or secondary indexes:

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 index database, using the HD Reorganization Reload utility. Make an image copy of your database once it is reloaded.

Procedure for Reorganizing a Primary or Secondary Index

HIDAM has a primary index. HISAM, HDAM, and HIDAM have separate secondary index databases when secondary indexing is being used. Both index types are reorganized in the same way:

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.

Procedure for Reorganizing a HISAM or HD Database (with Logical Relationships or Secondary Indexes)

Figure 143 on page 328 shows you the steps you must perform to reorganize a HISAM or HD database that uses logical relationships or secondary indexes:

1. Run the HD Unload utility to unload the database.
2. Run the Prereorganization utility to get information needed later to resolve secondary index or logical relationships.
3. If databases *not* being reorganized contain segments involved in a logical relationship with the database being reorganized, run the scan utility. This utility is run against the database that is *not* being reorganized. The scan utility gathers information needed later to resolve logical relationships.
4. Any time you unload a data set, you should delete and redefine space before reloading.
5. Run the HD Reload utility to reload the database.
6. Run the Prefix Resolution utility to accumulate and sort the prefix information collected in steps 3 and 5.
7. If the reloaded database uses secondary indexes, run the HISAM unload utility to format the secondary index information produced by the Prefix Resolution utility.
8. If Step 7 was run, then run the HISAM reload utility against the reloaded database to do one of the following:
   - Create the secondary index for the database being reorganized

- Merge the secondary index into the shared index database if a shared index database is being used

9. If the reloaded database uses logical relationships, run the Prefix Update utility against the reloaded and logically related databases. It updates the prefixes of all segments involved in logical relationships.
10. Remember to make an image copy of your database.

Changing DL/I Access Methods

When you originally chose a DL/I access method (or type of database), you chose it based on such things as:

- The type of processing you needed to do (sequential, direct, or both)
- The volatility of your data

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. “Chapter 4. Designing a Fast Path Database” on page 33 describes which type of DL/I access method to choose given your application’s characteristics. Assuming that you have decided to change access methods, this section 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 earlier in this chapter 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 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).

Procedure for Changing from HISAM to HIDAM

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. See “Chapter 6. Database Design Considerations for Full Function” on page 165, “Specifying Free Space (HDAM and HIDAM Only),” for a description of free space and how it is specified.
- 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. See “Chapter 4. Designing a Fast Path Database” on page 33, “Types of Pointers You Can Specify,” for a description of types of pointers and how to specify them.

- 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. See “Chapter 6. Database Design Considerations for Full Function” on page 165, “Choosing a Logical Record Length for HD Databases,” for a discussion of what things to consider in choosing a logical record length and how logical record lengths are specified.
- 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. See “Chapter 6. Database Design Considerations for Full Function” on page 165, “Determining the Size of CIs and Blocks,” for a discussion of what things to consider in choosing a CI or block size and how CI and block size are specified.
- 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. See “Chapter 6. Database Design Considerations for Full Function” on page 165, for a discussion of what to consider in choosing buffer number and size and how they are specified.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate database size.

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 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. Rebuild the ACB if you are pre-building ACBs, rather than built them dynamically.
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 the database using the new DBD and the HD 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 immediately before and after reloading your database. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing from HISAM to HDAM

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. See “Chapter 4. Designing a Fast Path Database” on page 33, “Types of Pointers You Can Specify,” for a description of types of pointers and how to specify them.

- 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. See "Chapter 4. Designing a Fast Path Database" on page 33 under "Determining Which Randomizing Module You Will Use (HDAM Only)," for information on choosing a randomizing module and how use of one is specified.
- 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.)

  See "Chapter 4. Designing a Fast Path Database" on page 33 under "Choosing HDAM Options" for information on choosing the options and how they are specified.
- 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). See "Chapter 6. Database Design Considerations for Full Function" on page 165, "Choosing a Logical Record Length for HD Databases," for a discussion of what things to consider in choosing a logical record length and what logical record lengths are allowed.
- 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. See "Chapter 4. Designing a Fast Path Database" on page 33 under "Determining the Size of CIs and Blocks" for a discussion of what things to consider in choosing a CI or block size and how CI and block size are specified.
- 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. See "Chapter 6. Database Design Considerations for Full Function" on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space. See "Chapter 12. Loading Your Database" on page 285 under "Estimating the Minimum Size of the Database" for a description of how to calculate 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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
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 the database using the new DBD and the HD 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 will need to run additional utilities before reloading your database. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing from HIDAM to HISAM

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. See “Chapter 6. Database Design Considerations for Full Function” on page 165, “Choosing a Logical Record Length for HD Databases,” for a discussion of what things to consider in choosing a logical record length and what logical record lengths are allowed.
- 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. See “Chapter 4. Designing a Fast Path Database” on page 33 under “Determining the Size of CIs and Blocks” for a discussion of what things to consider in choosing a CI or block size and how CI and block size are specified.
- 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. See “Chapter 6. Database Design Considerations for Full Function” on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.
- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate database size.

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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
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 the database using the new DBD and the HD 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, run additional utilities right before and after reloading your database. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing from HIDAM to HDAM

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. See “Chapter 4. Designing a Fast Path Database” on page 33 under “Types of Pointers You Can Specify” for a description of types of pointers, their uses, and how to specify them.
- 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. See “Chapter 4. Designing a Fast Path Database” on page 33 under “Determining Which Randomizing Module You Will Use (HDAM Only)” for information on choosing a randomizing module and how use of one is specified.
- 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.

    See “Chapter 6. Database Design Considerations for Full Function” on page 165 under “Choosing HDAM Options” for information on choosing the options and how they are specified.
- 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. See “Chapter 6. Database Design Considerations for

Full Function" on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.

- Recalculate database space. You need to do this because the changes you are making will result in different requirements for database space. See "Chapter 12. Loading Your Database" on page 285 under "Estimating the Minimum Size of the Database" for a description of how to calculate database size.

Once 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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
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 the database using the new DBD and the HD 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. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing from HDAM to HISAM

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. See "Chapter 6. Database Design Considerations for Full Function" on page 165 under "Choosing a Logical Record Length for HD Databases" for a discussion of what things to consider in choosing a logical record length and what logical record lengths are allowed.
- 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. See "Chapter 6. Database Design Considerations for Full Function" on page 165 under "Determining the Size of CIs and Blocks" for a discussion of what things to consider in choosing a CI or block size and how CI and block size are specified.
- 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. See "Chapter 6. Database Design Considerations for Full Function" on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.

- Recalculate database space. You need to recalculate database space because the changes you are making will result in different requirements for database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate database size.

Once 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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
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 the database using the new DBD and:
   - Your load program, or
   - The HD 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 will need to run additional utilities right before and after reloading your database. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing from HDAM to HIDAM

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. See “Chapter 6. Database Design Considerations for Full Function” on page 165 under “Specifying Free Space (HDAM and HIDAM Only)” for a description of free space and how it is specified.

- 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. See “Chapter 4. Designing a Fast Path Database” on page 33 under “Types of Pointers You Can Specify” for a description of types of pointers, their uses, and how to specify them.
- 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. See “Chapter 6. Database Design Considerations for Full Function” on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.
- Recalculate database space. You need to recalculate database space because the changes you are making will result in different requirements for database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate database size.

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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
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 the database using the new DBD and one of the following:
   - Your load program
   - The HD 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 will need to run additional utilities before reloading your database. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Procedure for Changing to DEDBs

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 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. Rebuild the ACB if you are having ACBs pre-built rather than built dynamically.
5. 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.
6. Run the DEDB initialization utility (DBFUMIN0).
7. Run the user DEDB load program.

Changing the Hierarchic Structure

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 hierarchic sequence of segment types in your database record to improve performance. The second is combining segments to maximize the use of space.

Changes involving adding and deleting segments in the hierarchy are covered in “Chapter 15. Modifying Your Database” on page 365.

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 and HIDAM)

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. For detailed information on this report, see *IMS/ESA Utilities Reference: Database Manager*.

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 hierarchic 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

The second type of change you might need to make in the structure of your database record is combining segment types to maximize use of space. 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.)

Procedure for Changing the Hierarchic Structure

To change the hierarchic 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. Rebuild the ACB if you have ACBs pre-built rather then built dynamically.
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 using your load program and the new DBD. Remember to make an image copy of your database as soon as it is reloaded.
8. If your database uses logical relationships or secondary indexes, you must run some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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. For example, in a HIDAM database, only one data set containing data and another containing an index exists. When both are on the same device, extra time is required for seek operations as the arm moves between the index and database data sets. By putting one of the data sets on a different device, the amount of arm movement is decreased, thereby improving performance.

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. Information on calculating CI or block size is contained in “Chapter 6. Database Design Considerations for Full Function” on page 165 under “Determining the Size of CIs and Blocks”.
3. Code a new DBD.
4. Rebuild the ACB if you have ACBs pre-built rather than built dynamically.
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 the new DBD and the appropriate reload utility. 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 some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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. (Tuning of SB buffers is discussed in the section “OSAM Sequential Buffering” on page 355.)

Well-Organized Database

Well-organized databases are by far the most important of these two factors. When the databases SB processes are well organized, you note larger 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.

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. (For details on //DFSSTAT reports, see *IMS/ESA Utilities Reference: System*.) 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.

Ensuring a Well-Organized Database

You can ensure your databases are reasonably well-organized by:

- Providing 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). For details on how to do this, see “Specifying Free Space (HDAM and HIDAM Only)” on page 166.
- Selecting an appropriate database reorganization frequency. For more information on when and how to reorganize your databases, see “Reorganizing the Database” on page 325.
- Using efficient HDAM randomizing modules and randomizing parameters. Information on this can be found in the section “Determining Which Randomizing Module To Use (HDAM Only)” on page 168.

Adjusting HDAM Options

The HDAM options you can choose are described in “Chapter 6. Database Design Considerations for Full Function” on page 165 under “Choosing HDAM Options”. The performance implications of HDAM options are also discussed. To improve performance, reread that section and reassess the original choices you made.

You can adjust HDAM 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. If you changed your CI or block size, you need to allocate buffers for the new size. See “Chapter 6. Database Design Considerations for Full Function” on page 165 for a discussion of what things to consider in choosing buffer number and size and how they are specified.
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 pre-built rather than built dynamically.
6. Determine whether you need to recalculate database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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 and the appropriate reload utility. 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 some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Adjusting Buffers

The size and number of buffers you can choose are described in “Multiple Buffers in Virtual Storage” on page 174. This section also discusses the performance implications of choosing a buffer size and number. To improve performance, reread that section and reassess the original choices you made before you adjust your buffers.

VSAM Buffers

Monitoring VSAM Buffers

If you are using VSAM, you can monitor buffers using the DB monitor reports described in “Chapter 13. Monitoring Your Database” on page 309. For each buffer size you define, a VSAM subpool report is produced. 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).

When to Adjust VSAM Buffers

Adjust VSAM buffers when you see buffer performance begin to degrade, or if you wish to add options to boost performance in anticipation of increased buffer activity.

VSAM Buffer Adjustment Options

1. 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.
2. 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. *IMS/ESA Installation Volume 2: System Definition and Tailoring* tells you how to specify these options.
3. If 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 sequential mode. With 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. Sequential mode is not available in data sharing.

4. VSAM buffers can take advantage of MVS/ESA Hiperspace buffering.

Hiperspace Buffering Parameters: To use Hiperspace buffering, you must specify one or two optional parameters on the VSRBF subpool definition statement:

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.

Coding the HSO|HSR and HSn parameters to activate Hiperspace buffering on VSAM buffers is described in *IMS/ESA Installation Volume 2: System Definition and Tailoring*. See *MVS/ESA System Programming Library: Initialization and Tuning* for more information about Hiperspace.

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 which 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.

Related Reading: For detailed information on these formats see the *IMS/ESA Application Programming: Design Guide*.

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.

Related Reading:

- For detailed information about the DB Monitor Database Buffer Pool report, see the *IMS/ESA Utilities Reference: System*.
- For more information on the ***co*** (caching option) parameter of the IOBF control statement, OSAM buffer pools and the use of the coupling facility for OSAM data caching see the *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

Procedure for Adjusting VSAM and OSAM Database Buffers

To adjust VSAM and OSAM database buffers, change the control statements that specify buffer size and number. Then put the new control statements in the:

- DFSVSAMP data set in batch and utility environments
- IMS.PROCLIB data set with the member name DFSVSMnn in IMS TM and DBCTL environments

Detailed information on how to code these control statements is in *IMS/ESA Installation Volume 2: System Definition and Tailoring* .

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.

Procedure for Adjusting Sequential Buffers

You can change the number of buffer sets allocated to each SB buffer pool in two ways:

- Coding an SBPARM control statement with the BUFSETS keyword.
- Using 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 the *IMS/ESA Utilities Reference: Database Manager*.
- Detailed instructions on how to code an SBPARM control statement are in the *IMS/ESA Installation Volume 2: System Definition and Tailoring*.
- Details on the SB Initialization Exit Routine are in the *IMS/ESA Customization Guide*.

Adjusting VSAM Options

The VSAM options you can choose are described in "Determining Which VSAM Options to Use" on page 184. In "Chapter 4. Designing a Fast Path Database" on page 33, the performance implications of each VSAM option are also discussed. To improve performance, reread that section and reassess the original choices you made.

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/ESA Utilities Reference: System*. 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 may 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.

Two types of adjustable VSAM options exist:

- Options specified in the OPTIONS control statement
- Options specified in the Access Method Services `DEFINE CLUSTER` command

Procedure for Adjusting VSAM Options Specified in the OPTIONS Control Statement

To adjust these VSAM options, change the appropriate parameters in the OPTIONS control statement. Then put the new control statement in the:

- DFSVSAMP data set in a batch system

- IMS.PROCLIB data set with the member name DFSVSMnn in an online system

Detailed information on how to code these control statements is in *IMS/ESA Installation Volume 2: System Definition and Tailoring* .

Procedures for Adjusting VSAM Options Specified in the Access Method Service 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, as described in the following sections.

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. See “Chapter 12. Loading Your Database” on page 285, “Estimating the Minimum Size of the Database” on page 286 for a description of how to calculate database space.
3. Delete the old database cluster and define the new database cluster with a change to the FREESPACE parameter.
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.
5. If you changed the DBD, rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
6. 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.
7. If your database uses logical relationships or secondary indexes, you must run some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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.

Changing the IMBED / NOIMBED or REPLICATE / NOREPLICATE Parameter

You can use the reorganization utilities to change whatever you’ve specified in the IMBED|NOIMBED and REPLICATE|NOREPLICATE parameters. To change either or both:

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. See

"Chapter 12. Loading Your Database" on page 285, "Estimating the Minimum Size of the Database" on page 286 for a description of how to calculate database space.

3. Delete the old database cluster and define the new database cluster.
4. Reload your database using the existing DBD and the appropriate reload utility.
5. If your database uses logical relationships or secondary indexes, you must run some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Adjusting OSAM Options

The OSAM options you can choose are described in "Determining Which OSAM Options to Use" on page 190. Performance implications of each OSAM option are also discussed there. To improve performance, reread that section and reassess the original choices you made.

You cannot specifically monitor any OSAM options. To adjust OSAM options, change the appropriate parameters in the OPTIONS control statement. Then put the new control statement in the:

- DFSVSAMP data set in a batch system
- IMS.PROCLIB data set with the member name DFSVSMnn in an online system

Detailed information on how to code these control statements is in *IMS/ESA Installation Volume 1: Installation and Verification* .

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. The DL/I Call Summary report is described in *IMS/ESA Utilities Reference: System*. 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. (LISTCAT ALL report is described in "Chapter 13. Monitoring Your Database" on page 309.)
- For non-VSAM data sets, you can get a report on the VTOC using the `LISTVTOC` command. In the report, check the NOEXT field. (LISTCAT ALL report is described in "Chapter 13. Monitoring Your Database" on page 309.)

If you decide to change the amount of space allocated for your database, do it with JCL or with MVS 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. See “Chapter 12. Loading Your Database” on page 285 , “Estimating the Minimum Size of the Database” on page 286 for a description of how to calculate 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.
5. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically if you changed the DBD.
6. 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.
7. You must run some of the reorganization utilities before and after reloading to resolve prefix information if your database uses logical relationships or secondary indexes. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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, you:

1. Code a new DBD (unless you have already done this as described in Step 1).
2. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
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. These topics are covered in preceding sections of this chapter:

   “Adjusting Buffers” on page 353

   “Adjusting VSAM Options” on page 356

   “Adjusting OSAM Options” on page 358
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 database uses logical relationships or secondary indexes, you must run some of the reorganization utilities before and after loading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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. “Using Multiple Data Set Groups” on page 158 tells you:

- What data set groups are
- When they can be used
- What situations might prompt you to use them
- How they are specified in the DBD

Changing the Number of Data Set Groups

You should be familiar with these topics. You should also have decided to 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.

To change the number of data set groups in your database, (see Figure 156 on page 361) you:

1. Unload your database using the existing DBD.
2. Code a new DBD.
3. Rebuild the ACB if you have ACBs prebuilt rather than built dynamically.
4. Recalculate database space. You need to recalculate database space because the change you are making will result in different requirements for database space. See "Chapter 12. Loading Your Database" on page 285, "Estimating the Minimum Size of the Database" on page 286 for a description of how to calculate database space.
5. Reallocate data sets because the number and size of data sets you are using will change. See "Chapter 12. Loading Your Database" on page 285, "Allocating Data Sets" on page 293 for information on allocating data sets.
6. 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.
7. Reload your database using the new DBD. Remember to make an image copy of your database as soon as it's reloaded.
8. Run some of the reorganization utilities before and after reloading to resolve prefix information if your database uses logical relationships or secondary indexes. The flowchart in Figure 143 on page 328 shows you which utilities to use and the order in which they must be run.

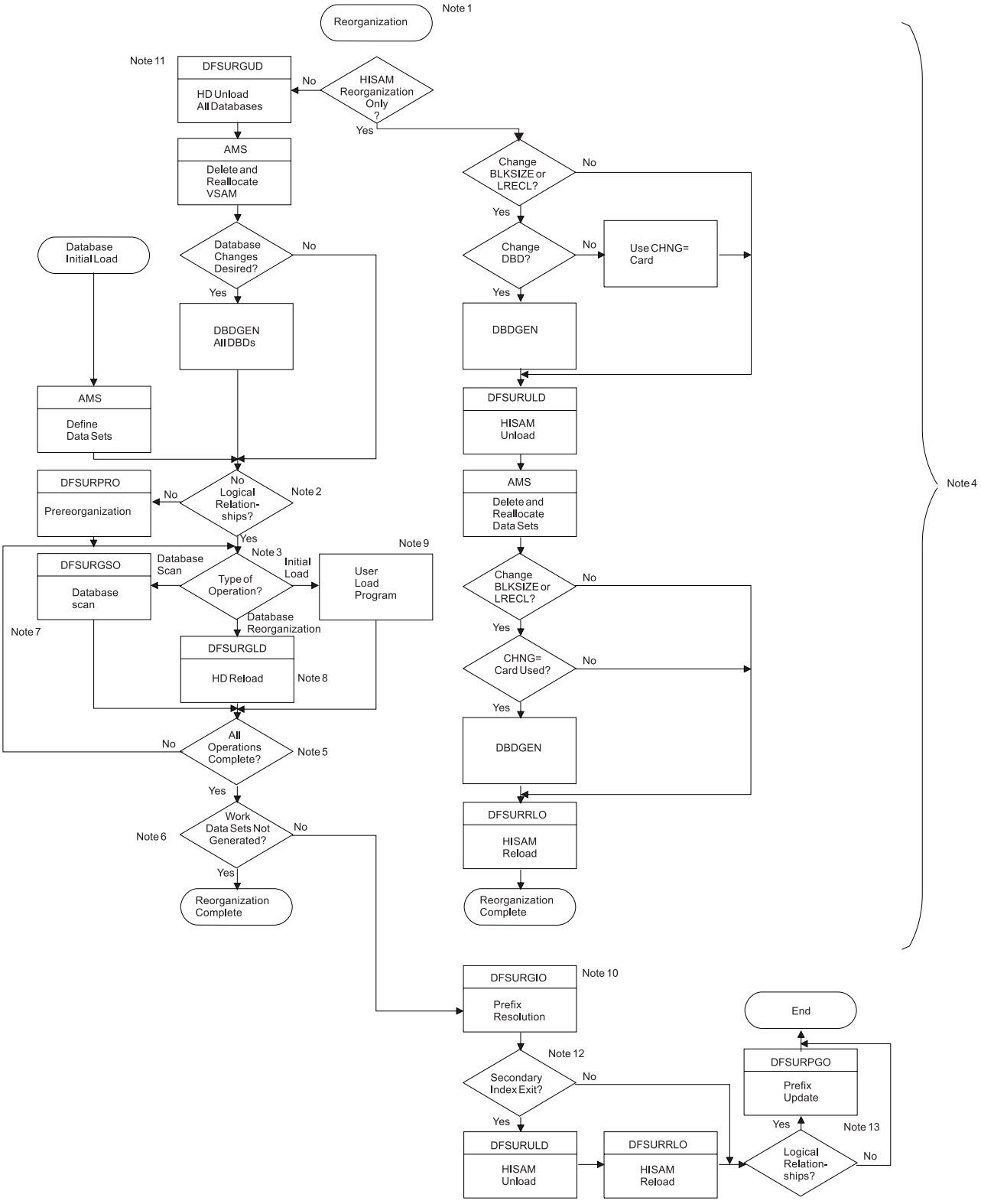


Figure 156. Utility Sequence of Execution When Making Database Changes during Reorganization

Notes to Figure 156:

1. You can use the database reorganization/load processing utilities (that is, the HISAM Unload/Reload, HD Unload/Reload, Prefix Resolution and Prefix

Update utilities) to operate 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.

2. 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 Pre-reorganization utility to determine which database operations must be performed.
3. Based upon the information given to it on control statements, the database Pre-reorganization 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 pre-reorganization control statement between reload and prefix resolution.
4. 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 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.
5. You must ensure that all operations indicated by the Prereorganization utility (if it was executed) are completed prior to taking the YES branch.
6. 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.
7. 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 Pre-reorganization 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 Pre-reorganization utility.
8. The HD Reorganization Reload utility can cause the generation of a work data set to be later used by the Prefix Resolution utility. Databases to be reorganized using the HD Unload/Reload utilities are listed after the character “DBR=” in one or more output messages of the Pre-reorganization utility.

9. 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 Pre-reorganization utility.
10. The database Prefix Resolution utility combines the workfile output from the Database Scan utility, the HD Reorganization Reload utility, and the user's initial database load execution to create an output data set for use by the Prefix Update utility. The Prefix Update utility then completes all logical relationships defined for the databases that were operated upon.
11. This path must be taken for HISAM databases with logical relationships. This path must also be taken if structural changes are required (for example, HISAM to HDAM, pointer changes, additional segments, or adding a secondary index).
12. 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.
13. For information on scratching and allocating OSAM data sets, see "Allocation for OSAM Data Sets Using IEFBR14" under "Designing the IMS Online System" in *IMS/ESA Administration Guide: System*.

Chapter 15. Modifying Your Database

About This Chapter

Under several circumstances, you must modify your database. Over time, user requirements can change, necessitating changes in the database design. Or you might chose to use new or different options or features. Or perhaps you have simply found a more efficient way to structure the database. This chapter describes the various types of structural changes you can make to your database and tells you when and how you can make the changes using the reorganization utilities.

This chapter examines the following areas of modifying a database:

- Adding segment types
- Deleting segment types
- Moving segment types
- Changing segment size
- Changing data in a segment
- Changing the position of data in a segment
- Adding logical relationships
- Adding a secondary index
- Adding or converting to variable-length segments
- Converting to the segment edit/compression facility
- Converting databases for Data Capture exit routines and Asynchronous Data Capture
- Converting to multiple data set groups [3]
- Converting a logical parent concatenated key from virtual to physical or physical to virtual
- Using the online example function
- Extending DEDB independent overflow online

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 section has procedures to guide you through making each type of change. If you

3. Conversion to multiple data set groups is a type of change you might also make to tune your database. Therefore, information on this topic is discussed in “Changing the Number of Data Set Groups” on page 359.

make more than one change at a time, you should look at Figure 156 on page 361. The flowchart, when used with the individual procedures in this chapter, will guide you in making some types of multiple changes to the database.

If you share data, additional information about modifications is in *IMS/ESA Administration Guide: System* .

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.

Adding Segment Types

There are three ways to add a segment type to a database:

- 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

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. Figure 157 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.

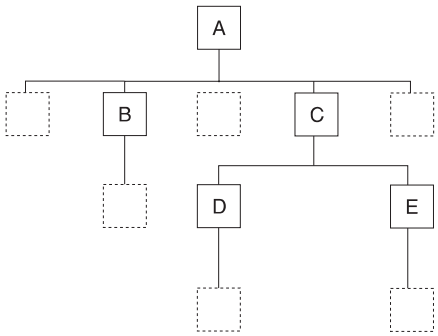


Figure 157. Where Segment Types Can Be Added in a Database Record

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. If you have the DB/DC Data Dictionary, it can help you determine the application programs and PCBs that 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 will result in different requirements for database space. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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, run some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.
10. Code and execute an application program to insert the new segment types into the database.

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 hierarchic 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

You must write your own unload and reload program to add a segment type to the database, if the segment type does not meet the qualifications described in the two preceding sections.

Deleting Segment Types

You can delete a segment type from a database by:

- Using the reorganization utilities
- 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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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 some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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 some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

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 the IMS System Utilities/Database Tools (DBT) DB Segment Restructure Utility. See “Using Field-Level Sensitivity” for a description of how this function works.

Adding Logical Relationships

Logical relationships are explained in detail in “Chapter 4. Designing a Fast Path Database” on page 33 under “Using Logical Relationships”. This section 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 section. 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 in this section are followed by Table 13 on page 383 which tells you what to do when reorganizing a database to add a logical relationship. Following the figure are some restrictions on modifying existing logical relationships.

The examples in this section 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.

Example 1. DBX Exists, DBY Is to Be Added

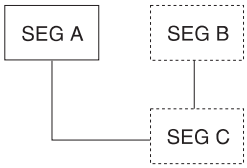


Figure 158. DBX Exists, DBY Is to Be Added

DBX must be reorganized to add the counter field to the segment prefix for A. DBIL must be specified in the control statement for DBX. In the following procedure, 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.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83, explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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 2. DBX and DBY Exist, DBZ Is to Be Added

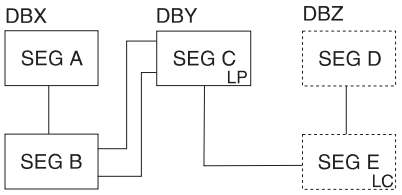


Figure 159. DBX and DBY Exist, DBZ Is to Be Added

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. DBIL must be specified in the control statement for DBX and DBY. In the following procedure, the segment A counter field is updated to show the number of C segments because segment C is loaded with a user load program.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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.

Example 3. DBX and DBY Exist, DBZ Is to Be Added

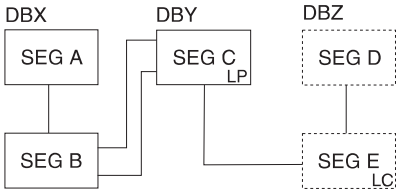
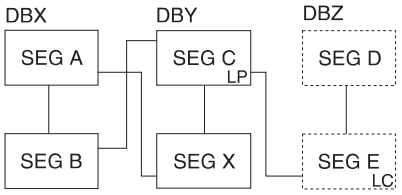


Figure 160. DBX and DBY Exist, DBZ Is to Be Added

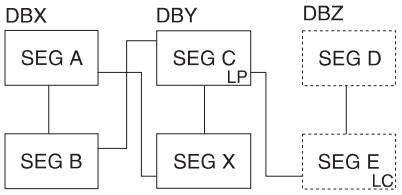
DBY must be reorganized to add the counter field to the segment C prefix. DBIL must be specified in the control statement for DBY. DBX must be reorganized because an initial load (DBIL) of the logical parent (segment C) assumes an initial load (DBIL of the logical child). The procedure for this example (and all conditions and considerations) is exactly the same as example 2.

Example 4. DBX and DBY Exist, DBZ Is to Be Added



The procedure for this example (and all conditions and considerations) is exactly the same as for example 2.

Example 5. DBX Exists, DBY Is to Be Added



DBX must be reorganized to add the logical child pointers in the segment A prefix.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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 6. DBX and DBY Exist, DBZ Is to Be Added

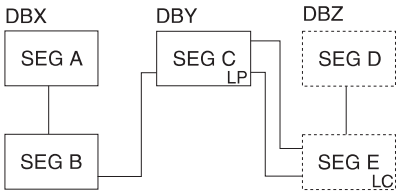


Figure 161. DBX and DBY Exist, DBZ Is to Be Added

DBY must be reorganized to add the logical child pointers to the segment C prefix. One of the following three options should be used.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization utility, specifying DBR in the control statement for DBY, and DBIL in the control statement for DBZ. (The output from the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization utility, specifying DBR in the control statement for DBY, and DBIL in the control statement for DBZ. (The output from the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.

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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization utility, specifying DBR in the control statements for DBX and DBY, and DBIL in the control statement for DBZ. (The output from the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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.

Example 7. DBX and DBY Exist, DBZ Is to Be Added

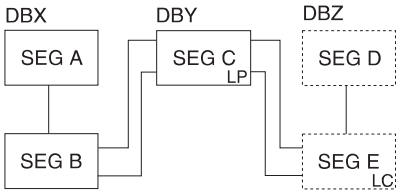


Figure 162. DBX and DBY Exist, DBZ Is to Be Added

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.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization utility, specifying DBR in the control statements for DBY, and DBIL in the control statement for DBZ. (The output from the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization utility, specifying DBR in the control statements for DBX and DBY, and DBIL in the control statement for DBZ. (The output from the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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.

Example 8. DBX and DBY Exist, DBZ Is to Be Added

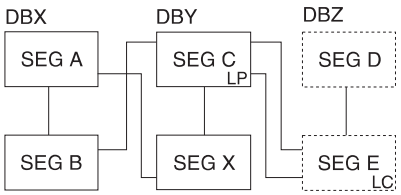


Figure 163. DBX and DBY Exist, DBZ Is to Be Added

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) is exactly the same as the procedures for example 6.

Example 9. DBY Exists, DBZ Is to Be Added

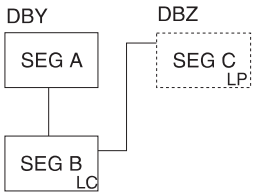


Figure 164. DBY Exists, DBZ Is to Be Added

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.

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. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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

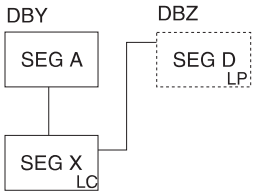


Figure 165. DBY Exists, DBZ Is to Be Added

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 11. DBX and DBY Exist, DBZ Is to Be Added

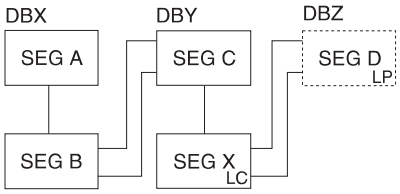


Figure 166. DBX and DBY Exist, DBZ Is to Be Added

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) is exactly the same as for Example 2.

Example 12. DBX and DBY Exist, DBZ Is to Be Added

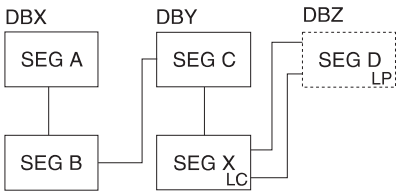


Figure 167. DBX and DBY Exist, DBZ Is to Be Added

In this example, segment B has a symbolic pointer. The procedure for this example (and all conditions and considerations) is exactly the same as for example 2.

Example 13. DBX and DBY Exist, Segment Y and DBZ Are to Be Added

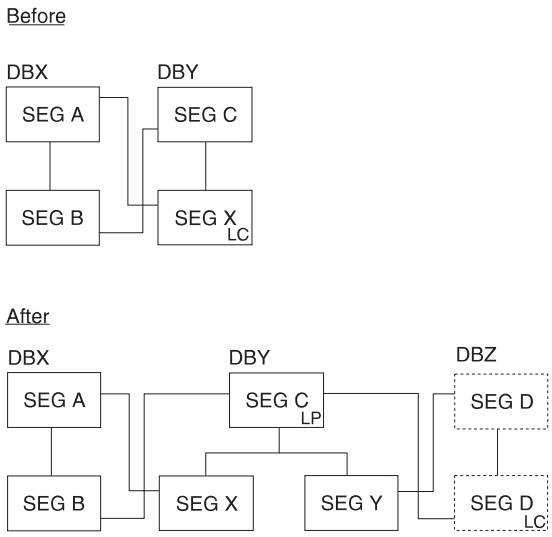


Figure 168. DBX and DBY Exist, Segment Y and DBZ Are to Be Added

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 HD Unload utility.
3. Code a new DBD for DBY and DBZ. “How to Specify Use of Logical Relationships in the Logical DBD” in “Chapter 5. Choosing Additional Database Functions” on page 83 explains how the DBD is coded for logical relationships.
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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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. Run the Pre-reorganization 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. See “Chapter 12. Loading Your Database” on page 285 under “Writing a Load Program” for a description of how to write 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.

Steps in Reorganizing a Database to Add a Logical Relationship

Table 13 on page 383 shows you:

- 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

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.

Table 13 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 Pre-reorganization utility is run.

The following two examples guide you in use of the figure:

Example 1. How to use Table 13

Assume your database has unidirectional symbolic pointers and you are not changing pointers. On the left side of Table 13, in the FROM column, find unidirectional symbolic pointers. The follow across to the right in the TO row and find 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. How to use Table 13

Assume your database has bidirectional symbolic pointers and you need to change to bidirectional direct pointers. Table 13 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 Pre-reorganization 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 Pre-reorganization 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 13. Steps in Reorganizing a Database to Add a Logical Relationship

		What You Must Do to Reorganize When You Need:			
Type of Database	**Type of Reorganization**	**Unidirectional Symbolic Pointers**	**Unidirectional Direct Pointers**	**Bidirectional Symbolic Pointers**	**Bidirectional Direct Pointers**
Unidirectional with symbolic pointers	Logical parent database only	Finished*	Not valid, because symbolic LP pointers exist now and direct LP pointers must be added to the logical child database.	1. Scan logical child data base. 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 data base. 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**	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 13. Steps in Reorganizing a Database to Add a Logical Relationship (continued)

		What You Must Do to Reorganize When You Need:			
Type of Database	**Type of Reorganization**	**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 data base. 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 data base. 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**	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 data base. 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 data base. 2. Run prefix resolution and update.	1. Scan logical parent data base. 2. Run prefix resolution and update. 3. Specify DBIL for the logical child data base.
	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 13. Steps in Reorganizing a Database to Add a Logical Relationship (continued)

		What You Must Do to Reorganize When You Need:			
Type of Database	**Type of Reorganization**	**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 data base. 2. Run prefix resolution and update.	1. Scan logical parent data base. 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:

* The preorganization utility says to scan the logical child database and the DFSURWF1 records will be produced if scan is run.

** DFSURWF1 records are produced; however, the prefix resolution and update utilities need *not* be run.

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. Two examples are as follows:

Example 1: Changing from Bidirectional Virtual to Bidirectional Physical Pairing

This example shows the change in pairing from virtual to physical:

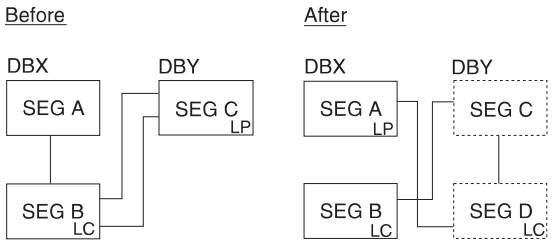


Figure 169. The Change in Pairing from Virtual to Physical

Example 2: Changing the Location of the Real Logical Child in a Bidirectional Logical Relationship

This example shows the position change of a real logical child from one logically related database to another:

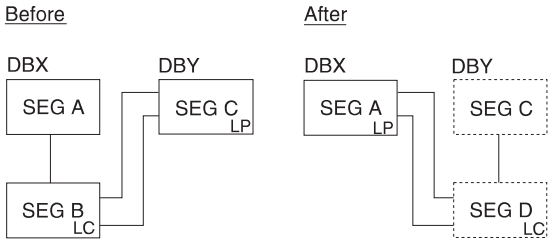


Figure 170. The Position Change of a Real Logical Child From One Logically Related Database to Another

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

- 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.

Adding a Secondary Index

Secondary indexes are explained in “Chapter 5. Choosing Additional Database Functions” on page 83. If you need to add a secondary index to your database:

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 new DBDs. “How to Specify Use of Secondary Indexing in the DBD” in “Chapter 4. Designing a Fast Path Database” on page 33 explains how the DBD is coded for secondary indexes. You need two 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. 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 database, using the new DBD and the HD Reload utility.
8. Run the Prefix Resolution utility, using the DFSURWF1 work file that is output from Step 7 as input.
9. Run the HISAM unload utility, using the DFSURIDX work file that is output from Step 8 as input. Be sure to indicate in the utility control statement that HISAM unload is being used to build a secondary index.
10. Run the HISAM reload utility using as input the output from HISAM unload.
11. When you add a secondary index, remember to change your JCL. You need a DD statement for the secondary index data set even when you are not using the secondary index to process the main database. You also need to change your reorganization procedures when adding a secondary index. Whenever you reorganize the data set the secondary index points to, you must execute the reorganization utilities to rebuild the secondary index.

Adding or Converting to Variable-Length Segments

Variable-length segments are explained in “Chapter 5. Choosing Additional Database Functions” on page 83. 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.

Method 1. Converting Segments or a 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

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 use of the segment edit/compression facility 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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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 facility can exit. Your edit routine should add a size field to each segment it receives. (Information on the segment edit/compression facility and the edit routine you must write is contained in “Chapter 5. Choosing Additional Database Functions” on page 83 under “Using the Segment Edit/Compression Facility”.)
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 some of the reorganization utilities before and after reloading to resolve prefix information. The flowchart in Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.
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.

Converting to the Segment Edit/Compression Facility

The segment edit/compression facility is explained in “Chapter 5. Choosing Additional Database Functions” on page 83 under “Using the Segment Edit/Compression Facility”. If you need to convert an existing database so it can use the facility, 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 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. See “Chapter 12. Loading Your Database” on page 285 under “Estimating the Minimum Size of the Database” for a description of how to calculate 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 some of the reorganization utilities before and after reloading to resolve prefix information. Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.

Converting Databases for Data Capture Exit Routines and Asynchronous Data Capture

This section contains general-use programming interface information.

Data Capture exit routines are explained in “Using Data Capture Exit Routines” on page 145. 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. See *IMS/ESA Utilities Reference: System* for details on the DBD parameters required for Data Capture exit routines or Asynchronous Data Capture. *IMS/ESA Customization Guide*explains the exit routines in detail, how to code them, and how they work.
3. Run DBDGEN.
4. If you use prebuilt ACBs rather than dynamically built ACBs, rebuild the ACB.

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 do this conversion:

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. See “Estimating the Minimum Size of the Database” on page 286 for a description of how to calculate database space.

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 some of the reorganization utilities before and after reloading to resolve prefix information. Figure 143 on page 328 tells you which utilities to use and the order in which they must be run.
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.

Using the Online Change Function

Adding, changing, and deleting databases (except MSDBs) online without stopping IMS can be done using 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 randomizer 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 which 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 affected need to be stopped.

To use online change, you must do the following:

1. Allocate the required new data sets (see *IMS/ESA Installation Volume 1: Installation and Verification* for planning these data sets).
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/ESA Administration Guide: System* for more information).
3. Run the necessary DBDGEN (see *IMS/ESA Utilities Reference: Database Manager*), PSBGEN, and ACBGEN (see *IMS/ESA Utilities Reference: System*).

   Note: *All* changes to ACBLIB members resulting from the ACBGEN execution are available to the online system after the online change (provided that the changed resources—PSBs and DBDs—are defined in the online system).
4. Run the Security Maintenance utility if IMS security must be defined for new databases (see *IMS/ESA Installation Volume 1: Installation and Verification* for information on the Security Maintenance utility).
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 an MVS environment, run the dynamic allocation utility.

9. Use the online change utility to copy your updated staging libraries to the inactive libraries (see *IMS/ESA Utilities Reference: System* for information on running this utility).
10. Issue the operator commands to cause your inactive libraries to become your active libraries (see *IMS/ESA Operator's Reference* for the commands used).

If a database in an MVS environment needs to be reorganized because of changes to the active ACBLIB data set, /DBR must be issued to deallocate the database prior to 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 prior to issuing the `/MODIFY COMMIT` command.

Maintaining Continuous Availability of IFP and MPP Regions

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.

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 pseudo-abend 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.

Using the Online Change Function

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.
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 pseudo-abend 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 database 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 Reading: See the *IMS/ESA Messages and Codes* for more information on this and other messages.

If area level changes are made to DEDBs while a BMP or DBCTL thread is active, then on the next access to the newly changed area, the application should continue processing as usual.

Changing Randomizer and Exit Routines

Randomizer 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.

New Randomizer Routine

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 described below:

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 linkedit the new randomizer into the IMS RESLIB or one of the libraries in the IMS RESLIB 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. ACBLIB 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. Enter the normal /DBR operator command sequence to remove access to the DEDB from the active IMS system.
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. Enter the normal /START DATABASE and /START AREA commands to make the DEDB and its AREAs accessible to the active IMS system.
11. Use a specific customer-developed application program or OEM utility to reload the DEDB with the new randomizer routine in effect.

Changed Randomizer Routine

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. THERE IS NO WARNING OR INDICATION OF THE FOLLOWING:

- The changed randomizer has not been loaded.
- The old version is still being used.

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. Enter the normal /DBR DATABASE operator command sequence to remove access to the DEDBs from the active IMS system. The /DBR DATABASE command unloads the randomizer for the DEDBs designated as operands.

When all the DEDBs that reference the randomizer are stopped, the randomizer is removed from the active IMS system. If a DEDB is not stopped and references a randomizer that has been removed from the IMS system, then a U1021 abend results on the next DL/I call.

3. Assemble and linkedit the changed randomizer into the IMS RESLIB or one of the libraries of the IMS RESLIB 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. Enter the /START DATABASE command for each of the DEDBs that use the changed randomizer. For DEDBs, the /START DATABASE command causes the randomizer to be loaded.
6. Use a specific customer-developed application program or OEM utility to reload the DEDB with the changed randomizer routine in effect.

Deleted Randomizer Routine

To delete a randomizer from the active IMS system, follow the procedural steps that are documented under ″New Randomizer Routine″. Once all the DEDBs that were using the old randomizer have been unloaded and had the /DBR command run successfully against them, then the randomizer can be deleted. Customers with data sharing IMS systems that do not share RESLIBs must be careful to delete the randomizer from both systems. A message (DFS2838) is generated when the randomizer is deleted.

Adding, Changing or Deleting 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 randomizer routines.

Adding, Changing or Deleting Data Capture Exit Routines

Data Capture exit routines are defined for the DEDB on the DBD statement and/or for a specific segment on the SEGM statement (″EXIT=″). Multiple exit routines can be specified on a single DBD or SEGM statement.

Adding a New Data Capture Exit Routine: To add a new Data Capture exit routine, follow the procedure below:

1. Assemble and linkedit the new exit routine into the IMS RESLIB or one of the libraries in the IMS RESLIB 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 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 Data Capture Exit Routine: 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 linkedit the changed exit routine into the IMS RESLIB or one of the libraries of the IMS RESLIB 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 re-usable are unloaded at dependent region termination time. Otherwise, they are unloaded after every DL/I call.

Deleting a Data Capture Exit Routine: 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. ACBGEN is also needed to build the application control blocks 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 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 Root Addressable Space with Two Stage Randomizer

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 section is not applicable to such changes.

If, however, a ″Two Stage″ randomizer is used for the DEDB, a change to an individual AREA UOW root addressable definition is an *AREA Level* change. A ″Two Stage″ randomizer 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″ randomizer is designated in the DBDGEN by coding the randomizer 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 randomizer routines.

Changing the DEDB AREA UOW Structural Definition

Changing the DEDB AREA UOW structural definition requires the following procedural 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.

Making Online Changes at the DEDB and Area Level

Adding or Deleting DEDBs

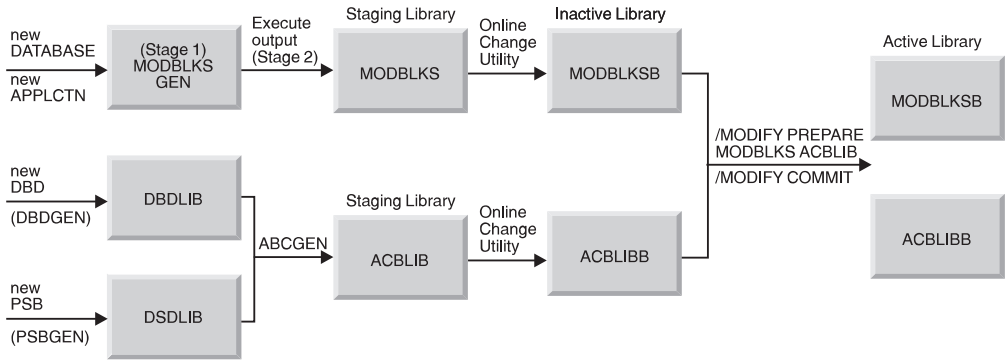


Figure 171. Adding a Database Using Online Change

Adding or deleting a DEDB and implementing the change by means of the IMS online change facility requires that you follow the steps described below. See Figure 171 for an overall picture.

1. MODBLKs Level SYSGEN (Stage 1 and Stage 2) to add or delete the DEDB. The changed MODBLKs should be generated into the active IMS system's staging copy of MODBLKs, 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 ACBLIB, which is offline.
3. Run the online change utility, DFSUOCU0, to move the changed MODBLKs and ACBLIB changes 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=), then message DFS2832 will appears and the PREPARE process aborts.

   Finally, if a DEDB is added to an IMS system that was initialized without any DEDBs, then message DFS2837 appears and the PREPARE process aborts.

   Othreads 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.

Related Reading: See the *IMS/ESA Messages and Codes* for more information on messages.

Changing DEDBs by Adding or Deleting Segments

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 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

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 “Changing DEDBs by Adding or Deleting Segments” on page 398 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 Root Addressable Space Allocation

Random Distribution of DB Records Across All 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 Randomizers: Standard DEDB randomizing routines attempt to evenly distribute database records across all AREAs and within the selected AREA. Such randomizers 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.

Changing Dependent and Independent Overflow Space Allocation

Starting in IMS Version 3, Fast Path has provided limited support for extending DEDB AREA Independent Overflow space allocation. That support continues unchanged. 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. See “Changing the DEDB AREA UOW Structural Definition” on page 396 for details.

Changing CI Size

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 randomizer 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.

If the new CI size of an AREA exceeds the Fast Path buffer size of the system, then the DFS2832 error message will appear during the PREPARE stage and the process will be aborted.

Extending DEDB Independent Overflow Online

You can extend the independent overflow (IOVF) portion of a DEDB area while IMS is online by following the procedure described in this section. 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 you must:

1. Run the DBDGEN utility to obtain an updated DBD. Update *only* the following operands on the ROOT= keyword of the AREA statement:

   number
   specifies the total number of units of work (UOWs) allocated to the root addressable and the IOVF parts of the area. Increase **number** to reflect the number of UOWs you need to add to the IOVF.

   overflow
   specifies the space reserved for the IOVF, expressed as the number of UOWs. Increase the number on this operand by the same amount you increase the **number** operand. For example, if the original values were **number**=*x* and **overflow**=*y*, and if **number** is changed to *x* + 2, then **overflow** must be changed to *y* + 2.

   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.

Related Reading:

- For more information on the DBRC definitions for the shared AREAs with SDEP segments, see the *IMS/ESA DBRC Guide and Reference*.
- For more information on SDEP scan utility keywords and change boundaries, see the *IMS/ESA Utilities Reference: Database Manager*.
- For more information on the SDEP scan utility user-written exit routine parameter interface, see the *IMS/ESA Customization Guide*.

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. See *MVS/DFP Access Method Services for the Integrated Catalog Facility* 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.

    Note: It is recommended that you 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.

    IMS/ESA Messages and Codes explains message DFS3709A and the reason for this failure.
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: Remember that you cannot use the online change function to update the ACBLIB with the altered ROOT= parameter.

Chapter 16. Establishing Security

The two aspects of database security are as follows:

- *User verification* (how you establish that the person using an online database is in fact the person you have authorized)
- *User authority* (once you have verified the user's identity, how you control what is seen—and what can be done with what is seen)

This chapter deals primarily with how you can control a user's view of data and the user's actions with respect to the data. (For a discussion of user verification, see "Establishing Security" in *IMS/ESA Administration Guide: System*).

This chapter examines the following areas:

- Restricting the scope of data access
- Restricting processing authority
- Restricting access of non-IMS programs
- Using the dictionary to help establish security

CICS users should see *CICS/ESA Facilities and Planning Guide* for information on establishing security.

Restricting the Scope of Data Access

The PCB defines a program's (and therefore the user's) view of the database. The PCB can be thought of as a "mask" over the data structure defined by the DBD, hiding certain parts of it. Therefore, it is possible, simply by limiting the scope of the PCB, to limit the user's access to (and even knowledge of) elements of the database you need to restrict.

Figure 172 on page 404 shows an example. The top of the figure shows the data structure for a PAYROLL database as perceived by you and defined by the DBD. For certain applications it is not necessary (nor desirable) to access the SALARY segment. Through SENSEG statements, 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. The bottom of the figure shows this modified data structure as perceived by the application programmer.

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 segment key 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. You can do

this through the PROCOPT parameter of the SENSEG statement and through the PCB statement. The PROCOPT statement tells IMS what actions you will permit against the database. A program can do what is declared in the PROCOPT.

In addition to restricting access and authority, 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.

```
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=
:
```

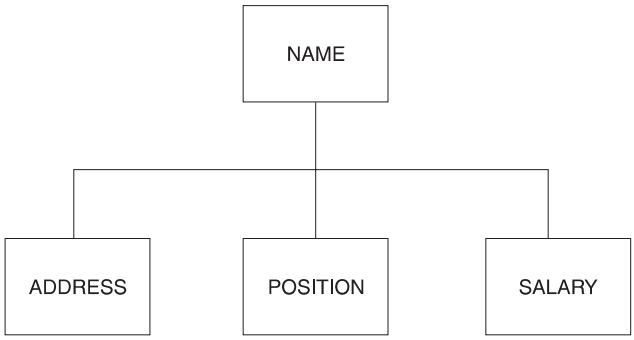


Figure 172. The NAME data structure

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

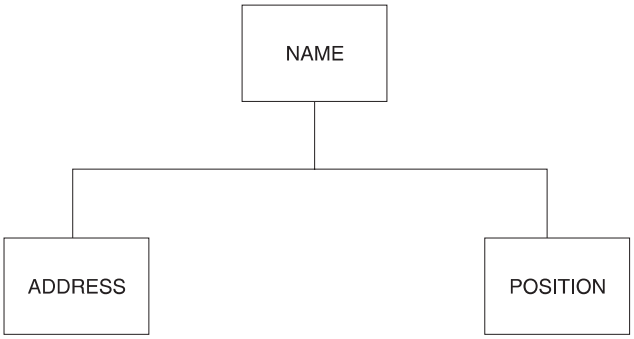


Figure 173. Using the PCB to Mask Segments

Restricting Access by Non-IMS Programs

One potential security exposure is from people attempting to access IMS/ESA 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, specify password protection for your VSAM data sets and code PASSWD=YES on the DBD statement. IMS then passes the DBD name as the password. If you specify PASSWD=NO on the DBD statement, 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.)

Details of the PASSWD parameter of the DBD statement can be found in *IMS/ESA Utilities Reference: System* .

Encrypting Your Database

Another precaution you can take against non-IMS programs reading DL/I databases is to encrypt the databases. 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.

You can learn more about segment edit/compression routines in “Using the Segment Edit/Compression Facility” on page 142.

Using the Dictionary to Help Establish Security

The dictionary monitors relationships among entities in your computing environment (such as, which programs use which data elements). This ability makes the dictionary the ideal tool to administer security.

Using the Dictionary to Help Establish 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

Appendix A. Meaning of Bits in the Delete Byte

This appendix examines the meanings of:

- Bits in the delete byte
- Bits in the prefix description byte

Bits in the Delete Byte

This section contain diagnosis, modification or 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).

Bits in the Prefix Descriptor Byte

This section contains diagnosis, modification, or tuning information.

The delete byte is also used for the root segment of a DEDB, only there it is called a prefix descriptor byte. 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.

This concludes the appendix, “Meaning of Bits in a Delete Byte”. Appendix B deals with replacing, inserting, and deleting rules for logical relationships, which includes how to specify rules in a physical DBD and a rules summary.

Appendix B. Replace, Insert, and Delete 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 appendix examines the following information on rules:

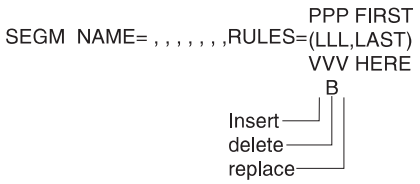
- How to specify rules in the physical DBD
- Insert, delete, and replace rules summary

How to Specify Rules in the Physical DBD

This appendix contains general-use programming interface information.

Here is how insert, delete, and replace rules are specified in the DBD:

How to Specify Rules in the Physical DBD



where:

P = physical
L = ogical
V = virtual
B = bidirectional virtual (for delete only)

Figure 174. How Insert, Delete, and Replace Rules are Specified in the DBD

The operands in the RULES= parameter are positional. Position 1 defines the insert rule, position 2 defines the delete rule, and position 3 defines 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.

The 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. Figure 175 on page 411 shows an example of the physical replace rule.
- 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. Figure 176 on page 412 shows an example of the logical replace rule.
- When RULES=V is specified, the segment can be replaced when retrieved by either a physical or logical path. Figure 177 on page 413 shows an example of the virtual replace rule.

The Replace Call

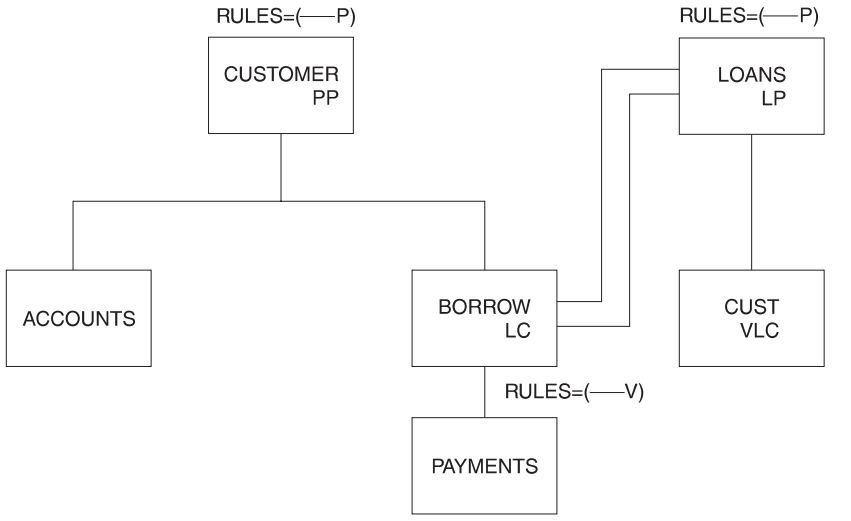
A replace operation can be done only on that portion of a concatenated segment to which an application program is data sensitive. If no data is changed in a segment, no data is replaced. Therefore, no replace rule is violated. The replace rule is not checked for a segment that is part of a concatenated segment but is not retrieved.

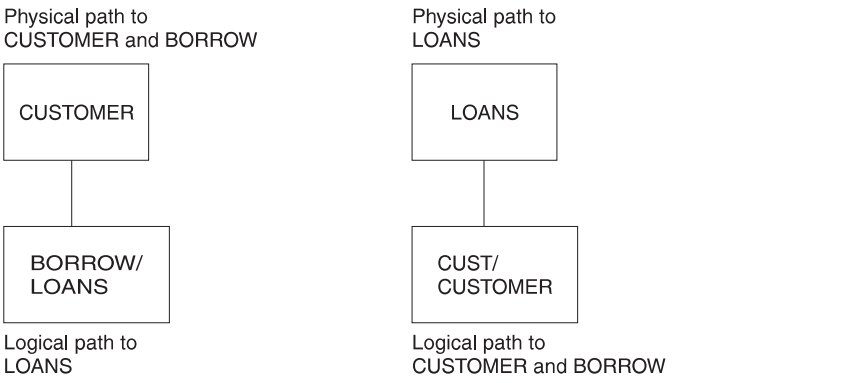
For all DL/I calls, either an error is detected and an error status code returned (in which case no data is changed), or the required changes are made to all segments affected by the call. Therefore, if the required function cannot be performed for both parts of the concatenated segment, an error status code is returned, and no change is made to either the logical child or the destination parent.

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 PROCOPTR
- DA—the key field of a segment or a non-replaceable field was changed
- RX—the replace rule was violated






```
GHU      'CUSTOMER'          STATUS CODE='bb'

REPL                         STATUS CODE='bb'

GHN      'BORROW/LOANS'      STATUS CODE='bb'

REPL                         STATUS CODE='RX'
```

Figure 175. 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.

How to Specify Rules in the Physical DBD

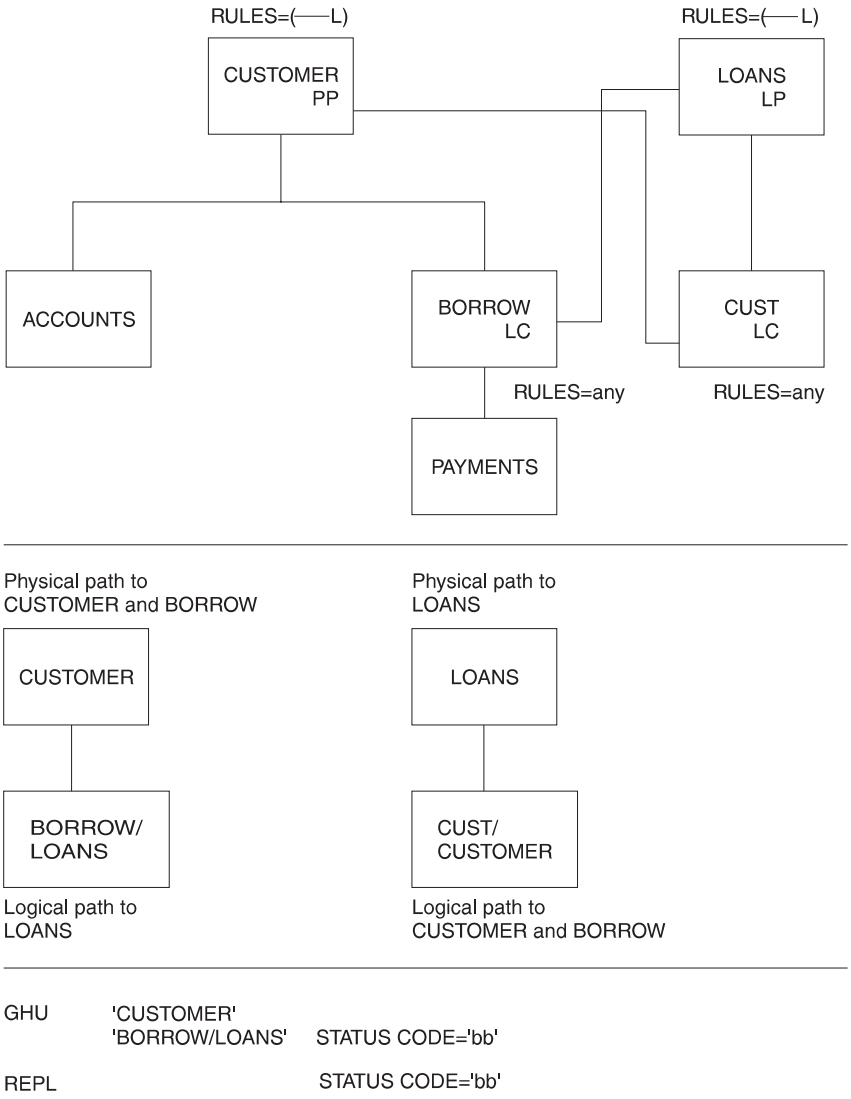
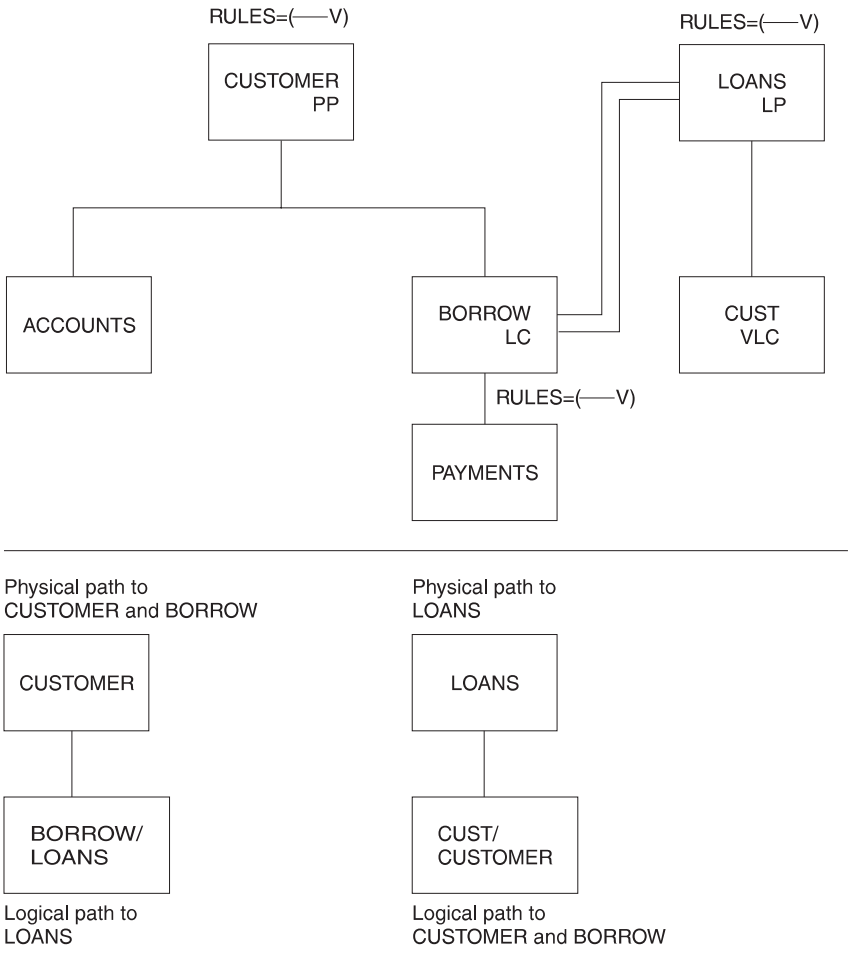


Figure 176. Logical Replace Rule Example

As shown in Figure 176, 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.




```
GHU     'LOANS'
        'CUST/CUSTOMER'     STATUS CODE='bb'

REPL                        STATUS CODE='bb'
```

Figure 177. Virtual Replace Rule Example

As shown in Figure 177, the V replace rule allows replacing the CUSTOMER segment using its logical path as part of a concatenated segment.

Replace Rules Summary

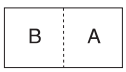
Specifying the replace rule as P, on any segment in a logical relationship, prevents replacing that segment except when it is retrieved using its physical path. When the replace rule for the logical parent is specified as L, IMS returns a blank status code without replacing any data when the logical 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.

Figure 178 on page 414 shows all possible combinations of replace rules that can be specified. It shows what actions take place for each combination when a call is issued to replace a concatenated segment in a logical database.

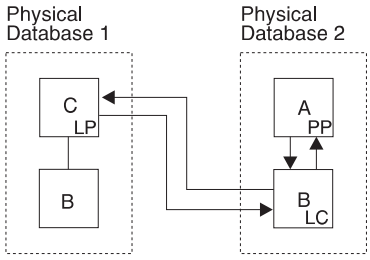
Logical View 1

Replace rule specified		Denotes segment you're attempting to replace		Status code	Data replaced? Y=Yes N=No	
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
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

Logical View 2



Replace rule specified		Denotes segment you're attempting to replace		Status code	Data replaced? Y=Yes N=No	
B	A	B	A		B	A
P	P	X		RX	Y	
P	P		X	RX		N
P	P	X	X	RX	N	N
P	L	X		RX	Y	
P	L		X			N
P	L	X	X	RX	Y	N
P	V	X		RX	Y	
P	V		X			Y
P	V	X	X	RX	Y	N
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
L	V	X			Y	
L	V		X			Y
L	V	X	X		Y	Y
V	P	X		RX	Y	
V	P		X	RX		N
V	P	X	X		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



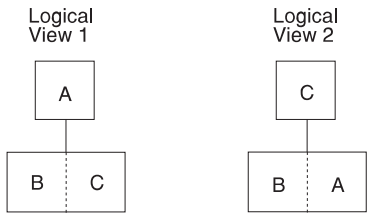


Figure 178. Replace Rules

The 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. Figure 179 on page 416 shows an example of the physical insert rule.
- 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. Figure 180 on page 417 shows an example of the logical insert rule.
- 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. Figure 181 on page 418 shows an example of the virtual insert rule.

The Logical Child Insert Call

To insert the logical child segment, the I/O area in an application program must contain one of the following segments in accordance with the destination parent's insert rule:

- The logical child
- The logical child/destination parent concatenated segment

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.

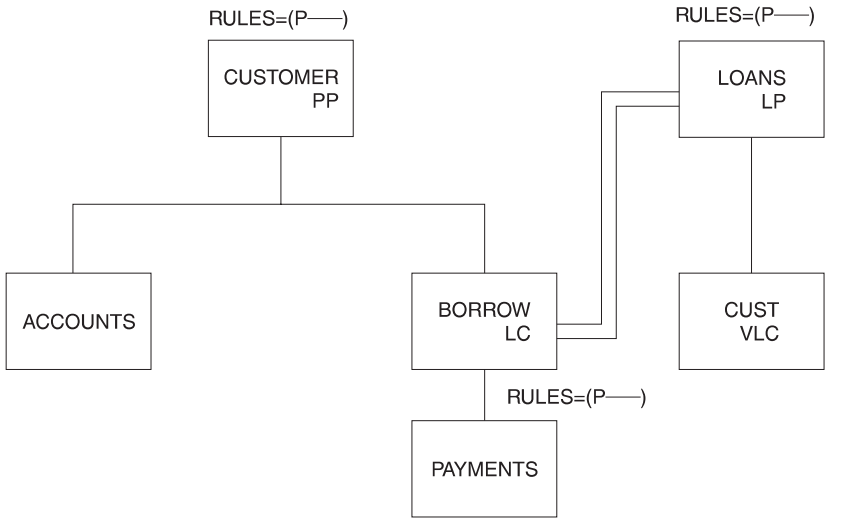
The insert operation is not affected by KEY or DATA sensitivity as specified in a logical DBD or a PCB. This means that if a program is other than DATA sensitive to both the logical child and destination parent of a concatenated segment, and if the insert rules is L or V, the program must still supply 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.

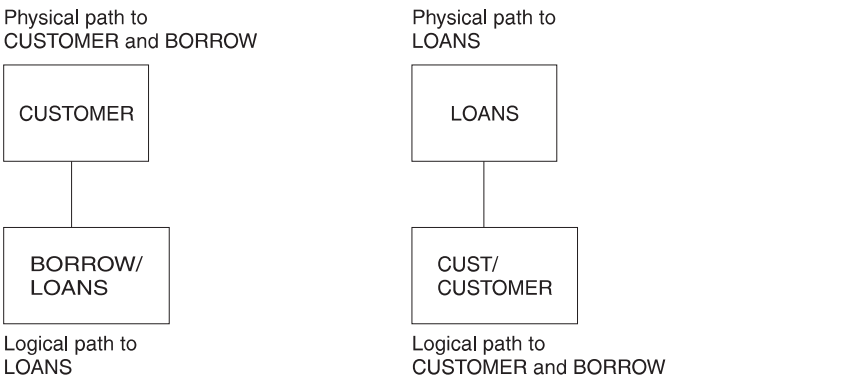
Status Codes

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 PROCOPTI
- 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.





If the LOANS segment does exist, then:

ISRT	'CUSTOMER'	STATUS CODE='bb'
ISRT	'BORROW'	STATUS CODE='bb'

However, if the LOANS segment does not exist, then:

ISRT	'CUSTOMER'	STATUS CODE='bb'
ISRT	'BORROW'	STATUS CODE='IX'

Figure 179. Physical Insert Rule Example

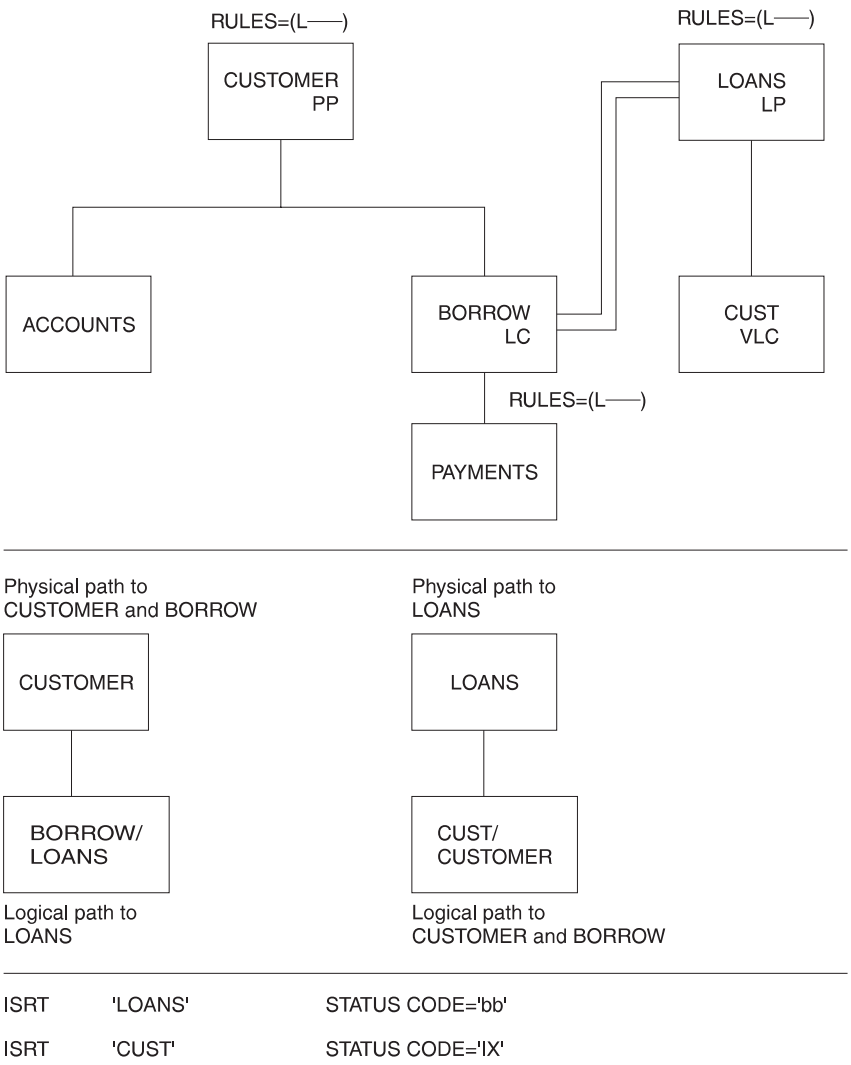


Figure 180. Logical Insert Rule Example

The IX status code shown in Figure 180 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.

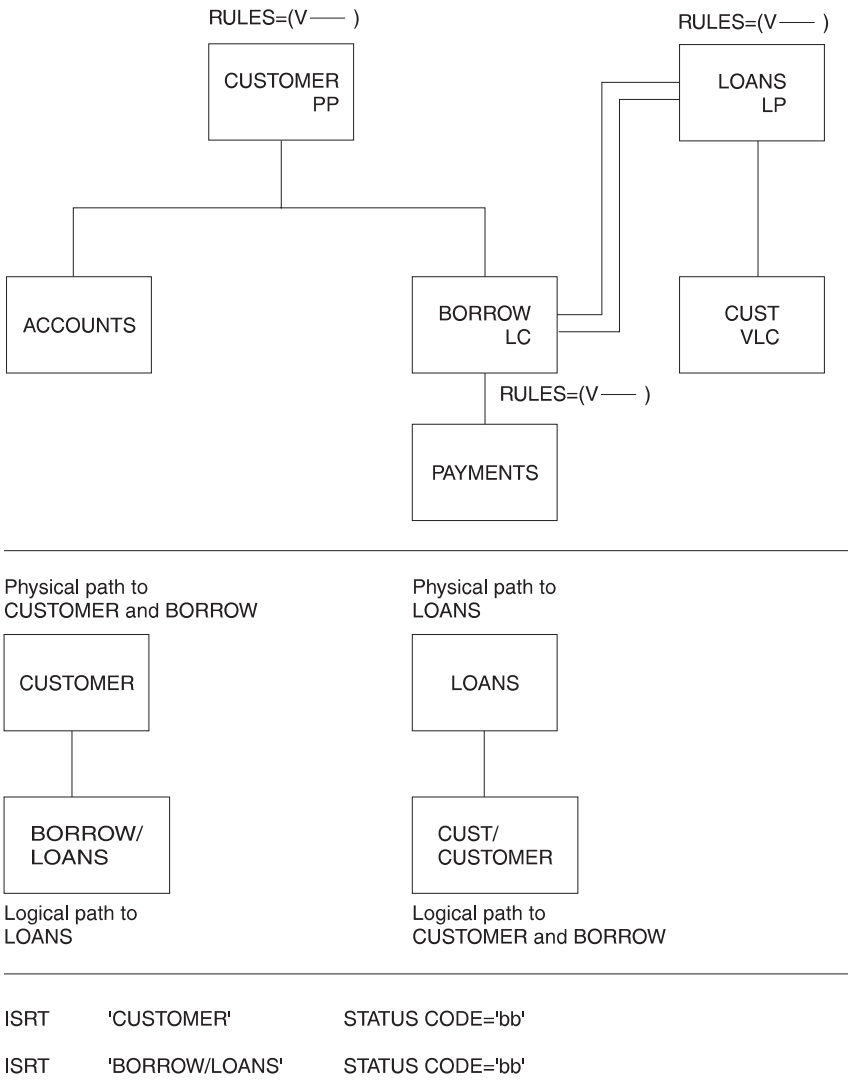


Figure 181. Virtual Insert Rule Example

The code shown in Figure 181 will replace the LOANS segment if present, and insert the LOANS segment if not. The V insert rule is a powerful option.

Insert Rules Summary

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.

Introduction to Delete Rules

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.

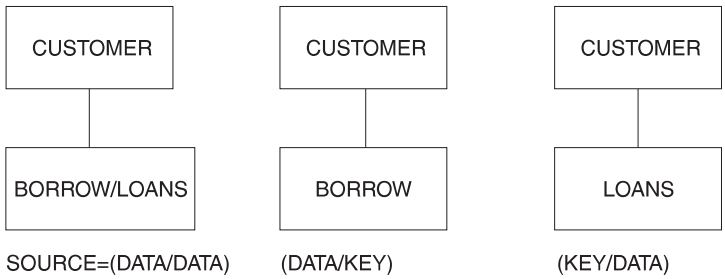
Physical and Logical Deletion

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 Third Access Path

In Figure 182 on page 420, three paths to the logical child segment SEG4 exist:

- 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)

Note: See "Possibility of Abnormal Termination" later in this Appendix for more information on potential abends.

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.

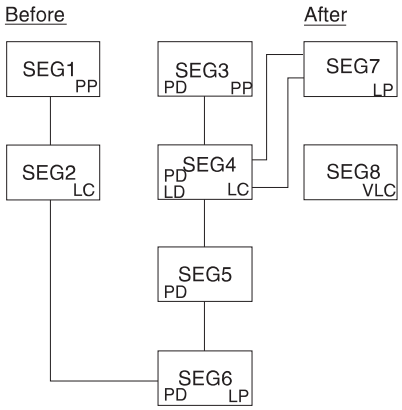


Figure 182. Third Access Path Example

Use of the Delete Byte

The delete byte is used by IMS to maintain the delete status of segments within a database. The meaning of each bit within the delete byte is given in Appendix A. The bit is 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.

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.

Delete Rules

The following is a description of how the delete values work for the logical parent, physical parent, and logical child.

Logical Parent

- 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)

- 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

- 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

Figure 183 through Figure 194 show 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.

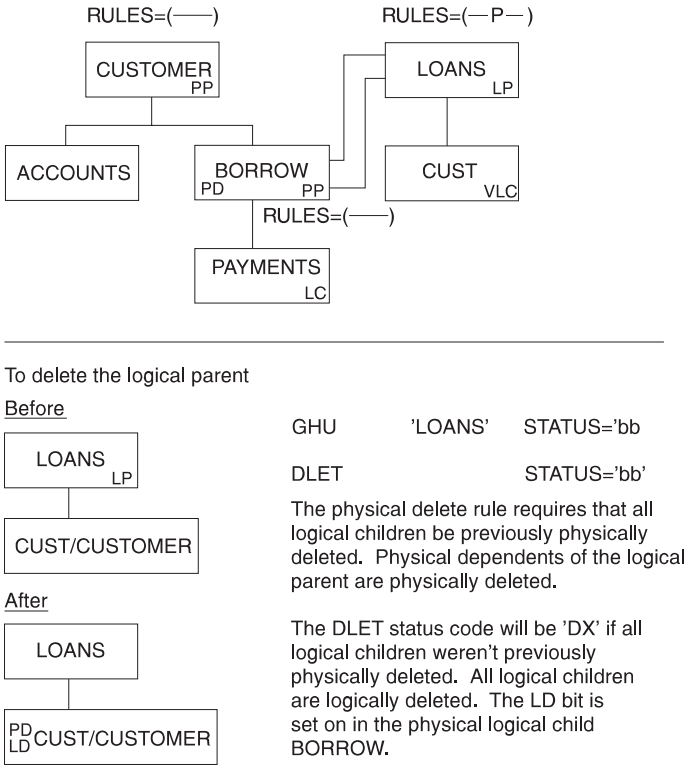


Figure 183. Logical Parent, Virtual Pairing—Physical Delete Rule Example

How to Specify Rules in the Physical DBD

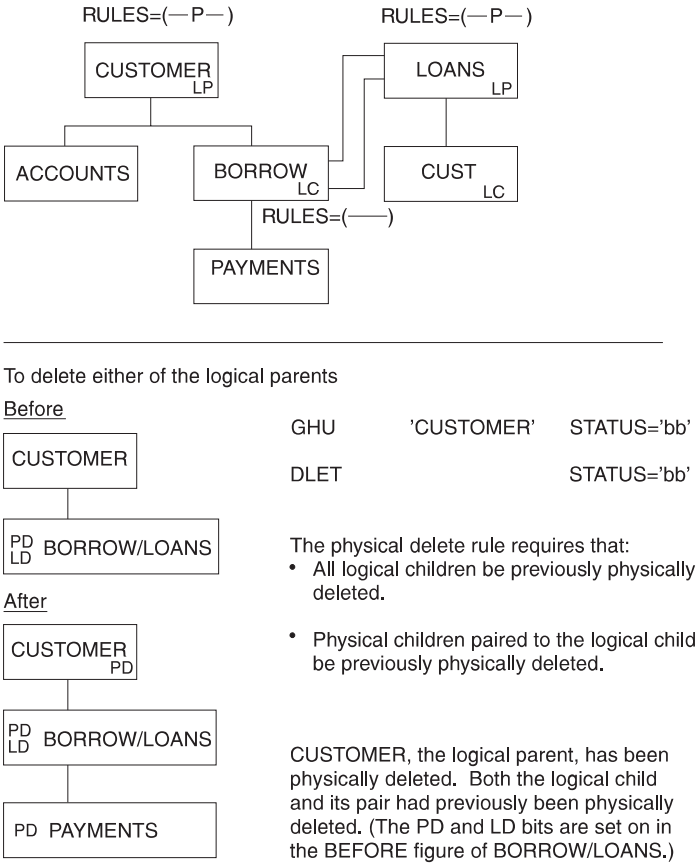


Figure 184. Logical Parent, Physical Pairing—Physical Delete Rule Example

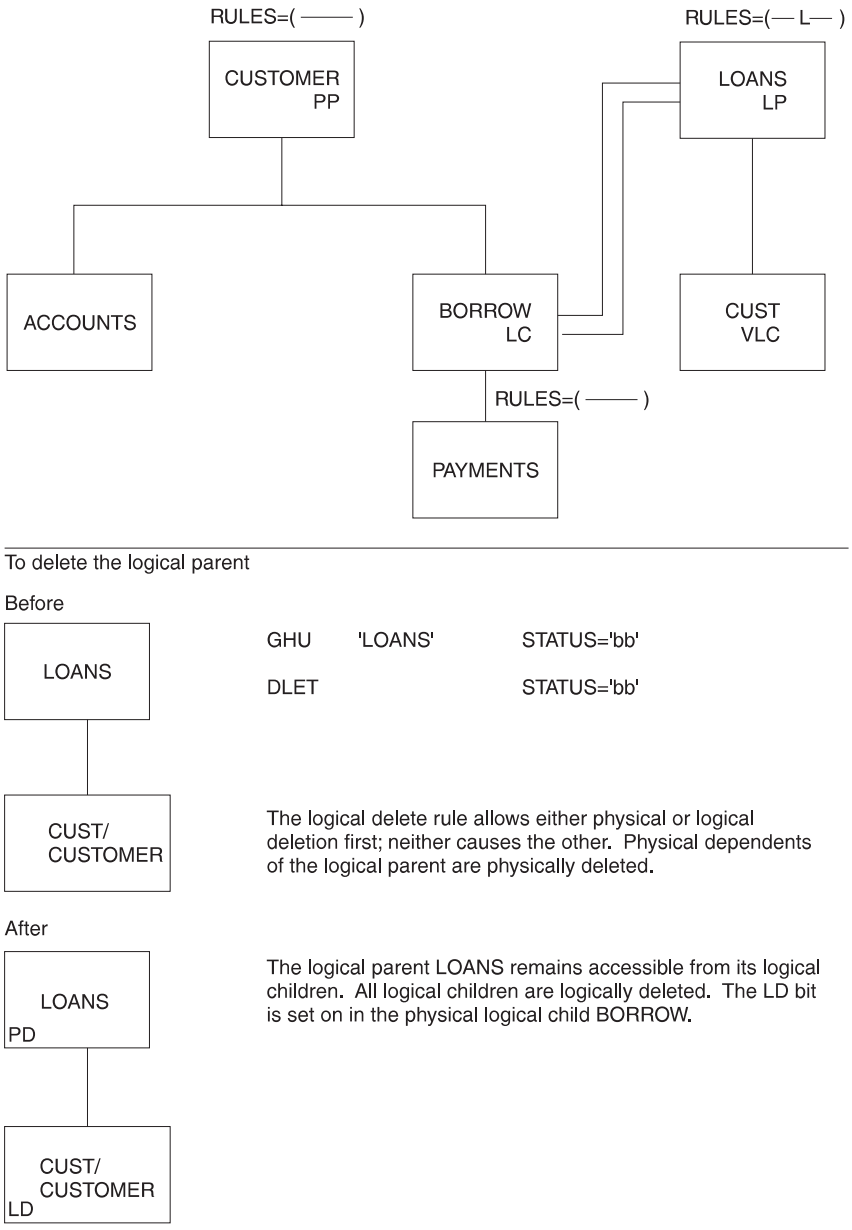


Figure 185. Logical Parent, Virtual Pairing—Logical Delete Rule Example

The processing and results shown in Figure 185 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.

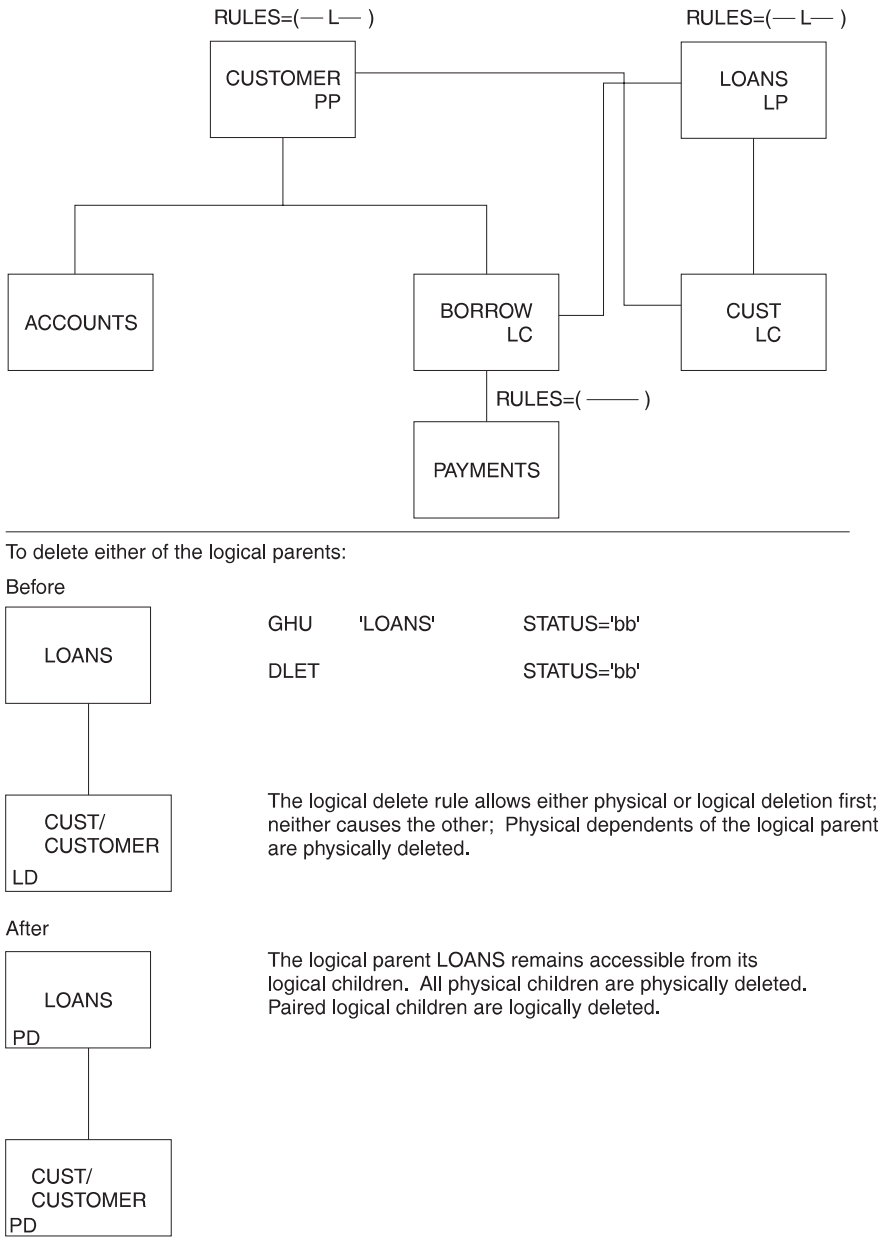


Figure 186. Logical Parent, Physical Pairing—Logical Delete Rule Example

The processing and results shown in Figure 186 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 Figure 187 on page 427.

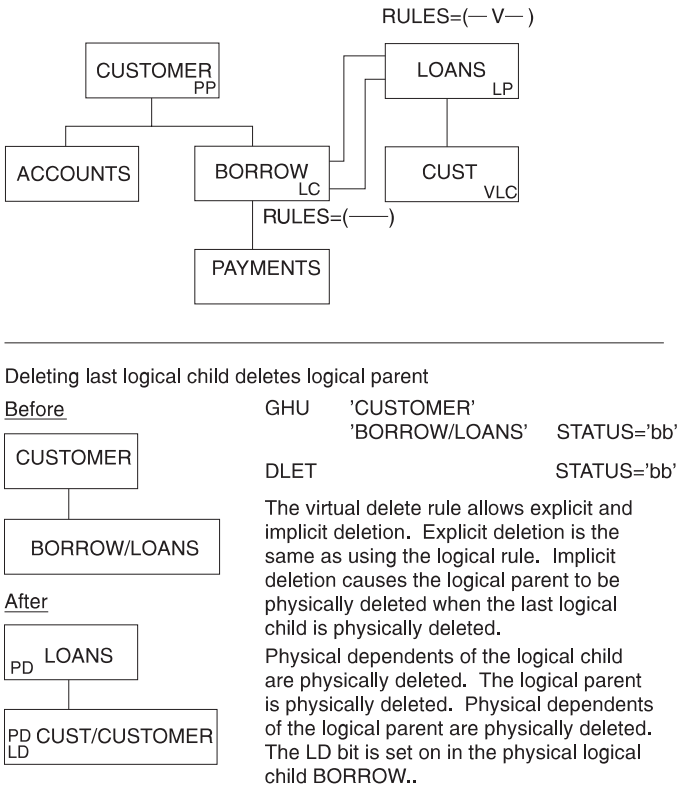


Figure 187. Logical Parent, Virtual Pairing—Virtual Delete Rule Example

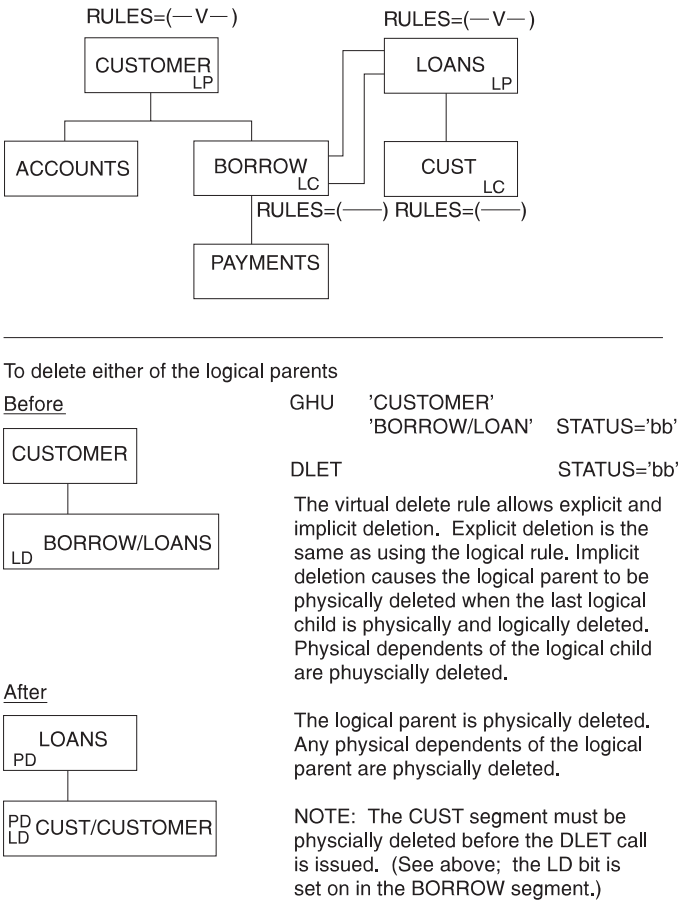


Figure 188. Logical Parent, Physical Pairing—Virtual Delete Rule Example

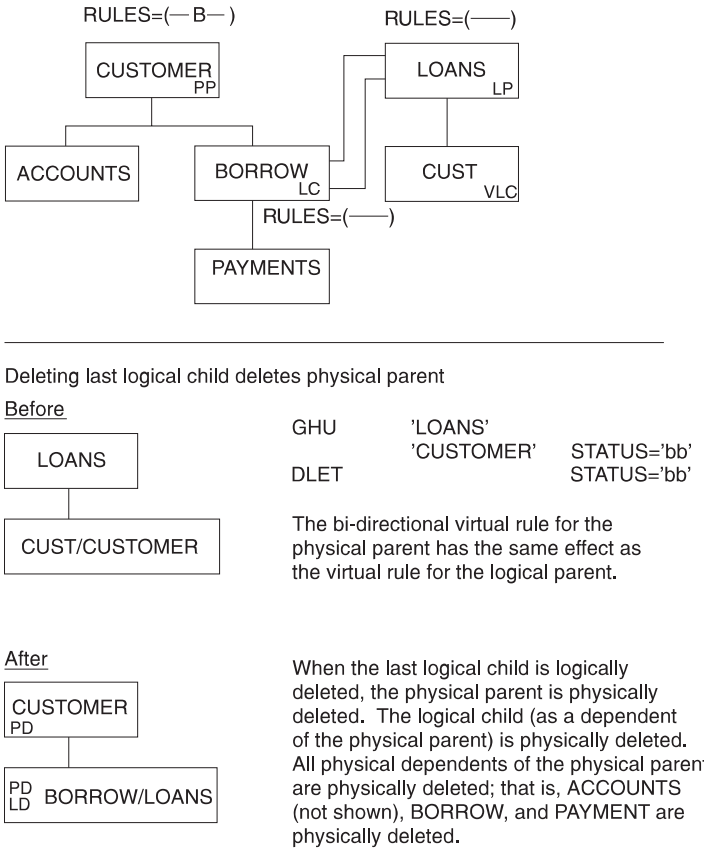


Figure 189. Physical Parent, Virtual Pairing—Bidirectional Virtual Example

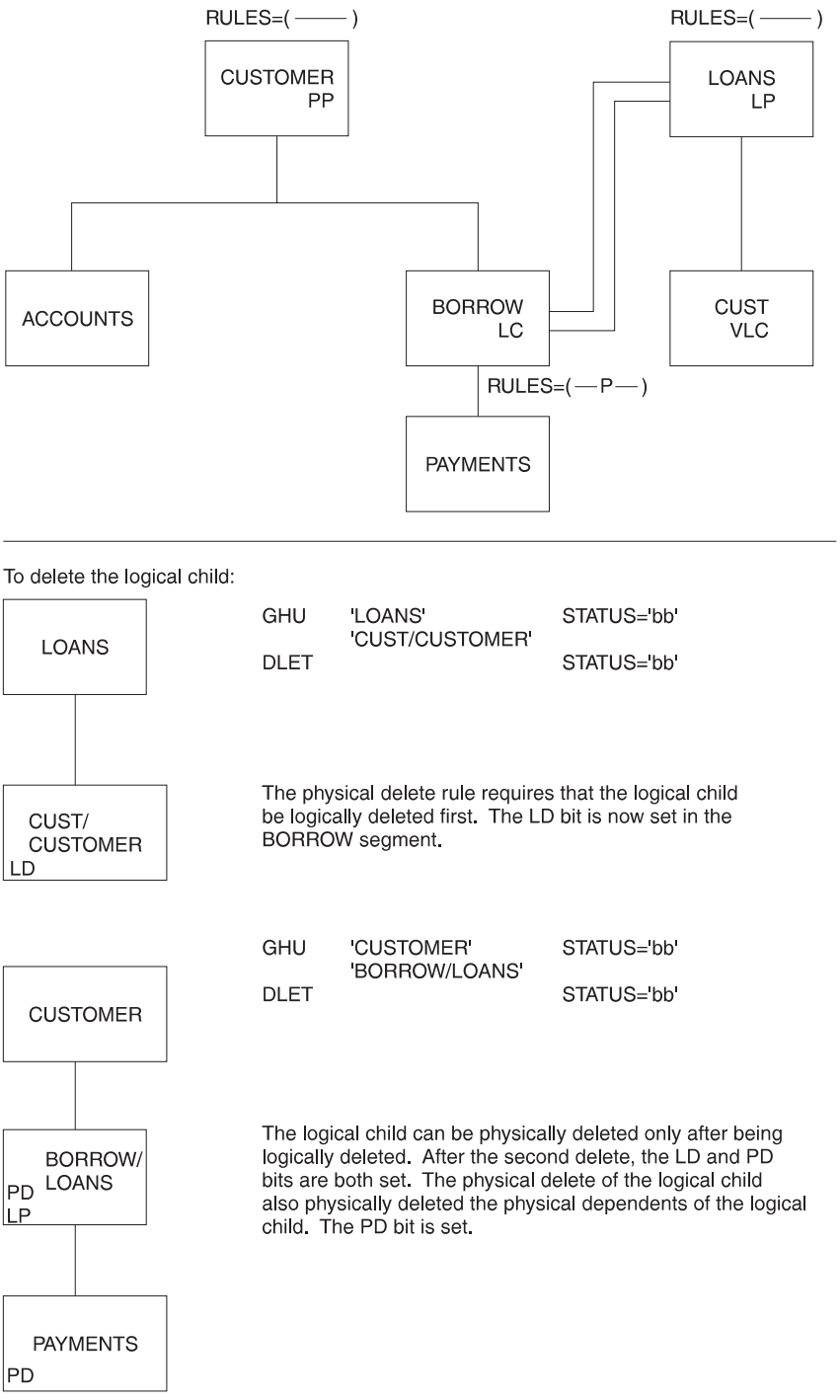


Figure 190. Logical Child, Virtual Pairing—Physical Delete Rule Example

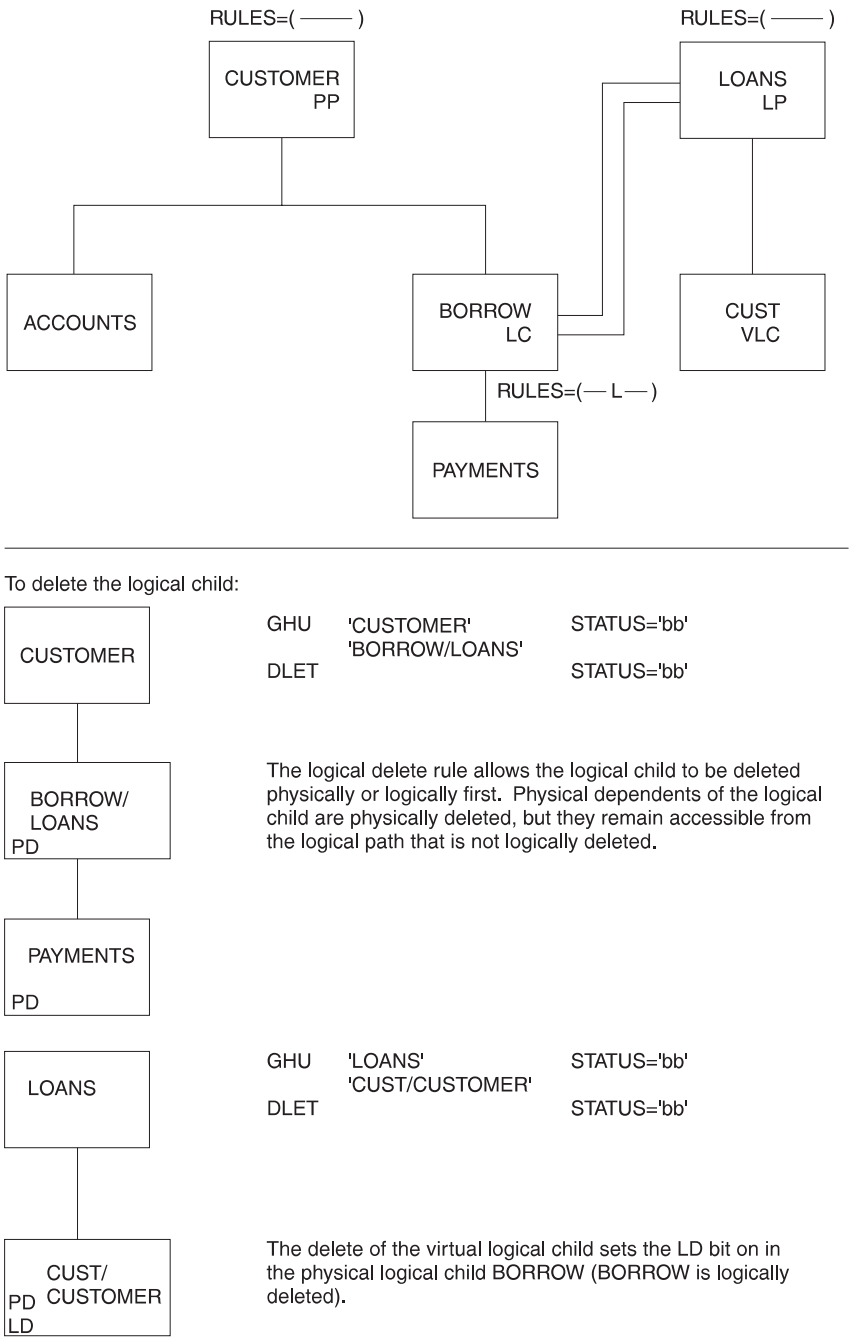


Figure 191. Logical Child, Virtual Pairing—Logical Delete Rule Example

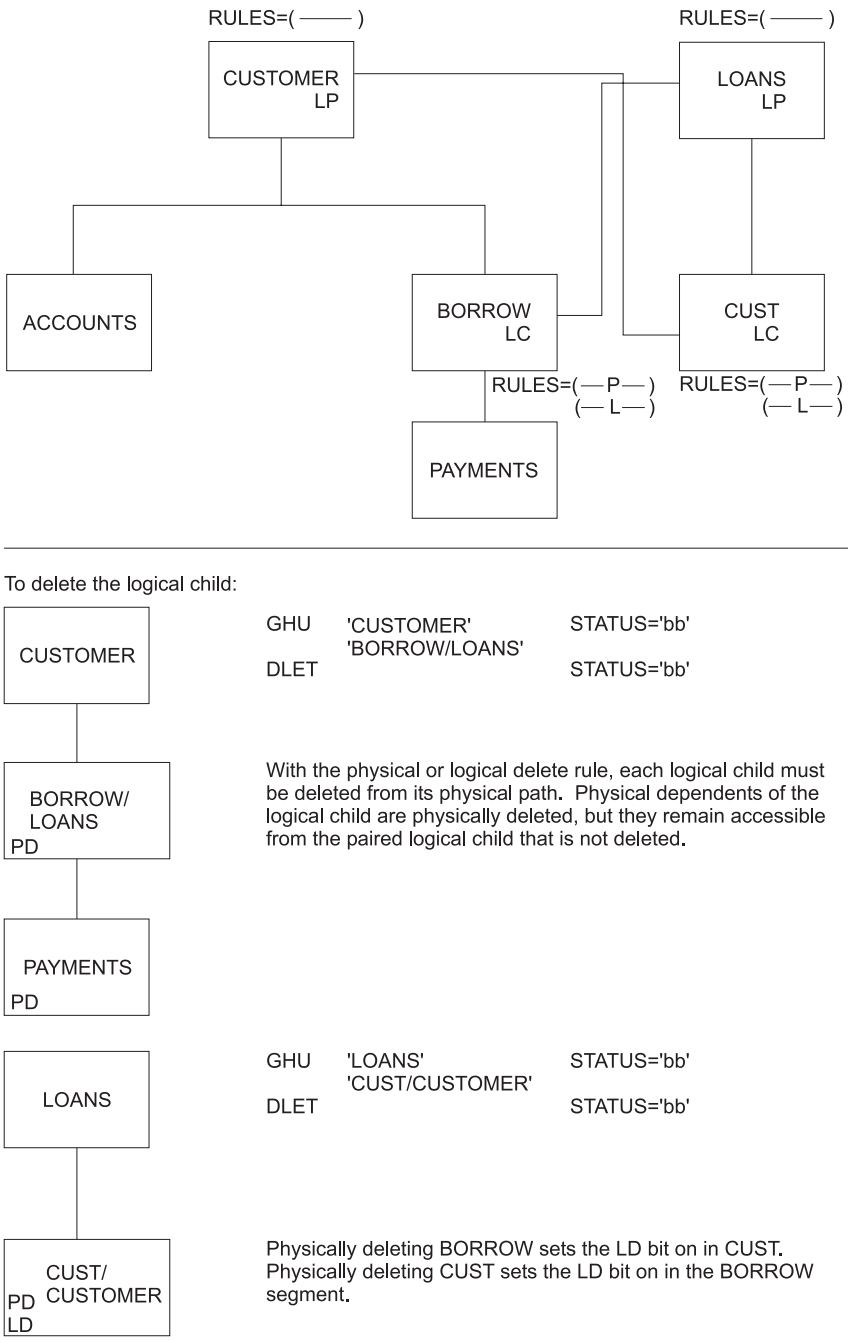


Figure 192. Logical Child, Physical Pairing—Physical or Logical Delete Rule Example

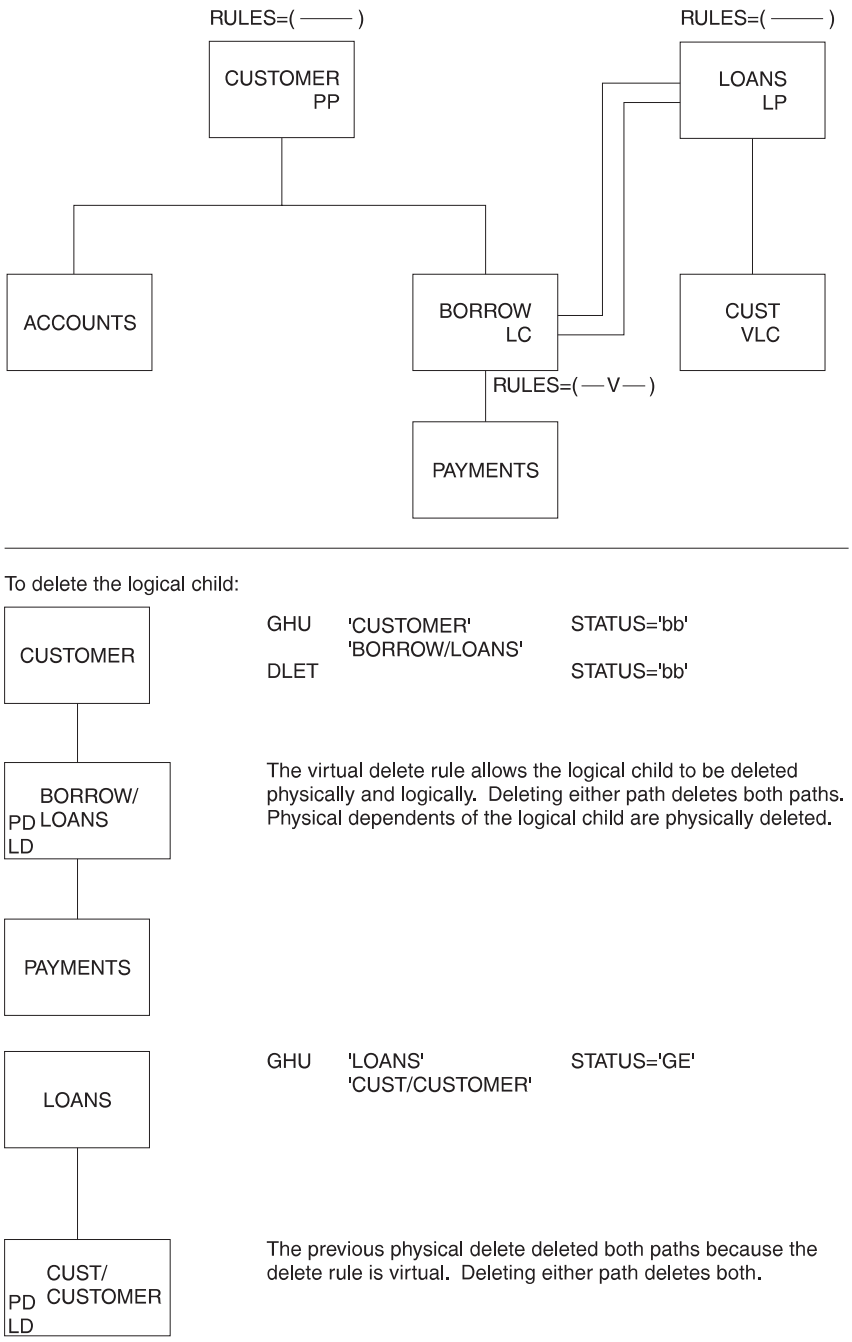


Figure 193. Logical Child, Virtual Pairing—Virtual Delete Rule Example

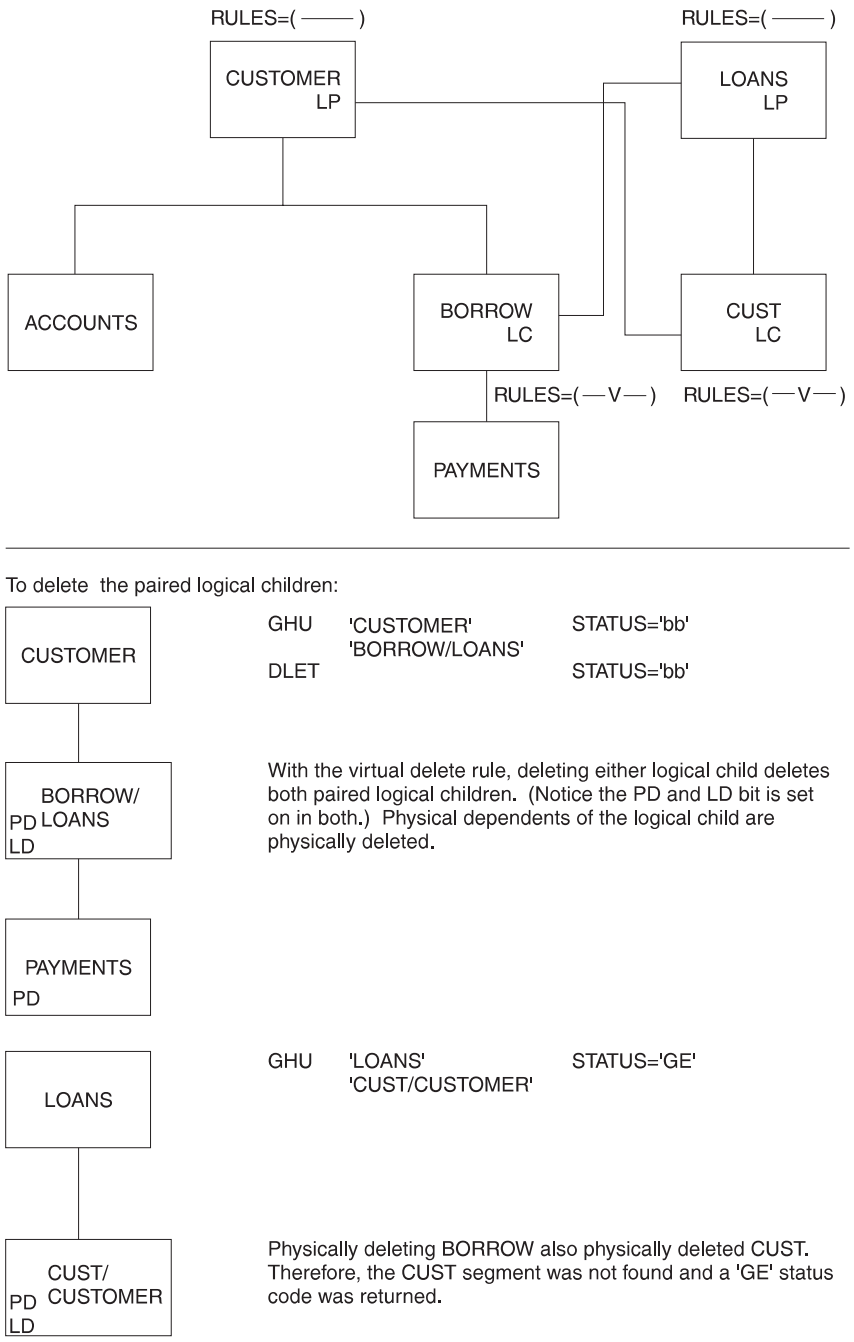


Figure 194. Logical Child, Physical Pairing—Virtual Delete Rule Example

Accessibility of Deleted Segments

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. Figure 195 shows the accessibility of deleted segments:

When the physical dependent of a deleted segment is a logical parent with logical children that aren't physically deleted, the logical parent and its physical parents are accessible from those logical children.

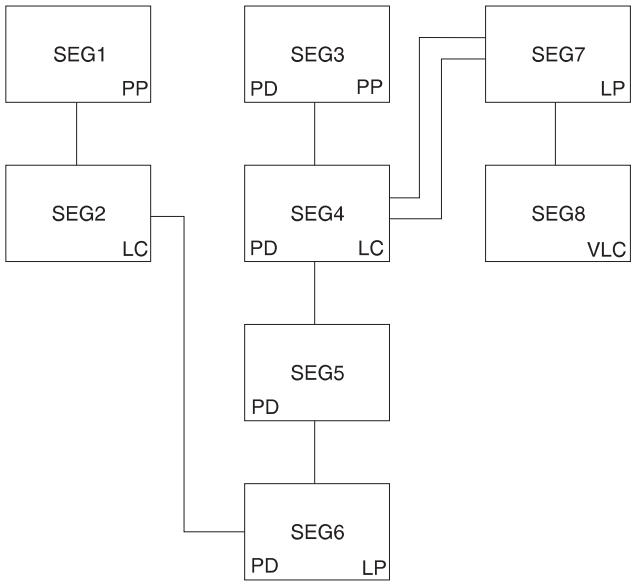


Figure 195. (Part 1 0f 4). Example of Deleted Segments Accessibility

The physical structure above 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.

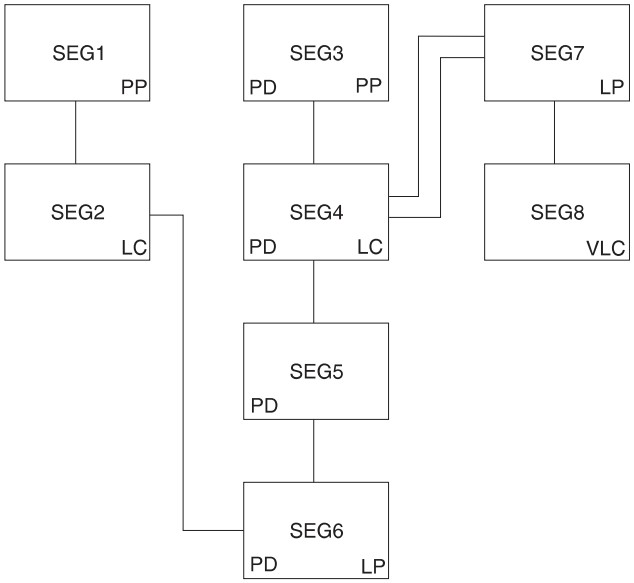


Figure 196. (Part 2 0f 4). Example of Deleted Segments Accessibility

The physical structure above shows that SEG3, SEG4, SEG5, and SEG6 have been physically deleted.

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.

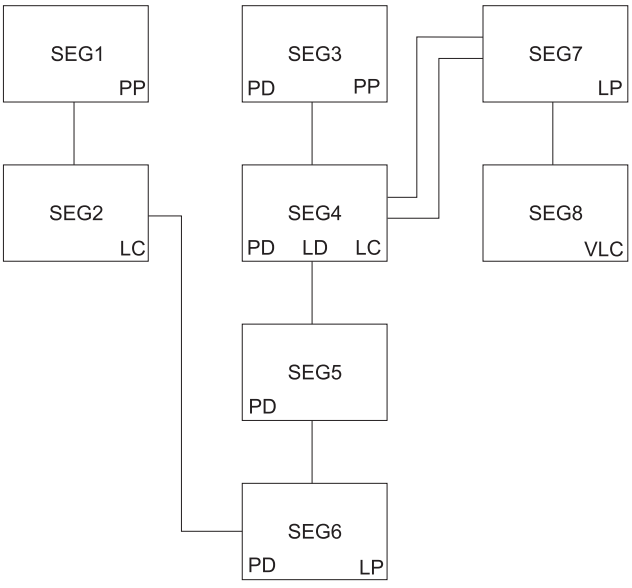


Figure 197. (Part 3 0f 4). Example of Deleted Segments Accessibility

The physical structure above 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 doesn't 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.

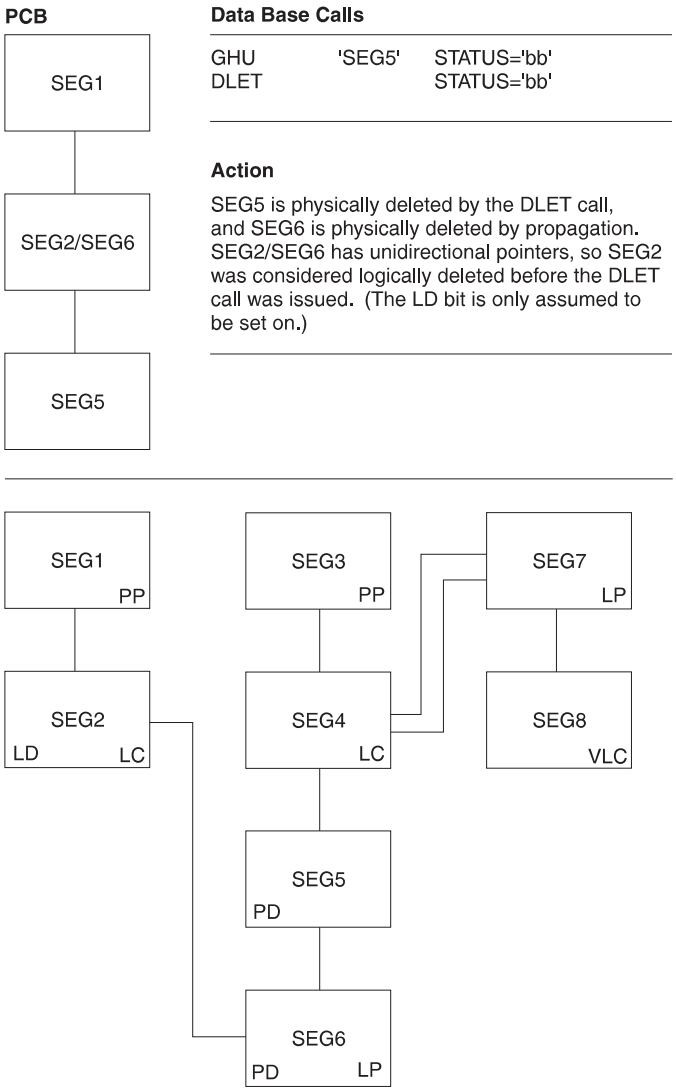


Figure 198. (Part 4 0f 4). Example of Deleted Segments Accessibility

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

Figure 199 shows an example of abnormal termination:

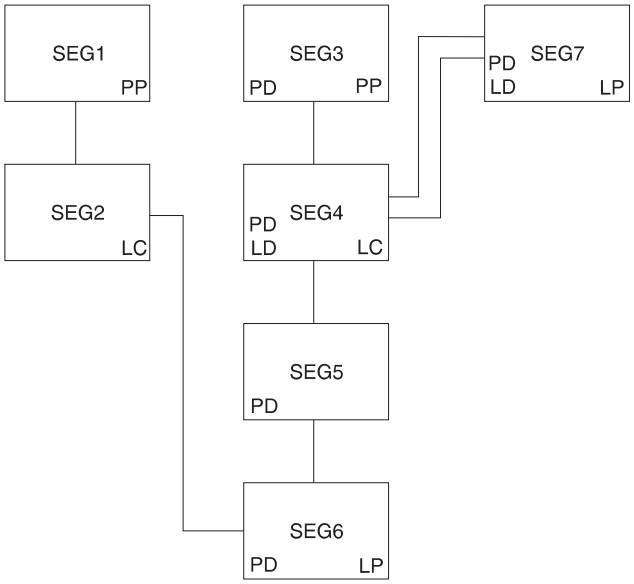


Figure 199. Example of Abnormal Termination

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 doesn't lead to the expected segment.

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. (For more information on this, see the next section, “Detecting Physical Delete Rule Violations”.) 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.

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. Figure 200 shows an example of violating the physical delete rule:

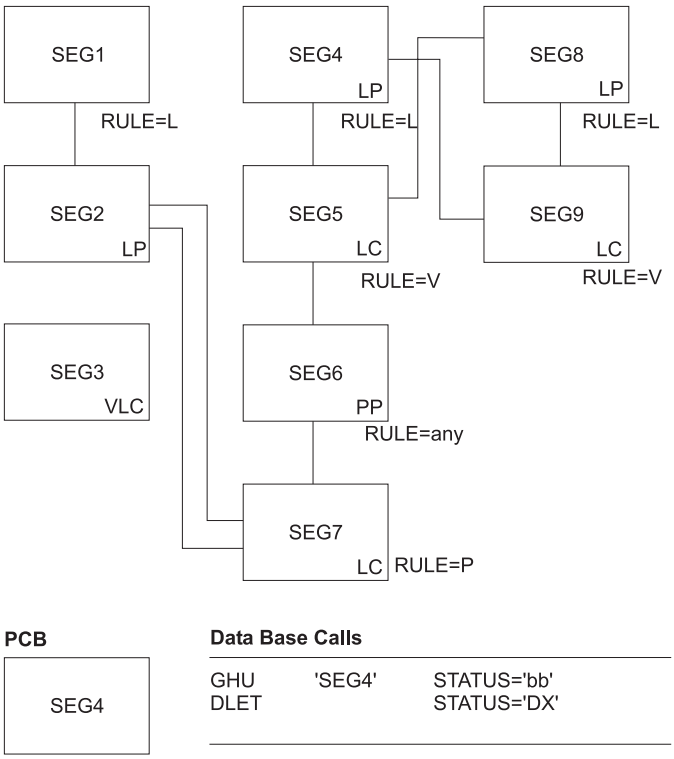


Figure 200. Example of Violation of the Physical Delete Rule

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.

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. Figure 201 shows an example of treating the physical delete rule as logical.

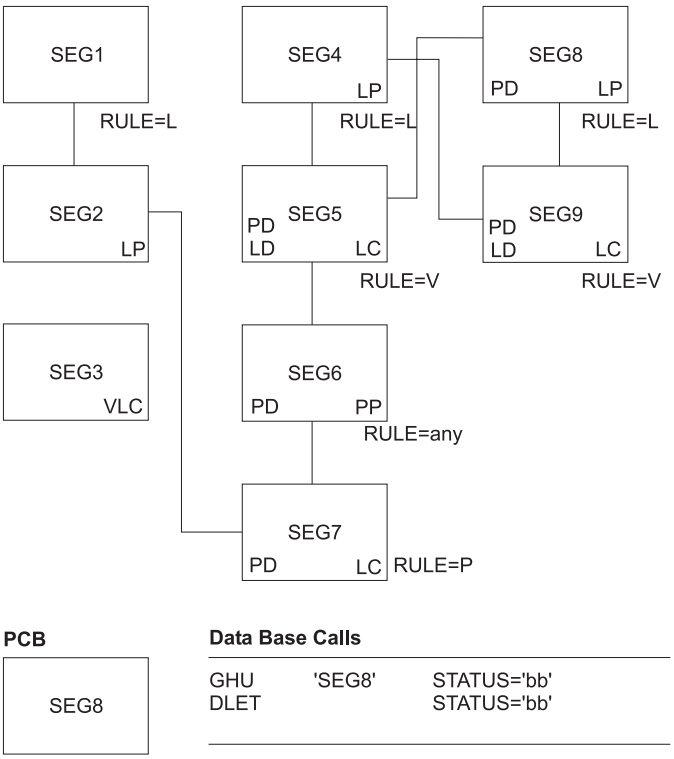


Figure 201. Example of Treating the Physical Delete Rule as Logical

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/or 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:

- 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 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.

Insert, Delete, and Replace Rules Summary

Figure 202 summarizes rules by stating a desired result and then indicating the rule which can be used to obtain that result.

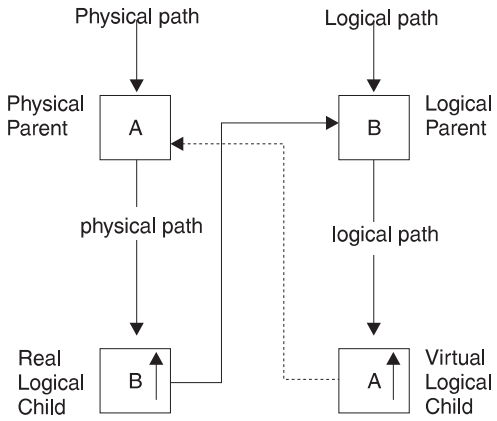


Figure 202. Insert, Delete, and Replace Rules Summary

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 Segment A: The insert rules for segment A control the insert of A using the logical path to A. The rules are as follows:

- To disallow the insert of A on its logical path, use the physical insert rule.
- To allow the insert of A on its logical path (concatenated with virtual segment A↑) use either the logical or virtual rule.

  Where A is already present, a logical connection is established to the existing A segment. The existing A can either be replaced or remain unchanged:

  – If A is to remain unchanged by the insert call, use the logical insert rule.

  – If A is to be replaced by the insert call, use the virtual insert rule.

Delete Rules for Segment A: The delete rules for segment A control the deletion of A using the logical path to A. The rules are as follows:

- To cause segment A to automatically be deleted when the last logical connection (through B↑ to segment A) is broken, use the bidirectional virtual delete rule.
- The other delete rules for A are not meaningful.

Replace Rules for Segment A: The replace rules for segment A control the replacement of A using the logical path to A. The rules are as follows:

- To disallow the replacement of A on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace A, use the physical replace rule.
- To disregard the replacement of A on its logical path, use the logical replace rule.
- To allow the replacement of A on its logical path, use the virtual replace rule.

Insert Rules for Segment B:

Note: These rules are identical to the insert rules for segment A.

The insert rules for segment B control the insert of B using the logical path to B. The rules are as follows:

- To disallow the insert of B on its logical path, use the physical insert rule.
- To allow the insert of B on its logical path (concatenated with virtual segment B↑) use either the logical or virtual rule.

  Where B is already present, a logical connection is established to the existing B segment. The existing B can either be replaced or remain unchanged:

  – If B is to remain unchanged by the insert call, use the logical insert rule.

  – If B is to be replaced by the insert call, use the virtual insert rule.

Delete Rules for Segment B: The delete rules for segment B control the deletion of 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 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 B is made while there are occurrences of B↑ pointing to segment B, a 'DX' status code is returned and no segment is deleted.
- To allow segment B to be deleted on its physical path, choose the logical delete rule. When B is deleted, it is no longer accessible on its physical path. It is still possible to access B from A via B↑ as long as B↑ exists.
- Use the virtual delete rule to physically delete segment B when it has been explicitly deleted by a delete call or implicitly deleted when all B↑s pointing to it have been physically deleted.

Replace Rules for Segment B:

Note: These rules are identical to the replace rules for segment A.

The replace rules for segment B control the replacement of B using the logical path to B. The rules are as follows:

- Use the physical replace rule to disallow the replacement of B on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace B.
- Use the logical replace rule to disregard the replacement of B on its logical path.
- Use the virtual replace rule to allow the replacement of B on its logical path.

Insert Rules for Segment B↑: The insert rules do not apply to a logical child.

Delete Rules for Segment B↑: The delete rules for 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 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 B↑.

Replace Rules for Segment B↑:

Note: These rules are identical to the replace rules for segment A.

The replace rules for segment B control the replacement of B↑ using the logical path to B↑. The rules are as follows:

- Use the physical replace rule to disallow the replacement of B↑ on its logical path and receive an 'RX' status code if the rule is violated by an attempt to replace B↑.
- To disregard an attempt to replace B↑ on its logical path, use the logical replace rule.
- To allow the replacement of B↑ on its logical path, use the virtual replace rule.

Appendix C. Using OSAM as the Access Method

OSAM Information for Database Access

This appendix 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:

- OSAM is a special access method supplied with IMS.
- 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.
- The number of extents in an OSAM data set is limited by:
  - The maximum length of the data extent block (DEB)
  - The length of the sector number table that is created for rotational position sensing (RPS) devices

  The length of a DEB is represented in a single byte that is expressed as the number of doublewords. The sector number table exists only for RPS devices and consists of a fixed area of eight bytes plus one byte for each block on a track, rounded up to an even multiple of eight bytes. A minimum-sized sector table (7 blocks per track) requires two doublewords. A maximum-sized sector table (255 blocks per track) requires 33 doublewords.

  In addition, for each extent area (two doublewords), OSAM requires a similar area that contains device geometry data. Each extent requires a total of four doublewords. The format and length (expressed in doublewords) of an OSAM DEB are as follows:

```
Appendage sector table             5
Basic DEB                          4
Access method dependent section    2
Subroutine name section            1
Standard DEB extents             120 (60 extents)
OSAM extent data                 120
Minimum sector table               2
```

  With a minimum-sized sector table, the DEB can reflect a maximum of 60 DASD extents. With a maximum-sized sector table, the DEB can reflect a maximum of 52 DASD extents.
- 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.
- An OSAM data set with an even length blocksize has an 8 gigabyte size limit. An OSAM data set with an odd length blocksize has a 4 gigabyte size limit.

- 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 the DFDSS Dump Restore Utility (GC26-3949)—DFDSS 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 which have not been pre-allocated, the DS1LSTAR field for each DSCB will be valid and DFDSS 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, multivolume 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 to be not updated. The DSCB for the last volume loaded is updated during close processing of the data set.

  DFDSS physical DUMP/RESTORE with parameters ALLEXCP or ALLDATA must be used when migrating OSAM data sets span pre-allocated, multivolumes. These parameters will allow DFDSS to correctly migrate OSAM data sets.

Other MVS access methods (VSAM and SAM) are used in addition to OSAM for physical storage of data.

For information about defining OSAM subpools, refer to *IMS/ESA Installation Volume 2: System Definition and Tailoring* .

Appendix D. Correcting Bad Pointers

Ordinarily, bad pointers should not occur in your database. When they do, the cause 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.

Bibliography

This bibliography includes all the publications cited in this book, including the publications in the IMS library.

Batch Terminal Simulator Program Reference and Operations Manual, SH20-5523

CICS/ESA Facilities and Planning Guide, SC33-0504

Cross System Product/370 Application Development Guide, SH23-0514

Data Extraction, Processing, and Restructuring System Program Description/Operations Manual, SH20-2177

DataPropagator NonRelational IMS/ESA An Introduction, GH19-5034

DB/DC Data Dictionary General Information Manual, GH20-9104

DB/DC Data Dictionary Terminal User's Guide and Command Reference, SH20-9189

DBT DB Segment Restructure User's Guide, SH20-6582

IMSASAP II Program Description/Operations Manual, SB21-1793

IMS/ESA System Utilities/Database Tools (DBT) General Information Manual, GH20-6579

MVS/DFP Access Method Services for the Integrated Catalog Facility, SC26-4562

MVS/ESA System Programming Library: Initialization and Tuning, GC28-1828

MVS/ESA VSAM Administration Guide, SC26-4518

OS/VS2 Access Method Services, GC26-3841

IMS/ESA Version 6 Library

SC26-8725	ADB	Administration Guide: Database Manager
SC26-8730	AS	Administration Guide: System
SC26-8731	ATM	Administration Guide: Transaction Manager
SC26-8727	APDB	Application Programming: Database Manager
SC26-8728	APDG	Application Programming: Design Guide
SC26-8726	APCICS	Application Programming: EXEC DLI Commands for CICS and IMS
SC26-8729	APTM	Application Programming: Transaction Manager
SC26-8732	CG	Customization Guide
SC26-9517	CQS	Common Queue Server Reference
SC26-8733	DBRC	Database Recovery Control Guide and Reference
LY37-3731	DGR	Diagnosis Guide and Reference
LY37-3732	FAST	Failure Analysis Structure Tables (FAST) for Dump Analysis
GC26-8736	IIV	Installation Volume 1: Installation and Verification
GC26-8737	ISDT	Installation Volume 2: System Definition and Tailoring
SC26-8740	MIG	Master Index and Glossary
GC26-8739	MC	Messages and Codes
SC26-8743	OTMA	Open Transaction Manager Access Guide
SC26-8741	OG	Operations Guide
SC26-8742	OR	Operator's Reference
GC26-8744	RPG	Release Planning Guide
SC26-8767	SOP	Sample Operating Procedures
SC26-8769	URDB	Utilities Reference: Database Manager
SC26-8770	URS	Utilities Reference: System
SC26-8771	URTM	Utilities Reference: Transaction Manager

Supplementary Publications

GC26-8738	LPS	Licensed Program Specifications
SC26-8766	SOC	Summary of Operator Commmands

Online Softcopy Publications

LK3T-2326	CDROM	IMS/ESA Version 6 Softcopy Library
SK2T-0730	CDROM	IBM Online Library: Transaction Processing and Data
SK2T-0710	CDROM	MVS Collection
SK2T-6700	CDROM	OS/390 Collection

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Readers' Comments — We'd Like to Hear from You

IMS/ESA
Administration Guide:
Database Manager

Publication No. SC26-8725-04

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SC26-8725-04

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department BWE/H3
P.O. Box 49023
San Jose, CA
95161-9945

Fold and Tape **Please do not staple** Fold and Tape

SC26-8725-04

Cut or Fold
Along Line

Program Number: 5655-158

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-8725-04

Spine information:

IMS/ESA IMS/ESA V6 Admin Guide: DB Version 6