

Designing objects and relationships

- Logical database design with entity-relationship model 84
- Logical database design with Unified Modeling Language 99
- Physical database design 101
- For more information 106
- Practice exam questions 108
- Answers to practice exam questions 110

Exam topics that this chapter covers

Working with DB2 UDB objects:

- Ability to demonstrate use of DB2 UDB data types
- Knowledge to identify characteristics of a table, view, or index

When you design any sort of database, you need to answer many different questions. The same is true when you are designing a DB2 database. How will you organize your data? How will you create relationships between tables? How should you define the columns in your tables? What kind of table space should you use?

To design a database, you perform two general tasks. The first task is logical data modeling, and the second task is physical data modeling. In logical data modeling, you design a model of the data without paying attention to specific functions and capabilities of the DBMS that will store the data. In fact, you could even build a logical data model without knowing which DBMS you will use. Next comes the task of physical data modeling. This is when you move closer to a physical implementation. The primary purpose of the physical design stage is to optimize performance while ensuring the integrity of the data.


This chapter begins with an introduction to the task of logical data modeling. The logical data modeling section focuses on the entity-relationship model and provides an overview of the Unified Modeling Language (UML). The chapter ends with the task of physical database design.

After completing the logical and physical design of your database, you implement the design. You can read about this task in “Chapter 7. Implementing your database design.”

Logical database design with entity-relationship model

Before you implement a database, you should plan or design it so that it satisfies all requirements. This section introduces the first task of designing a database—logical design.

Modeling your data

Designing and implementing a successful database, one that satisfies the needs of an organization, requires a logical data model. *Logical data modeling* is the process of documenting the comprehensive business information requirements in an accurate and consistent format. Analysts who do data modeling define the data items and the business rules that affect those data items. The process of data modeling acknowledges that business data is a vital asset that the organization needs to understand and carefully manage. This section contains information that was adapted from *Handbook of Relational Database Design*. 

Consider the following business facts that a manufacturing company needs to represent in its data model:

- Customers purchase products.
- Products consist of parts.
- Suppliers manufacture parts.
- Warehouses store parts.
- Transportation vehicles move the parts from suppliers to warehouses and then to manufacturers.

These are all business facts that a manufacturing company's logical data model needs to include. Many people inside and outside the company rely on information that is based on these facts. Many reports include data about these facts.

Any business, not just manufacturing companies, can benefit from the task of data modeling. Database systems that supply information to decision makers, customers, suppliers, and others are more successful if their foundation is a sound data model.

An overview of the data modeling process

You might wonder how people build data models. Data analysts can perform the task of data modeling in a variety of ways. (This process assumes that a data analyst is performing the steps, but some companies assign this task to other people in the organization.) Many data analysts follow these steps:

1. **Build critical user views.**

Analysts begin building a logical data model by carefully examining a single business activity or function. They develop a *user view*, which is the model or representation of critical information that the business activity requires. (In a later stage, the analyst combines each individual user view with all the other user views into a consolidated logical data model.) This initial stage of the data modeling process is highly interactive. Because data analysts cannot fully

understand all areas of the business that they are modeling, they work closely with the actual users. Working together, analysts and users define the major entities (significant objects of interest) and determine the general relationships between these entities.

2. Add keys to user views.


Next, analysts add key detailed information items and the most important business rules. Key business rules affect insert, update, and delete operations on the data.

Example: A business rule might require that each customer entity have at least one unique identifier. Any attempt to insert or update a customer identifier that matches another customer identifier is not valid. In a data model, a unique identifier is called a primary key, which you read about in “Primary keys” on page 50.

3. Add detail to user views and validate them.

After the analysts work with users to define the key entities and relationships, they add other descriptive details that are less vital. They also associate these descriptive details, called *attributes*, to the entities.

Example: A customer entity probably has an associated phone number. The phone number is a nonkey attribute of the customer entity.

Analysts also validate all the user views that they have developed. To validate the views, analysts use the normalization process (which you can read about later in this chapter) and process models. *Process models* document the details of how the business will use the data. You can read more about process models and data models in other books on those subjects. 

4. Determine additional business rules that affect attributes.

Next, analysts clarify the data-driven business rules. *Data-driven business rules* are constraints on particular data values, which you read about in “Referential integrity and referential constraints” on page 54. These constraints need to be true, regardless of any particular processing requirements. Analysts define these constraints during the data design stage, rather than during application design. The advantage to defining data-driven business rules is that programmers of many applications don’t need to write code to enforce these business rules.

Example: Assume that a business rule requires that a customer entity have a phone number, an address, or both. If this rule doesn’t apply to the data itself, programmers must develop, test, and maintain applications that verify the existence of one of these attributes.

Data-driven business requirements have a direct relationship with the data, thereby relieving programmers from extra work.

5. Integrate user views.

In this last phase of data modeling, analysts combine into a consolidated logical data model the different user views that they have built. If other data models already exist in the organization, the analysts integrate the new data model with the existing ones. At this stage, analysts also strive to make their data model flexible so that it can support the current business environment and possible future changes.

Example: Assume that a retail company operates in a single country and that business plans include expansion to other countries. Armed with knowledge of these plans, analysts can build the model so that it is flexible enough to support expansion into other countries.



Recommendations for logical data modeling

To build sound data models, analysts follow a well-planned methodology, which includes these tasks:

- Work interactively with the users as much as possible.
- Use diagrams to represent as much of the logical data model as possible.
- Build a *data dictionary* to supplement the logical data model diagrams. (A data dictionary is a repository of information about an organization's application programs, databases, logical data models, users, and authorizations. A data dictionary can be manual or automated.)

Data modeling: Some practical examples

“An overview of the data modeling process” on page 85 summarizes the key activities in data modeling. This section shows how you might perform these activities in real life.

You begin by defining your entities, the significant objects of interest. Entities are the things about which you want to store information. For example, you might want to define an entity, called **EMPLOYEE**, for employees because you need to store information about everyone who works for your organization. You might also define an entity, called **DEPARTMENT**, for departments.

Next, you define primary keys for your entities. A primary key is a unique identifier for an entity. In the case of the **EMPLOYEE** entity, you probably need to store lots of information. However, most of this information (such as gender, birth date, address, and hire date) would not be a good choice for the primary key. In this case, you could choose a unique employee ID or number (**EMPLOYEE_NUMBER**) as

the primary key. In the case of the DEPARTMENT entity, you could use a unique department number (DEPARTMENT_NUMBER) as the primary key.

After you have decided on the entities and their primary keys, you can define the relationships that exist between the entities. The relationships are based on the primary keys. If you have an entity for EMPLOYEE and another entity for DEPARTMENT, the relationship that exists is that employees are assigned to departments. You can read more about this topic in the next section.

After defining the entities, their primary keys, and their relationships, you can define additional attributes for the entities. In the case of the EMPLOYEE entity, you might define the following additional attributes:

- Birth date
- Hire date
- Home address
- Office phone number
- Gender
- Resume

You can read more about defining attributes later in this chapter.

Finally, you normalize the data, a task that is outlined in “Normalizing your entities to avoid redundancy” on page 94.

Defining entities for different types of relationships

In a relational database, you can express several types of relationships. Consider the possible relationships between employees and departments. If a given employee can work in only one department, this relationship is *one-to-one* for employees. One department usually has many employees; this relationship is *one-to-many* for departments. Relationships can be one-to-many, many-to-one, one-to-one, or many-to-many.

The type of a given relationship can vary, depending on the specific environment. If employees of a company belong to several departments, the relationship between employees and departments is many-to-many.

You need to define separate entities for different types of relationships. When modeling relationships, you can use diagram conventions to depict relationships by using different styles of lines to connect the entities.

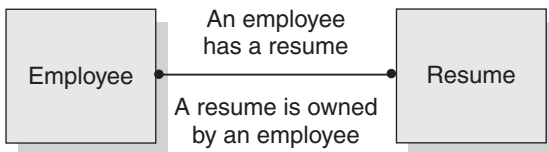


Figure 4.1
Assigning one-to-one facts to an entity

One-to-one relationships

When you are doing logical database design, *one-to-one* relationships are bidirectional relationships, which means that they are single-valued in both directions. For example, an employee has a single resume; each resume belongs to only one person. Figure 4.1 illustrates that a one-to-one relationship exists between the two entities. In this case, the relationship reflects the rules that an employee can have only one resume and that a resume can belong to only one employee.

One-to-many and many-to-one relationships

A *one-to-many relationship* occurs when one entity has a multivalued relationship with another entity. In Figure 4.2, you see that a one-to-many relationship exists between the two entities—employee and department. This figure reinforces the business rules that a department can have many employees, but that each individual employee can work for only one department.

Many-to-many relationships

A *many-to-many relationship* is a relationship that is multivalued in both directions. Figure 4.3 illustrates this kind of relationship. An employee can work on more than one project, and a project can have more than one employee assigned. If you look

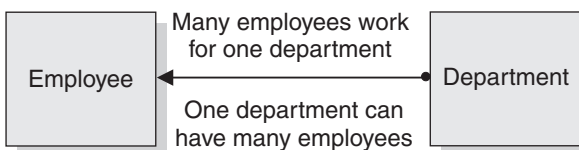
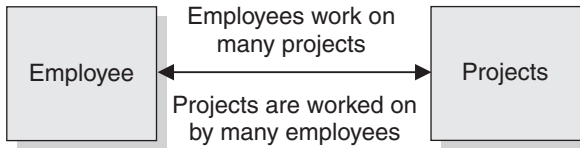


Figure 4.2
Assigning many-to-one facts to an entity

**Figure 4.3**

Assigning many-to-many facts to an entity

at this book’s example tables (in “Appendix A. Example tables in this book”), you can find answers for the following questions:

- What does Wing Lee work on?
- Who works on project number OP2012?

Both questions yield multiple answers. Wing Lee works on project numbers OP2011 and OP2012. The employees who work on project number OP2012 are Ramlal Mehta and Wing Lee.

Applying business rules to relationships

Whether a given relationship is one-to-one, one-to-many, many-to-one, or many-to-many, your relationships need to make good business sense. Therefore, database designers and data analysts can be more effective when they have a good understanding of the business. If they understand the data, the applications, and the business rules, they can succeed in building a sound database design.

When you define relationships, you have a big influence on how smoothly your business runs. If you don’t do a good job at this task, your database and associated applications are likely to have many problems, some of which may not manifest themselves for years.

Defining attributes for the entities

When you define attributes for the entities, you generally work with the data administrator (DA) to decide on names, data types, and appropriate values for the attributes.

Naming attributes

Most organizations have naming conventions. In addition to following these conventions, DAs also base attribute definitions on class words. A *class word* is a single word that indicates the nature of the data that the attribute represents.

Example: The class word NUMBER indicates an attribute that identifies the number of an entity. Attribute names that identify the numbers of entities should there-

fore include the class word of NUMBER. Some examples are EMPLOYEE_NUMBER, PROJECT_NUMBER, and DEPARTMENT_NUMBER.

When an organization does not have well-defined guidelines for attribute names, the DAs try to determine how the database designers have historically named attributes. Problems occur when multiple individuals are inventing their own naming schemes without consulting one another.

Choosing data types for attributes

In addition to choosing a name for each attribute, you must specify a data type. Most organizations have well-defined guidelines for using the different data types. Here is an overview of the main data types that you can use for the attributes of your entities.

String

Data that contains a combination of letters, numbers, and special characters. Some of the string data types are listed below:

- CHARACTER: Fixed-length character strings. The common short name for this data type is CHAR.
- VARCHAR: Varying-length character strings.
- CLOB: Varying-length character strings, typically used when a character string might exceed the limits of the VARCHAR data type.
- GRAPHIC: Fixed-length graphic strings that contain double-byte characters.
- VARGRAPHIC: Varying-length graphic strings that contain double-byte characters.
- DBCLOB: Varying-length strings of double-byte characters.
- BLOB: Varying-length binary strings.

Numeric

Data that contains digits. The numeric data types are listed below:

- SMALLINT: for small integers.
- INTEGER: for large integers.
- DECIMAL(p,s) or NUMERIC(p,s), where p is precision and s is scale: for packed decimal numbers with precision p and scale s . *Precision* is the total number of digits and *scale* is the number of digits to the right of the decimal point.
- REAL, for single-precision floating-point numbers.
- DOUBLE, for double-precision floating-point numbers.

Datetime

Data values that represent dates, times, or timestamps. The datetime data types are listed below:

- **DATE:** Dates with a three-part value that represents a year, month, and day.
- **TIME:** Times with a three-part value that represents a time of day in hours, minutes, and seconds.
- **TIMESTAMP:** Timestamps with a seven-part value that represents a date and time by year, month, day, hour, minute, second, and microsecond.

Examples: You might use the following data types for attributes of the EMPLOYEE entity:

- **EMPLOYEE_NUMBER:** CHAR(6)
- **EMPLOYEE_LAST_NAME:** VARCHAR(15)
- **EMPLOYEE_HIRE_DATE:** DATE
- **EMPLOYEE_SALARY_AMOUNT:** DECIMAL(9,2)

The data types that you choose are business definitions of the data type. During physical database design you might need to change data type definitions or use a subset of these data types. The database or the host language might not support all of these definitions, or you might make a different choice for performance reasons.

For example, you might need to represent monetary amounts, but DB2 and many host languages do not have a data type MONEY. In the United States, a natural choice for the SQL data type in this situation is DECIMAL(10,2) to represent dollars. But you might also consider the INTEGER data type for fast, efficient performance.

“Determining column attributes” on page 223 provides additional details about selecting data types when you define columns.

Deciding what values are appropriate for attributes

When you design a database, you need to decide what values are acceptable for the various attributes of an entity. For example, you would not want to allow numeric data in an attribute for a person’s name. The data types that you choose limit the values that apply to a given attribute, but you can also use other mechanisms. These other mechanisms are domains, null values, and default values.

Domain

A *domain* describes the conditions that an attribute value must meet to be a valid value. Sometimes the domain identifies a range of valid values. By defining the

domain for a particular attribute, you apply business rules to ensure that the data will make sense.

Examples:

- A domain might state that a phone number attribute must be a 10-digit value that contains only numbers. You would not want the phone number to be incomplete, nor would you want it to contain alphabetic or special characters and thereby be invalid. You could choose to use either a numeric data type or a character data type. However, the domain states the business rule that the value must be a 10-digit value that consists of numbers.
- A domain might state that a month attribute must be a 2-digit value from 01 to 12. Again, you could choose to use datetime, character, or numeric data types for this value, but the domain demands that the value must be in the range of 01 through 12. In this case, incorporating the month into a datetime data type is probably the best choice. This decision should be reviewed again during physical database design.

Null values

When you are designing attributes for your entities, you will sometimes find that an attribute does not have a value for every instance of the entity. For example, you might want an attribute for a person's middle name, but you can't require a value because some people have no middle name. For these occasions, you can define the attribute so that it can contain null values.

A *null value* is a special indicator that represents the absence of a value. The value can be absent because it is unknown, not yet supplied, or nonexistent. The DBMS treats the null value as an actual value, not as a zero value, a blank, or an empty string.

Just as some attributes should be allowed to contain null values, other attributes should not contain null values.

Example: For the EMPLOYEE entity, you might not want to allow the attribute EMPLOYEE_LAST_NAME to contain a null value.

You can read more about null values in "Chapter 7. Implementing your database design."

Default values

In some cases, you may not want a given attribute to contain a null value, but you don't want to require that the user or program always provide a value. In this case, a default value might be appropriate.

A *default value* is a value that applies to an attribute if no other valid value is available.

Example: Assume that you don't want the EMPLOYEE_HIRE_DATE attribute to contain null values and that you don't want to require users to provide this data. If data about new employees is generally added to the database on the employee's first day of employment, you could define a default value of the current date.

You can read more about default values in "Chapter 7. Implementing your database design."

Normalizing your entities to avoid redundancy

After you define entities and decide on attributes for the entities, you normalize entities to avoid redundancy. An entity is normalized if it meets a set of constraints for a particular normal form, which this section describes. *Normalization* helps you avoid redundancies and inconsistencies in your data. This section summarizes rules for first, second, third, and fourth normal forms of entities, and it describes reasons why you should or shouldn't follow these rules.

The rules for normal form are cumulative. In other words, for an entity to satisfy the rules of second normal form, it also must satisfy the rules of first normal form. An entity that satisfies the rules of fourth normal form also satisfies the rules of first, second, and third normal form.

In this section, you will see many references to the word instance. In the context of logical data modeling, an *instance* is one particular occurrence. An instance of an entity is a set of data values for all of the attributes that correspond to that entity.

Example: Figure 4.4 shows one instance of the EMPLOYEE entity.

EMPLOYEE

EMPLOYEE _NUMBER	EMPLOYEE _FIRST _NAME	EMPLOYEE _LAST _NAME	DEPARTMENT _NUMBER	EMPLOYEE _HIRE _DATE	JOB _NAME	EDUCATION _LEVEL	EMPLOYEE _YEARLY _SALARY _AMOUNT	COMMISSION _AMOUNT
000010	CHRISTINE	HAAS	A00	1975-01-01	PRES	18	52750.00	4220.00

Figure 4.4

One instance of an entity

First normal form

A relational entity satisfies the requirement of first normal form if every instance of an entity contains only one value, never multiple repeating attributes. Repeating attributes, often called a *repeating group*, are different attributes that are inherently the same. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name.

Example: Assume that an entity contains the following attributes:

EMPLOYEE_NUMBER
 JANUARY_SALARY_AMOUNT
 FEBRUARY_SALARY_AMOUNT
 MARCH_SALARY_AMOUNT

This situation violates the requirement of first normal form, because JANUARY_SALARY_AMOUNT, FEBRUARY_SALARY_AMOUNT, and MARCH_SALARY_AMOUNT are essentially the same attribute, EMPLOYEE_MONTHLY_SALARY_AMOUNT.

Second normal form

An entity is in second normal form if each attribute that is not in the primary key provides a fact that depends on the entire key. (For a quick refresher on keys, see “Keys” on page 49.)

A violation of the second normal form occurs when a nonprimary key attribute is a fact about a subset of a composite key.

Example: An inventory entity records quantities of specific parts that are stored at particular warehouses. Figure 4.5 shows the attributes of the inventory entity.

Here, the primary key consists of the PART and the WAREHOUSE attributes together. Because the attribute WAREHOUSE_ADDRESS depends only on the value of WAREHOUSE, the entity violates the rule for second normal form. This design causes several problems:

- Each instance for a part that this warehouse stores repeats the address of the warehouse.



Figure 4.5
 A primary key that violates second normal form



Figure 4.6
Two entities that satisfy second normal form

- If the address of the warehouse changes, every instance referring to a part that is stored in that warehouse must be updated.
- Because of the redundancy, the data might become inconsistent. Different instances could show different addresses for the same warehouse.
- If at any time the warehouse has no stored parts, the address of the warehouse might not exist in any instances in the entity.

To satisfy second normal form, the information in Figure 4.5 would be in two entities, as Figure 4.6 shows.

Third normal form

An entity is in third normal form if each nonprimary key attribute provides a fact that is independent of other nonkey attributes and depends only on the key.

Employee_Department table before update

Key

EMPLOYEE _NUMBER	EMPLOYEE _FIRST _NAME	EMPLOYEE _LAST _NAME	DEPARTMENT _NUMBER	DEPARTMENT _NAME
000200	DAVID	BROWN	D11	MANUFACTURING SYSTEMS
000320	RAMAL	MEHTA	E21	SOFTWARE SUPPORT
000220	JENNIFER	LUTZ	D11	MANUFACTURING SYSTEMS

Employee_Department table after update

Key

EMPLOYEE _NUMBER	EMPLOYEE _FIRST _NAME	EMPLOYEE _LAST _NAME	DEPARTMENT _NUMBER	DEPARTMENT _NAME
000200	DAVID	BROWN	D11	INSTALLATION MGMT
000320	RAMAL	MEHTA	E21	SOFTWARE SUPPORT
000220	JENNIFER	LUTZ	D11	MANUFACTURING SYSTEMS

Figure 4.7
The update of an unnormalized entity. Information in the entity has become inconsistent.

A violation of the third normal form occurs when a nonprimary attribute is a fact about another nonkey attribute.

Example: The first entity in Figure 4.7 contains the attributes `EMPLOYEE_NUMBER` and `DEPARTMENT_NUMBER`. Suppose that a program or user adds an attribute, `DEPARTMENT_NAME`, to the entity. The new attribute depends on `DEPARTMENT_NUMBER`, whereas the primary key is on the `EMPLOYEE_NUMBER` attribute. The entity now violates third normal form.

Changing the `DEPARTMENT_NAME` value based on the update of a single employee, David Brown, does not change the `DEPARTMENT_NAME` value for other employees in that department. The updated version of the entity in Figure 4.7 illustrates the resulting inconsistency. Additionally, updating the `DEPARTMENT_NAME` in this table does not update it in any other table that might contain a `DEPARTMENT_NAME` column.

You can normalize the entity by modifying the `EMPLOYEE_DEPARTMENT` entity and creating two new entities: `EMPLOYEE` and `DEPARTMENT`. Figure 4.8 shows the new entities. The `DEPARTMENT` entity contains attributes for

Employee table

Key

<code>EMPLOYEE_NUMBER</code>	<code>EMPLOYEE_FIRST_NAME</code>	<code>EMPLOYEE_LAST_NAME</code>
000200	DAVID	BROWN
000329	RAMLAL	MEHTA
000220	JENNIFER	LUTZ

Department table

Key

<code>DEPARTMENT_NUMBER</code>	<code>DEPARTMENT_NAME</code>
D11	MANUFACTURING SYSTEMS
E21	SOFTWARE SUPPORT

Employee_Department table

Key

<code>DEPARTMENT_NUMBER</code>	<code>EMPLOYEE_NUMBER</code>
D11	000200
D11	000220
E21	000329

Figure 4.8

Normalized entities: `EMPLOYEE`, `DEPARTMENT`, and `EMPLOYEE_DEPARTMENT`

Key				
EMPID	SKILL_CODE	LANGUAGE_CODE	SKILL_PROFICIENCY	LANGUAGE_PROFICIENCY

Figure 4.9

An entity that violates fourth normal form

DEPARTMENT_NUMBER and DEPARTMENT_NAME. Now, an update such as changing a department name is much easier. You need to make the update only to the DEPARTMENT entity.

Fourth normal form

An entity is in fourth normal form if no instance contains two or more independent, multivalued facts about an entity.

Example: Consider the EMPLOYEE entity. Each instance of EMPLOYEE could have both SKILL_CODE and LANGUAGE_CODE. An employee can have several skills and know several languages. Two relationships exist, one between employees and skills, and one between employees and languages. An entity is not in fourth normal form if it represents both relationships, as Figure 4.9 shows.

Instead, you can avoid this violation by creating two entities that represent both relationships, as Figure 4.10 shows.

If, however, the facts are interdependent (that is, the employee applies certain languages only to certain skills), you should *not* split the entity.


You can put any data into fourth normal form. A good rule to follow when doing logical database design is to arrange all the data in entities that are in fourth normal form. Then decide whether the result gives you an acceptable level of performance. If the performance is not acceptable, denormalizing your design is a good approach to improving performance. You can read about this next step in “Denormalizing tables to improve performance” on page 102.

Key			Key		
EMPID	SKILL_CODE	SKILL_PROFICIENCY	EMPID	LANGUAGE_CODE	LANGUAGE_PROFICIENCY

Figure 4.10

Entities that are in fourth normal form

Logical database design with Unified Modeling Language

This chapter describes the entity-relationship model of database design. Another model that you can use is Unified Modeling Language (UML). The Object Management Group is a consortium that created the UML standard. This section provides a brief overview of UML. 

UML modeling is based on object-oriented programming principals. The basic difference between the entity-relationship model and the UML model is that, instead of designing entities as this chapter illustrates, you model objects. UML defines a standard set of modeling diagrams for all stages of developing a software system. Conceptually, UML diagrams are like the blueprints for the design of a software development project.

Some examples of UML diagrams are listed below:

- **Class:** Identifies high-level entities, known as classes. A *class* describes a set of objects that have the same attributes. A class diagram shows the relationships between classes.
- **Use case:** Presents a high-level view of a system from the user's perspective. A *use case* diagram defines the interactions between users and applications or between applications. These diagrams graphically depict system behavior. You can work with use-case diagrams to capture system requirements, learn how the system works, and specify system behavior.
- **Activity:** Models the workflow of a business process, typically by defining rules for the sequence of activities in the process. For example, an accounting company can use activity diagrams to model financial transactions.
- **Interaction:** Shows the required sequence of interactions between objects. Interaction diagrams can include sequence diagrams and collaboration diagrams.
 - Sequence diagrams show object interactions in a time-based sequence that establishes the roles of objects and helps determine class responsibilities and interfaces.
 - Collaboration diagrams show associations between objects that define the sequence of messages that implement an operation or a transaction.
- **Component:** Shows the dependency relationships between components, such as main programs and subprograms.

Many available tools from the WebSphere and Rational[®] product families ease the task of creating a UML model. Developers can graphically represent the

architecture of a database and how it interacts with applications using the following UML modeling tools:

- WebSphere Business Integration Workbench, which provides a UML modeler for creating standard UML diagrams.
- A WebSphere Studio Application Developer plug-in for modeling Java and Web services applications and for mapping the UML model to the entity-relationship model.
- Rational Rose[®] Data Modeler, which provides a modeling environment that connects database designers using entity-relationship modeling with developers of OO applications.
- Rational Rapid Developer, an end-to-end modeler and code generator that provides an environment for rapid design, integration, construction, and deployment of Web, wireless, and portal-based business applications.

Similarities exist between components of the entity-relationship model and UML diagrams. For example, the class structure corresponds closely to the entity structure.

Using the Rational Rose Data Modeler, developers use a specific type of diagram for each type of development model:

- Business models—Use case diagram, activity diagram, sequence diagram
- Logical data models or application models—Class diagram
- Physical data models—Data model diagram

The logical data model provides an overall view of the captured business requirements as they pertain to data entities. The data model diagram graphically represents the physical data model. The physical data model applies the logical data model's captured requirements to specific DBMS languages. Physical data models also capture the lower-level detail of a DBMS database.

Database designers can customize the data model diagram from other UML diagrams, which allows them to work with concepts and terminology, such as columns, tables, and relationships, with which they are already familiar. Developers can also transform a logical data model into a physical data model.

Because the data model diagram includes diagrams for modeling an entire system, it allows database designers, application developers, and other development team members to share and track business requirements throughout development. For example, database designers can capture information, such as constraints, triggers, and indexes, directly on the UML diagram. Developers can also transfer between

object and data models and use basic transformation types such as many-to-many relationships.

Physical database design

After completing the logical design of your database, you now move to the physical design. The purpose of building a physical design of your database is to optimize performance while ensuring data integrity by avoiding unnecessary data redundancies. During physical design, you transform the entities into tables, the instances into rows, and the attributes into columns. You and your colleagues must decide on many factors that affect the physical design, some of which are listed below.

- How to translate entities into physical tables
- What attributes to use for columns of the physical tables
- Which columns of the tables to define as keys
- What indexes to define on the tables
- What views to define on the tables
- How to denormalize the tables
- How to resolve many-to-many relationships

Physical design is the time when you abbreviate the names that you chose during logical design. For example, you can abbreviate the column name that identifies employees, `EMPLOYEE_NUMBER`, to `EMPNO`. In previous versions of DB2, you needed to abbreviate column and table names to fit the physical constraint of an 18-byte limit. In Version 8, this task is less restrictive with the increase to a 30-byte maximum.

The task of building the physical design is a job that truly never ends. You need to continually monitor the performance and data integrity characteristics of the database as time passes. Many factors necessitate periodic refinements to the physical design.

DB2 lets you change many of the key attributes of your design with `ALTER SQL` statements. For example, assume that you design a partitioned table so that it will store 36 months' worth of data. Later you discover that you need to extend that design to hold 84 months' worth of data. You can add or rotate partitions for the current 36 months to accommodate the new design.

The remainder of this chapter includes some valuable information that can help you as you build and refine your database's physical design. However, this task generally requires more experience with DB2 than most readers of this book are likely to have.

Denormalizing tables to improve performance

“Normalizing your entities to avoid redundancy” on page 94 describes normalization only from the viewpoint of logical database design. This viewpoint is appropriate because the rules of normalization do not consider performance.

During physical design, analysts transform the entities into tables and the attributes into columns. Consider again the example in “Second normal form” on page 95. The warehouse address column first appears as part of a table that contains information about parts and warehouses. To further normalize the design of the table, analysts remove the warehouse address column from that table. Analysts also define the column as part of a table that contains information only about warehouses.

Normalizing tables is generally the recommended approach. What if applications require information about both parts and warehouses, including the addresses of warehouses? The premise of the normalization rules is that SQL statements can retrieve the information by joining the two tables. The problem is that, in some cases, performance problems can occur as a result of normalization. For example, some user queries might view data that is in two or more related tables; the result is too many joins. As the number of tables increases, the access costs can increase, depending on the size of the tables, the available indexes, and so on. For example, if indexes are not available, the join of many large tables might take too much time. You might need to denormalize your tables. *Denormalization* is the intentional duplication of columns in multiple tables, and it increases data redundancy.

Example: Consider the design in which both tables have a column that contains the addresses of warehouses. If this design makes join operations unnecessary, it could be a worthwhile redundancy. Addresses of warehouses do not change often, and if one does change, you can use SQL to update all instances fairly easily.



Tip: Do not automatically assume that all joins take too much time. If you join normalized tables, you do not need to keep the same data values synchronized in multiple tables. In many cases, joins are the most efficient access method, despite the overhead they require. For example, some applications achieve 44-way joins in sub-second response time.

When you are building your physical design, you and your colleagues need to decide whether to denormalize the data. Specifically, you need to decide whether to combine tables or parts of tables that are frequently accessed by joins that have high-performance requirements. This is a complex decision about which this book cannot give specific advice. To make the decision, you need to assess the performance requirements, different methods of accessing the data, and the costs of denormalizing the data. You need to consider the tradeoff: is duplication, in several tables, of often-requested columns less expensive than the time for performing joins?

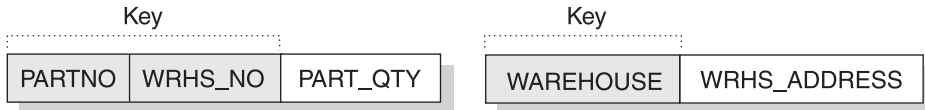


Figure 4.11
Two tables that satisfy second normal form



Recommendations:

- Do not denormalize tables unless you have a good understanding of the data and the business transactions that access the data. Consult with application developers before denormalizing tables to improve the performance of users' queries.
- When you decide whether to denormalize a table, consider all programs that regularly access the table, both for reading and for updating. If programs frequently update a table, denormalizing the table affects performance of update programs because updates apply to multiple tables rather than to one table.

In Figure 4.11, information about parts, warehouses, and warehouse addresses appears in two tables, both in normal form.

Figure 4.12 illustrates the denormalized table.

Resolving many-to-many relationships is a particularly important activity because doing so helps maintain clarity and integrity in your physical database design. To resolve many-to-many relationships, you introduce *associative tables*, which are intermediate tables that you use to tie, or associate, two tables to each other.

Example: Employees work on many projects. Projects have many employees. In the logical database design, you show this relationship as a many-to-many relationship between project and employee. To resolve this relationship, you create a new associative table, EMPLOYEE_PROJECT. For each combination of employee and project, the EMPLOYEE_PROJECT table contains a corresponding row. The primary key for the table would consist of the employee number (EMPNO) and the project number (PROJNO).



Figure 4.12
Denormalized table

Another decision that you must make relates to the use of repeating groups, which you read about in “First normal form” on page 95.

Example: Assume that a heavily used transaction requires the number of wires that are sold by month in a given year. Performance factors might justify changing a table so that it violates the rule of first normal form by storing repeating groups. In this case, the repeating group would be: MONTH, WIRE. The table would contain a row for the number of sold wires for each month (January wires, February wires, March wires, and so on).



Recommendation: If you decide to denormalize your data, document your denormalization thoroughly. Describe, in detail, the logic behind the denormalization and the steps that you took. Then, if your organization ever needs to normalize the data in the future, an accurate record is available for those who must do the work.

Using views to customize what data a user sees

Some users might find that no single table contains all the data they need; rather, the data might be scattered among several tables. Furthermore, one table might contain more data than users want to see or more than you want to authorize them to see. For those situations, you can create views. A view offers an alternative way of describing data that exists in one or more tables.

You might want to use views for a variety of reasons:

- To limit access to certain kinds of data

You can create a view that contains only selected columns and rows from one or more tables. Users with the appropriate authorization on the view see only the information that you specify in the view definition.

Example: You can define a view on the EMP table to show all columns except SALARY and COMM (commission). You can grant access to this view to people who are not managers because you probably don’t want them to have access to salary and commission information.

- To combine data from multiple tables

You can create a view that uses UNION or UNION ALL operators to logically combine smaller tables, and then query the view as if it were one large table.

Example: Assume that three tables contain data for a period of one month. You can create a view that is the UNION ALL of three fullselects, one for each month of the first quarter of 2004. At the end of the third month, you can view comprehensive quarterly data.

You can create a view any time after the underlying tables exist. The owner of a set of tables implicitly has the authority to create a view on them. A user with administrative authority at the system or database level can create a view for any owner on any set of tables. If they have the necessary authority, other users can also create views on a table that they didn't create. You can read more about authorization in "Authorizing users to access data" on page 328.

"Defining a view that combines information from several tables" on page 266 has more information about creating views.

Determining what columns to index

If you are involved in the physical design of a database, you will be working with other designers to determine what columns you should index. You will use process models that describe how different applications are going to be accessing the data. This information is important when you decide on indexing strategies to ensure adequate performance.

The main purposes of an index are as follows:

- **To optimize data access.** In many cases, access to data is faster with an index than without an index. If the DBMS uses an index to find a row in a table, the scan can be faster than when the DBMS scans an entire table.
- **To ensure uniqueness.** A table with a unique index cannot have two rows with the same values in the column or columns that form the index key.
Example: If payroll applications use employee numbers, no two employees can have the same employee number.
- **To enable clustering.** A clustering index keeps table rows in a specified sequence, to minimize page access for a set of rows. When a table space is partitioned, a special type of clustering occurs; rows are clustered within each partition.
Example: If the partition is on the month and the clustering index is on the name, the rows will be clustered on the name within the month.

In general, users of the table are unaware that an index is in use. DB2 decides whether to use the index to access the table.

You can read more about indexes in "Defining indexes" on page 254.

For more information

Table 4.1 lists additional information sources about topics that this chapter introduces.

Table 4.1 More information about topics in Chapter 4

For more information about...	Introduced in section that begins on page...	See...
Logical and physical relational data modeling theory and techniques	85	<ul style="list-style-type: none"> • <i>Handbook of Relational Database Design</i> by Candace C. Fleming and Barbara von Halle • <i>Data Modeling Essentials: Analysis, Design, and Innovation</i> by Graeme C. Simsion • <i>The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models</i> by Michael Reingruber and William Gregory • <i>Introduction to Information Engineering: From Strategic Planning to Information Systems</i> by Clive Finkelstein • <i>DB2 for z/OS and OS/390 Development for Performance Volume I, DB2 for z/OS and OS/390 Development for Performance Volume II</i> by Gabrielle Wiorkowski • <i>DB2 Developer's Guide</i> by Craig Mullins • <i>DB2 High Performance Design and Tuning</i> by Susan Lawson, Richard Yevich, and Warwick Ford • <i>DB2 Universal Database for Linux, UNIX, and Windows Administration Guide: Planning</i>

Table 4.1 More information about topics in Chapter 4 (Continued)

For more information about...	Introduced in section that begins on page...	See...
Process modeling	85	<ul style="list-style-type: none"> • <i>The Capability Maturity Model: Guidelines for Improving the Software Process</i> by Software Engineering Institute, Carnegie Mellon University • <i>Managing the Software Process</i> by Watts S. Humphrey
UML modeling	99	<i>Database Design for Smarties: Using UML for Data Modeling</i> by Robert Muller

Practice exam questions

The following practice exam questions test your knowledge of material that this chapter covers.

1. Which statement about domains is false?

- A. By defining a domain for a particular attribute, you apply business rules to ensure that the data will make sense.
- B. A domain describes the conditions that an attribute value must meet to be a valid value.
- C. A domain can identify a range of valid values.
- D. None of the above; all statements are true.

2. Which numeric data type is defined correctly?

- A. REAL data consists of single-precision floating-point numbers that are greater than 21.
- B. DOUBLE data is two-digit data in double-precision floating point numeric format.
- C. SMALLINT data is an integer less than 22.
- D. INTEGER data is for large integers.

3. Which two statements about null values are true?

- A. Depending on user-defined settings, the DBMS can treat a null value as a zero value, a blank, or an empty string.
- B. A null value is a special indicator that represents the absence of a value.
- C. A null value is equivalent to a blank.
- D. For an attribute that does not need to have a valid value at all times, use a null value.
- E. A null value is equivalent to a zero value.

4. Which statement is false?

- A. If a DBMS uses an index to find a row in a table, the performance is always better than when the DBMS scans the entire table.
- B. A table with a unique index can never have two rows with the same values in the columns that form the index key.
- C. A clustering index keeps table rows in a specified sequence, which always minimizes page access for a set of rows.
- D. All of the above.

5. Which statement about designing indexes is false?

- A. During physical database design, you decide what indexes to define on what columns of a table.
- B. You can use process models to determine how different applications and users will be accessing data.
- C. Users of a table with an index are generally aware that an index exists on the table.
- D. Decisions about indexes are important because of the performance implications they present.

6. Which statement does *not* explain a purpose of an index?

- A. An index can be used to optimize data access.
- B. An index can be used to ensure uniqueness.
- C. An index can be used to enable clustering.
- D. None of the above; all statements are correct.

Answers to practice exam questions

1. **Answer: D.**
2. **Answer: D.** The other definitions are incorrect. REAL data consists of single-precision floating-point numbers, which can be less than, equal to, or greater than 21. DOUBLE data consists of double-precision floating-point numbers. SMALLINT data consists of small integers, but they need not be less than 22.
3. **Answer: B and D.** The statements in the other options are false. Option A is false because no user-defined settings control how DB2 treats null values. Option C is false because a null value is not equivalent to a blank. Option E is false because a null value is not equivalent to a zero value.
4. **Answer: A.** The statements in the other options are true. Option A is false because using an index might improve performance, but in some cases a scan of the entire table actually results in better performance than when an index is used.
5. **Answer: C.** The statements in the other options are true. Option C is false because many users of tables are unaware of the presence or absence of particular indexes on those tables.
6. **Answer: D.** The statements in options A, B, and C do explain the purpose of an index.

Implementing your database design

- Defining tables 217
- Defining columns and rows in a table 223
- Defining a table space 240
- Defining indexes 254
- Defining views 265
- Defining large objects 268
- Defining databases 271
- Defining relationships with referential constraints 272
- Defining other business rules 277
- For more information 281
- Practice exam questions 284
- Answers to practice exam questions 287

Exam topics that this chapter covers

Accessing DB2 UDB data:

- Ability to create basic DB2 UDB objects

Working with DB2 UDB objects:


- Ability to demonstrate usage of DB2 UDB data types
- Given a situation, ability to create a table
- Knowledge to identify when referential constraints should be used
- Knowledge to identify methods of data validation
- Knowledge to identify characteristics of a table, view, or index

Earlier in this book, you read about different DB2 structures. In “Chapter 4. Designing objects and relationships,” you read about the task of building the logical and physical designs of your database. That task encompasses these key concepts:

- A table is a physical representation of an entity.
- A column is a physical representation of an entity’s attribute.
- A row is a physical representation of an instance of an entity.
- A primary key is a unique identifier for an instance of an entity.

After building a logical design and physical design of your relational database and collecting the processing requirements, you can move to the implementation stage. In general, implementing your physical design involves defining the various objects and enforcing the constraints on the data relationships.

The objects in a relational database are organized into sets called *schemas*. A schema provides a logical classification of objects in the database. The schema name is used as the qualifier of SQL objects such as tables, views, indexes, and triggers.

This chapter explains the task of implementing your database design in a way that most new users will understand. When you actually perform the task, you might perform the steps in a different order. 

You define, or create, objects by executing SQL statements. This chapter summarizes some of the naming conventions for the various objects that you can create. Also in this chapter, you will see examples of the basic SQL statements and keywords

that you can use to create objects in a DB2 database. (This chapter does not document the complete SQL syntax.) 


To illustrate how to create various objects, this chapter refers to the example tables, which you can see in “Appendix A. Example tables in this book.”



Tip: When you create DB2 objects (such as tables, databases, views, and indexes), you can precede the object name with a qualifier to distinguish it from objects that other people create. (For example, MYDB.TSPACE1 is a different table space than DSNDB04.TSPACE1.) When you use a qualifier, avoid using SYS as the first three characters. If you do not specify a qualifier, DB2 assigns a qualifier for the object.

Defining tables

Designing tables that many applications will use is a critical task. Table design can be difficult because you can represent the same information in many different ways. “Chapter 4. Designing objects and relationships” covers some of the issues that you need to consider when you make decisions about table design.

You create tables by using the SQL CREATE TABLE statement. At some point after you create and start using your tables, you might need to make changes to them. The ALTER TABLE statement lets you add and change columns, add or drop a primary key or foreign key, add or drop table check constraints, or add and change partitions. Carefully consider design changes to avoid or reduce the disruption to your applications. 

If you have DBADM (database administration) authority, you probably want to control the creation of DB2 databases and table spaces. These objects can have a big impact on the performance, storage, and security of the entire relational database. In most cases, you also want to retain the responsibility for creating tables. After designing the relational database, you can create the necessary tables for application programs. You can then pass the authorization for their use to the application developers, either directly or indirectly, by using views.


However, if you want to, you can grant the authority for creating tables to those who are responsible for implementing the application. For example, you probably want to authorize certain application programmers to create tables if they need temporary tables for testing purposes.

Some users in your organization might want to use DB2 with minimum assistance or control. You can define a separate storage group and database for these users and authorize them to create whatever data objects they need, such as tables. You can read more about authorization in “Authorizing users to access data” on page 328.

Types of tables

In DB2, you store user data in tables. DB2 supports the following types of tables:

Base table

The most common type of table in DB2. You create a base table with the SQL `CREATE TABLE` statement. The DB2 catalog table, `SYSIBM.SYSTABLES`, stores the description of the base table. The table (both its description and its data) is persistent. All programs and users that refer to this type of table refer to the same description of the table and to the same instance of the table. 

Result table

A table that contains a set of rows that DB2 selects or generates, directly or indirectly, from one or more base tables.

Created temporary table

A table that you define with the SQL `CREATE GLOBAL TEMPORARY TABLE` statement. The DB2 catalog table, `SYSIBM.SYSTABLES`, stores the description of the created temporary table. The description of the table is persistent and sharable. However, each individual application process that refers to a created temporary table has its own distinct instance of the table. That is, if application process A and application process B both use a created temporary table named `TEMPTAB`:

- Each application process uses the same table description.
- Neither application process has access to or knowledge of the rows in the other's instance of `TEMPTAB`.

Declared temporary table

A table that you define with the SQL `DECLARE GLOBAL TEMPORARY TABLE` statement. The DB2 catalog does not store a description of the declared temporary table. Therefore, neither the description nor the instance of the table is persistent. Multiple application processes can refer to the same declared temporary table by name, but they do not actually share the same description or instance of the table. For example, assume that application process A defines a declared temporary table named `TEMP1` with 15 columns. Application process B defines a declared temporary table named `TEMP1` with 5 columns. Each application process uses its own description of `TEMP1`; neither application process has access to or knowledge of rows in the other's instance of `TEMP1`.

Materialized query table

A table that you define with the SQL CREATE TABLE statement. Several DB2 catalog tables, including SYSIBM.SYSTABLES and SYSIBM.SYSVIEWS, store the description of the materialized query table and information about its dependency on a table, view, or function.

The attributes that define a materialized query table tell DB2 whether the table is:

- System-maintained or user-maintained.
- Refreshable: All materialized tables can be updated with the REFRESH TABLE statement. Only user-maintained materialized query tables can also be updated with the LOAD utility and the UPDATE, INSERT, and DELETE SQL statements.
- Enabled for query optimization: You can enable or disable the use of a materialized query table in automatic query rewrite (which you can read about in “Defining a materialized query table” on page 222).

Auxiliary table

A special kind of table that holds only large object data. You can read more about auxiliary tables in “Defining large objects” on page 268.

Base tables, temporary tables, and materialized query tables differ in many ways that this book does not describe. 

Table definitions

The table name is an identifier of up to 128 characters. You can qualify the table name with an SQL identifier, which is a schema. Remember that most organizations have naming conventions to ensure that objects are named in a consistent manner. When you define a table that is based directly on an entity, these factors also apply to the table names.

You can create base tables, temporary tables, auxiliary tables, or materialized query tables. You can read about creating auxiliary tables in “Defining large objects” on page 268. You can read about creating materialized query tables in “Defining a materialized query table” on page 222.

Defining a base table

To create a base table that you have designed, use the CREATE TABLE statement. When you create a table, DB2 records a definition of the table in the DB2 catalog.

Creating a table does not store the application data. You can put data into the table by using several methods, such as the LOAD utility or the INSERT statement.

Example: The following CREATE TABLE statement creates the EMP table, which is in a database named MYDB and in a table space named MYTS:

```
CREATE TABLE EMP
    (EMPNO      CHAR (6)          NOT NULL,
     FIRSTNAME  VARCHAR (12)     NOT NULL,
     LASTNAME   VARCHAR (15)     NOT NULL,
     DEPT       CHAR (3)          ,
     HIREDATE   DATE              ,
     JOB        CHAR (8)          ,
     EDL        SMALLINT         ,
     SALARY     DECIMAL (9, 2)    ,
     COMM       DECIMAL (9, 2)    ,
     PRIMARY KEY (EMPNO))
IN MYDB.MYTS;
```

The preceding CREATE TABLE statement shows the definition of multiple columns. You will learn about column definition in more detail in “Defining columns and rows in a table” on page 223.

Defining a temporary table

Temporary tables are especially useful when you need to do both of the following activities:

- Sort or query intermediate result tables that contain large numbers of rows
- Identify a small subset of rows to store permanently

You can use temporary tables to sort large volumes of data and to query that data. Then, when you have identified the smaller number of rows that you want to store permanently, you can store them in a base table. The two types of temporary tables in DB2 are the created temporary table and the declared temporary table. The following sections describe how to define each type.

Defining a created temporary table

Sometimes you need a permanent, sharable description of a table but need to store data only for the life of an application process. In this case, you can define and use a created temporary table. DB2 does not log operations that it performs on created temporary tables, so SQL statements that use them can execute more efficiently. Each application process has its own instance of the created temporary table.

Example: The following statement defines a created temporary table, TEMP-PROD:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD
(SERIALNO      CHAR(8)          NOT NULL,
DESCRIPTION   VARCHAR(60)     NOT NULL,
MFGCOSTAMT    DECIMAL(8,2)    ,
MFGDEPTNO     CHAR(3)         ,
MARKUPPCT     SMALLINT        ,
SALESDEPTNO   CHAR(3)         ,
CURDATE       DATE            NOT NULL);
```

Defining a declared temporary table

Sometimes you need to store data for the life of an application process, but you don't need a permanent, sharable description of the table. In this case, you can define and use a declared temporary table.

Unlike other DB2 DECLARE statements, DECLARE GLOBAL TEMPORARY TABLE is an executable statement that you can embed in an application program or issue interactively. You can also dynamically prepare the statement.

When a program in an application process issues a DECLARE GLOBAL TEMPORARY TABLE statement, DB2 creates an empty instance of the table. You can populate the declared temporary table by using INSERT statements, modify the table by using searched or positioned UPDATE or DELETE statements, and query the table by using SELECT statements. You can also create indexes on the declared temporary table. The definition of the declared temporary table exists as long as the application process runs.


Example: The following statement defines a declared temporary table, TEMP_EMP. (This example assumes that you have already created the TEMP database and corresponding table space for the temporary table.)

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
(EMPNO        CHAR(6)          NOT NULL,
SALARY        DECIMAL(9,2)    ,
COMM          DECIMAL(9,2) ) ;
```

All names of declared temporary tables must have SESSION as a qualifier.

At the end of an application process that uses a declared temporary table, DB2 deletes the rows of the table and implicitly drops the description of the table.

Defining a materialized query table

Materialized query tables improve the performance of complex queries that operate on very large amounts of data. Using a materialized query table, DB2 pre-computes the results of data that is derived from one or more tables. When you submit a query, DB2 can use the results stored in a materialized query table rather than compute the results from the underlying source tables on which the materialized query table is defined. If the rewritten query is less costly, DB2 chooses to optimize the query by using the rewritten query, a process called *automatic query rewrite*. 

To take advantage of automatic query rewrite, you must define, populate, and periodically refresh the materialized query table. You use the CREATE TABLE statement to create a new table as a materialized query table.

Example: The following CREATE TABLE statement defines a materialized query table named TRANSCNT. TRANSCNT summarizes the number of transactions in table TRANS by account, location, and year:

```
CREATE TABLE TRANSCNT (ACCTID, LOCID, YEAR, CNT) AS
  (SELECT ACCOUNTID, LOCATIONID, YEAR, COUNT(*)
   FROM TRANS
   GROUP BY ACCOUNTID, LOCATIONID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY SYSTEM
  ENABLE QUERY OPTIMIZATION;
```

The fullselect, together with the DATA INITIALLY DEFERRED clause and the REFRESH DEFERRED clause, defines the table as a materialized query table.

Defining a table with table-controlled partitioning

Before Version 8, when you defined a partitioning index on a table in a partitioned table space, you specified the partitioning key and the limit key values in the PART clause of the CREATE INDEX statement. This type of partitioning is known as *index-controlled partitioning*. Because the index is created separately from the associated table, you cannot insert data into the table until the partitioning index is created.

Version 8 introduces a new method, called *table-controlled partitioning*, for defining table partitions. You can use table-controlled partitioning instead of index-controlled partitioning. However, DB2 supports both methods in Version 8.

With table-controlled partitioning, you identify column values that delimit partition boundaries with the PARTITION BY clause and the PARTITION ENDING AT clause of the CREATE TABLE statement. When you use this type of partitioning, an index is not required for partitioning.

Example: Assume that you need to create a large transaction table that includes the date of the transaction in a column named `POSTED`. You want to keep the transactions for each month in a separate partition. To create the table, use the following statement:

```
CREATE TABLE TRANS
  (ACCTID ... ,
   STATE ... ,
   POSTED ... ,
   ... , ...)
PARTITION BY (POSTED)
(PARTITION 1 ENDING AT ('01/31/2003') ,
 PARTITION 2 ENDING AT ('02/28/2003') ,
 ...
 PARTITION 13 ENDING AT ('01/31/2004')) ;
```

Defining columns and rows in a table

After logical and physical database design is complete, you implement the definitions that were created during physical design. This section describes how to implement:

- Columns. See “Determining column attributes”
- Data types. See “Choosing a data type for the column” on page 224
- Null and default values. See “Using null and default values” on page 233
- Check constraints. See “Enforcing validity of column values with check constraints” on page 237

Throughout the implementation phase of database design, refer to the complete descriptions of SQL statement syntax and usage for each SQL statement that you work with.

Determining column attributes

A column contains values that have the same data type. If you are familiar with the concepts of records and fields, you can think of a *value* as a field in a record. A value is the smallest unit of data that you can manipulate with SQL. For example, in the `EMP` table, the `EMPNO` column identifies all employees by a unique employee number. The `HIREDATE` column contains the hire dates for all employees. You cannot overlap columns.


In Version 8, online schema enhancements provide flexibility that lets you change a column definition. Carefully consider the decisions that you make about column definitions. After you implement the design of your tables, you can change a column definition with minimal disruption of applications.

The two basic components of the column definition are the name and the data type.

Generally, the database administrator (DBA) is involved in determining the names of attributes (or columns) during the physical database design phase. To make the right choices for column names, DBAs follow the guidelines that the organization's data administrators have developed.

Sometimes columns need to be added to the database after the design is complete. In this case, DB2 rules for making column names unique must be followed. Column names must be unique within a table, but you can use the same column name in different tables. Try to choose a meaningful name to describe the data in a column to make your naming scheme intuitive. The maximum length of a column name is 30 bytes.

Choosing a data type for the column

“Choosing data types for attributes” on page 91 explains the need to determine what data type to use for each attribute. Every column in every DB2 table has a data type. The data type influences the range of values that the column can have and the set of operators and functions that apply to it. You specify the data type of each column at the time you create the table. 

In Version 8, you can also change the data type of a table column. The new data type definition is applied to all data in the associated table when the table is reorganized.

Some data types have parameters that further define the operators and functions that apply to the column. DB2 supports both IBM-supplied data types and user-defined data types. The data types that IBM supplies are sometimes called *built-in data types*. This section describes implementation of the following built-in data types:

- “String data types” on page 225
- “Numeric data types” on page 227
- “Date, time, and timestamp data types” on page 229
- “Large object data types” on page 230
- “ROWID data type” on page 231

In DB2 UDB for z/OS, user-defined data types are called *distinct types*. You can read more about distinct types in “Defining and using distinct types” on page 232.

String data types

DB2 supports several types of string data. *Character strings* contain text and can be either a fixed length or a varying length. *Graphic strings* contain graphic data, which can also be either a fixed length or a varying length. The third type of string data is *binary large object (BLOB) strings*, which you use for varying-length columns that contain strings of binary bytes. You will read more about BLOB data types in “Defining large objects” on page 268.

Table 7.1 describes the different string data types and indicates the range for the length of each string data type.

Table 7.1 String data types

Data type	Denotes a column of...
CHARACTER(<i>n</i>)	Fixed-length character strings with a length of <i>n</i> bytes. <i>n</i> must be greater than 0 and not greater than 255. The default length is 1.
VARCHAR(<i>n</i>)	Varying-length character strings with a maximum length of <i>n</i> bytes. <i>n</i> must be greater than 0 and less than a number that depends on the page size of the table space.
CLOB(<i>n</i>)	Varying-length character strings with a maximum of <i>n</i> characters. <i>n</i> cannot exceed 2,147,483,647. The default length is 1.
GRAPHIC(<i>n</i>)	Fixed-length graphic strings containing <i>n</i> double-byte characters. <i>n</i> must be greater than 0 and less than 128. The default length is 1.
VARGRAPHIC(<i>n</i>)	Varying-length graphic strings. The maximum length, <i>n</i> , must be greater than 0 and less than a number that depends on the page size of the table space.
DBCLOB(<i>n</i>)	Varying-length string of double-byte characters with a maximum of <i>n</i> double-byte characters. <i>n</i> cannot exceed 1,073,741,824. The default length is 1.
BLOB(<i>n</i>)	Varying-length binary string with a length of <i>n</i> bytes. <i>n</i> cannot exceed 2,147,483,647. The default length is 1.


In most cases, the content of the data that a column will store dictates the data type that you choose.

Example: The DEPT table has a column, DEPTNAME. The data type of the DEPTNAME column is VARCHAR(36). Because department names normally vary considerably in length, the choice of a varying-length data type seems appropriate.

If you choose a data type of CHAR(36), for example, the result is a lot of wasted, unused space. DB2 would assign all department names, regardless of length, the same amount of space (36 bytes). A data type of CHAR(6) for the employee number (EMPNO) is a reasonable choice because all values are fixed-length values (6 bytes).

Choosing the encoding scheme

Within a string, all the characters are represented by a common encoding representation. You can encode strings in Unicode, ASCII, or EBCDIC.

Multinational companies that engage in international trade often store data from more than one country in the same table. Some countries use different coded character set identifiers. DB2 UDB for z/OS supports the Unicode encoding scheme, which represents many different geographies and languages. (Unicode UTF-8 is for mixed-character data, and UCS2 or UTF-16 is for graphic data.) If you need to perform character conversion on Unicode data, the conversion is more likely to preserve all of your information. 

In some cases, you might need to convert characters to a different encoding representation. The process of conversion is known as *character conversion*. Most users do not need a knowledge of character conversion. When character conversion does occur, it does so automatically and a successful conversion is invisible to the application and users.

Choosing CHAR or VARCHAR

Using VARCHAR saves disk space, but it incurs a 2-byte overhead cost for each value. Using VARCHAR also requires additional processing for varying-length rows. Therefore, using CHAR is preferable to using VARCHAR unless the space that you save with VARCHAR is significant. The savings are not significant if the maximum column length is small or if the lengths of the values do not have a significant variation.



Recommendations:

- Generally, do not define a column as VARCHAR(*n*) or CLOB(*n*) unless *n* is at least 18 characters.
- Place VARCHAR and CLOB columns after the fixed-length columns of the table for better performance.

Using string subtypes

If an application that accesses your table uses a different encoding scheme than your DBMS uses, the following string subtypes can be important:

BIT

Does not represent characters.

SBCS

Represents single-byte characters.

MIXED

Represents single-byte characters and multibyte characters.

String subtypes apply only to CHAR, VARCHAR, and CLOB data types.

Choosing graphic or mixed data

When columns contain *double-byte character set (DBCS)* characters, you can define them as either graphic data or mixed data.

Graphic data can be either GRAPHIC, VARGRAPHIC, or DBCLOB. Using VARGRAPHIC saves disk space, but it incurs a 2-byte overhead cost for each value. Using VARGRAPHIC also requires additional processing for varying-length rows. Therefore, using GRAPHIC data is preferable to using VARGRAPHIC unless the space that you save by using VARGRAPHIC is significant. The savings are not significant if the maximum column length is small or if the lengths of the values do not vary significantly.



Recommendation: Generally, do not define a column as VARGRAPHIC(*n*) unless *n* is at least 18 double-byte characters (which is a length of 36 bytes).

Mixed-data character string columns can contain both *single-byte character set (SBCS)* and DBCS characters. You can specify the mixed-data character string columns as CHAR, VARCHAR, or CLOB with MIXED DATA.



Recommendation: If all of the characters are DBCS characters, use the graphic data types. (Kanji is an example of a language that requires DBCS characters.) For SBCS characters, use mixed data to save 1 byte for every single-byte character in the column.

Numeric data types

For numeric data, use numeric columns rather than string columns. Numeric columns require less space than string columns, and DB2 verifies that the data has the assigned type.

Example: Assume that DB2 is calculating a range between two numbers. If the values have a string data type, DB2 assumes that the values can include all combinations of alphanumeric characters. In contrast, if the values have a numeric data type, DB2 can calculate a range between the two values more efficiently.

Table 7.2 describes the numeric data types.

Table 7.2 Numeric data types

Data type	Denotes a column of...
SMALLINT	Small integers. A <i>small integer</i> is an IBM System/390 2-byte binary integer of 16 bits; the range is $-32,768$ to $+32,767$.
INTEGER or INT	Large integers. A <i>large integer</i> is an IBM System/390 fullword binary integer of 32 bits; the range is $-2,147,483,648$ to $+2,147,483,647$.
DECIMAL or NUMERIC	IBM System/390 packed-decimal numbers with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits. All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is $-n$ to $+n$, where n is the largest positive number that can be represented with the applicable precision and scale. The maximum range is $1 - 10^{*31}$ to $10^{*31} - 1$.
REAL	A <i>single-precision floating-point number</i> is an IBM System/390 short floating-point number of 32 bits. The range of single precision floating-point numbers is approximately $-7.2E+75$ to $7.2E+75$.
DOUBLE	A <i>double-precision floating-point number</i> is an IBM System/390 long floating-point number of 64 bits. The range of double precision floating-point numbers is approximately $-7.2E+75$ to $7.2E+75$.

Note: zSeries and z/Architecture use the S/390[®] format and support IEEE floating point.

For integer values, SMALLINT or INTEGER (depending on the range of the values) is generally preferable to DECIMAL.

You can define an exact numeric column as an identity column. An *identity column* has an attribute that enables DB2 to automatically generate a unique numeric value

for each row that is inserted into the table. Identity columns are ideally suited to the task of generating unique primary-key values. Applications that use identity columns might be able to avoid concurrency and performance problems that sometimes occur when applications implement their own unique counters. You can read more about concurrency and performance in “Improving performance for multiple users: Locking and concurrency” on page 301.

Date, time, and timestamp data types

Although you might consider storing dates and times as numeric values, instead you can take advantage of the datetime data types: DATE, TIME, and TIMESTAMP.

Table 7.3 describes the data types for dates, times, and timestamps.

Table 7.3 Date, time, and timestamp data types

Data type	Denotes a column of...
DATE	Dates. A <i>date</i> is a three-part value representing a year, month, and day in the range of 0001-01-01 to 9999-12-31.
TIME	Times. A <i>time</i> is a three-part value representing a time of day in hours, minutes, and seconds, in the range of 00.00.00 to 24.00.00.
TIMESTAMP	Timestamps. A <i>timestamp</i> is a seven-part value representing a date and time by year, month, day, hour, minute, second, and microsecond, in the range of 0001-01-01-00.00.00.000000 to 9999-12-31-24.00.00.000000.

DB2 stores values of datetime data types in a special internal format. When you load or retrieve data, DB2 can convert it to or from any of the formats in Table 7.4.

Table 7.4. Date and time format options

Format name	Abbreviation	Typical date	Typical time
International Standards Organization	ISO	2003-12-25	13.30.05
IBM USA standard	USA	12/25/2003	1:30 PM
IBM European standard	EUR	25.12.2003	13.30.05
Japanese Industrial Standard Christian Era	JIS	2003-12-25	13:30:05

Example: The following query displays the dates on which all employees were hired, in IBM USA standard form, regardless of the local default:

```
SELECT EMPNO, CHAR(HIREDATE, USA) FROM EMP;
```

When you use datetime data types, you can take advantage of DB2 built-in functions that operate specifically on datetime values and you can specify calculations for datetime values.

Example: Assume that a manufacturing company has an objective to ship all customer orders within five days. You define the SHIPDATE and ORDERDATE columns as DATE data types. The company can use datetime data types and the DAYS built-in function to compare the shipment date to the order date. Here is how the company might code the function to generate a list of orders that have exceeded the five-day shipment objective:

```
DAYS (SHIPDATE) - DAYS (ORDERDATE) > 5
```

As a result, programmers don't need to develop, test, and maintain application code to perform complex datetime arithmetic that needs to allow for the number of days in each month.

You can use the following sample user-defined functions (which come with DB2) to modify the way dates and times are displayed. 

- **ALTDATE** returns the current date in a user-specified format or converts a user-specified date from one format to another.
- **ALTTIME** returns the current time in a user-specified format or converts a user-specified time from one format to another.

At installation time, you also have the option of supplying an exit routine to make conversions to and from any local standard.

When loading date or time values from an outside source, DB2 accepts any format that Table 7.4 lists. DB2 converts valid input values to the internal format. For retrieval, a default format is determined when installing DB2. You can override that default by using a precompiler option for all statements in a program or by using the scalar function CHAR for a particular SQL statement and specifying the desired format.

“Preparing an application program to run” on page 182 has information about the precompiler.

Large object data types

The VARCHAR and VARGRAPHIC data types have a storage limit of 32 KB. Although this limit might be sufficient for small- to medium-size text data, applications often

need to store large text documents. They might also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images.

If the size of the data is greater than 32 KB, use the corresponding LOB data type. Storing such data as LOB data rather than as VARCHAR FOR BIT DATA provides advantages, even if the entire row fits on a page.

DB2 provides three LOB data types to store these data objects as strings of up to 2 GB in size:

- Character large objects (CLOBs)
Use CLOB to store SBCS or mixed data, such as documents that are written with a single character set. Use this data type if your data is larger (or may grow larger) than VARCHAR permits.
- Double-byte character large objects (DBCLOBs)
Use DBCLOB to store large amounts of DBCS data, such as documents that are written with a DBCS character set.
- Binary large objects (BLOBs)
Use BLOB to store large amounts of noncharacter data, such as pictures, or voice and mixed media.

If your data does not fit entirely within a data page, you can define one or more columns as LOB columns. An advantage to using LOBs is that you can create user-defined functions that are allowed only on LOB data types. “Large object table spaces” on page 248 has more information about the advantages of using LOBs.

ROWID data type

You use the ROWID data type to uniquely and permanently identify rows in a DB2 subsystem. DB2 can generate a value for the column when a row is added, depending on the option that you choose (GENERATED ALWAYS or GENERATED BY DEFAULT) when you define the column. You can use a ROWID column in a table for several reasons.


- You can define a ROWID column to include LOB data in a table; you can read about large objects in “Defining large objects” on page 268.
- You can use the ROWID column as a partitioning key for partitioned table spaces; you can read about partitioned table spaces in “Defining partitioned table spaces” on page 246.
- You can use direct-row access so that DB2 accesses a row directly through the ROWID column. If an application selects a row from a table that contains a ROWID column, the row ID value implicitly contains the location

of the row. If you use that row ID value in the search condition of subsequent SELECT statements, DB2 might be able to navigate directly to the row.

Comparing data types

DB2 compares values of different types and lengths. A comparison occurs when both values are numeric, both values are character strings, or both values are graphic strings. Comparisons can also occur between character and graphic data or between character and datetime data if the character data is a valid character representation of a datetime value. Different types of string or numeric comparisons might have an impact on performance.

Defining and using distinct types

A *distinct type* is a user-defined data type that is based on existing built-in DB2 data types. That is, they are internally the same as built-in data types, but DB2 treats them as a separate and incompatible type for semantic purposes. Defining your own distinct types ensures that only functions that are explicitly defined on a distinct type can be applied to its instances. 

Example: You might define a US_DOLLAR distinct type that is based on the DB2 DECIMAL data type to identify decimal values that represent United States dollars. The US_DOLLAR distinct type does not automatically acquire the functions and operators of its source type, DECIMAL.

Although you can have different distinct types based on the same built-in data types, distinct types have the property of *strong typing*. With this property, you cannot directly compare instances of a distinct type with anything other than another instance of that same type. Strong typing prevents semantically incorrect operations (such as explicit addition of two different currencies) without first undergoing a conversion process. You define which types of operations can occur for instances of a distinct type.

If your company wants to track sales in many countries, you must convert the currency for each country in which you have sales.

Example: You can define a distinct type for each country. For example, to create US_DOLLAR types and CANADIAN_DOLLAR types, you can use the following CREATE DISTINCT TYPE statements:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2);  
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2);
```

Example: After you define distinct types, you can use them in your CREATE TABLE statements:

```
CREATE TABLE US_SALES
(PRODUCT_ITEM_NO    INTEGER,
 MONTH              INTEGER,
 YEAR               INTEGER,
 TOTAL_AMOUNT       US_DOLLAR) ;

CREATE TABLE CANADIAN_SALES
(PRODUCT_ITEM_NO    INTEGER,
 MONTH              INTEGER,
 YEAR               INTEGER,
 TOTAL_AMOUNT       CANADIAN_DOLLAR) ;
```

User-defined functions support the manipulation of distinct types. You can read about defining user-defined functions in “Defining user-defined functions” on page 278.

Using null and default values

As you create table columns, you will discover that the content of some columns cannot always be specified; users and applications must be allowed to not supply a value. This section explains the use of null values and default values and provides some tips on when to choose each type of value.

Null values

Some columns cannot have a meaningful value in every row. DB2 uses a special value indicator, the null value, to stand for an unknown or missing value. “Null values” on page 93 introduces the concept of a null value, which is an actual value and not a zero value, a blank, or an empty string. It is a special value that DB2 interprets to mean that no data is present.

If you do not specify otherwise, the default is that any column can contain null values. Users can create rows in the table without providing a value for the column.

The NOT NULL clause disallows null values in the column. Primary keys must be defined as NOT NULL.

Example: The table definition for the DEPT table specifies when you can use a null value. Notice that you can use nulls for the MGRNO column only:

```
CREATE TABLE DEPT
(DEPTNO          CHAR (3)          NOT NULL,
 DEPTNAME        VARCHAR (36)      NOT NULL,
 MGRNO           CHAR (6)          ,
 ADMRDEPT        CHAR (3)          NOT NULL,
 PRIMARY KEY (DEPTNO)              )
IN MYDB.MYTS;
```

Before you decide whether to allow nulls for unknown values in a particular column, you should be aware of how nulls affect results of a query:

- Nulls in application programs

Nulls do not satisfy any condition in an SQL statement other than the special IS NULL predicate. DB2 sorts null values differently than nonnull values.

Null values do not behave like other values. For example, if you ask DB2 whether a null value is larger than a given known value, the answer is UNKNOWN. If you then ask DB2 whether a null value is smaller than the same known value, the answer is still UNKNOWN.

If getting UNKNOWN is unacceptable for a particular column, you could define a default value instead. Programmers are familiar with the way default values behave.

- Nulls in a join operation

Nulls need special handling in join operations. If you perform a join operation on a column that can contain null values, consider using an outer join. (You read about joins in “Joining data from more than one table” on page 152.)

Default values

DB2 defines some default values, and you define others (by using the DEFAULT clause in the CREATE TABLE or ALTER TABLE statement).

- If the column is defined as NOT NULL WITH DEFAULT or if you do not specify NOT NULL, DB2 stores a default value for a column whenever an insert or load does not provide a value for that column.
- If the column is defined as NOT NULL, DB2 does not supply a default value.

DB2-defined defaults

DB2 generates a default value for ROWID columns. DB2 also determines default values for columns that users define with NOT NULL WITH DEFAULT, but for which no specific value is specified. See Table 7.5.

Table 7.5 DB2-defined default values for data types

For columns of...	Data types	Default
Numbers	SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL, DOUBLE, or FLOAT	0
Fixed-length strings	CHAR or GRAPHIC	Blanks
Varying-length strings	VARCHAR, CLOB, VARGRAPHIC, DBCLOB, or BLOB	Empty string
Dates	DATE	CURRENT DATE
Times	TIME	CURRENT TIME
Timestamps	TIMESTAMP	CURRENT TIMESTAMP
ROWIDs	ROWID	DB2-generated

User-defined defaults

You can specify a particular default, such as:

```
DEFAULT 'N/A'
```


When you choose a default value, you must be able to assign it to the data type of the column. For example, all string constants are VARCHAR. You can use a VARCHAR string constant as the default for a CHAR column even though the type isn't an exact match. However, you could not specify a default value of 'N/A' for a column with a numeric data type.

In the next example, the columns are defined as CHAR (fixed length). The special registers (USER and CURRENT SQLID) that are referenced are varying length starting in Version 8. This example is valid.

Example: If you want a record of each user who inserts any row of a table, define the table with two additional columns:

```
PRIMARY_ID    CHAR(8)           WITH DEFAULT USER,
SQL_ID        CHAR(8)           WITH DEFAULT CURRENT SQLID,
```

You can then create a view that omits those columns and allows users to update the view instead of the base table. DB2 then adds, by default, the primary authorization ID and the SQLID of the process. You can read about authorization in “Authorizing users to access data” on page 328.

When you add columns to an existing table, you must define them as nullable or as not null with default. Assume that you add a column to an existing table and specify not null with default. If DB2 reads from the table before you add data to the column, the column values that you retrieve are the default values. With few exceptions, the default values for retrieval are the same as the default values for insert. 

Default for ROWID

DB2 always generates the default values for ROWID columns.

Comparing null values and default values

In some situations, using a null value is easier and better than using a default value.

Example: Suppose that you want to find out the average salary for all employees in a department. The salary column does not always need to contain a meaningful value, so you can choose between the following options:

- Allowing null values for the SALARY column
- Using a nonnull default value (such as, 0)

By allowing null values, you can formulate the query easily, and DB2 provides the average of all known or recorded salaries. The calculation does not include the rows that contain null values. In the second case, you probably get a misleading answer unless you know the nonnull default value for unknown salaries and formulate your query accordingly.

Figure 7.1 shows two scenarios. The table in the figure excludes salary data for employee number 200440 because the company just hired this employee and has not yet determined the salary. The calculation of the average salary for department E21 varies, depending on whether you use null values or nonnull default values.

- The left side of the figure assumes that you use null values. In this case, the calculation of average salary for department E21 includes only the three employees (000320, 000330, and 200340) for whom salary data is available.
- The right side of the figure assumes that you use a nonnull default value of zero (0). In this case, the calculation of average salary for department E21 includes all four employees, although valid salary information is available for only three employees. As you can see, only the use of a null value results in an accurate average salary for department E21.

```
SELECT DEPT, AVG(SALARY)
FROM EMP
GROUP BY DEPT;
```

With null value

EMPNO	DEPT	SALARY
000320	E21	19950.00
000330	E21	25370.00
200340	E21	23840.00
200440	E21	-----

↓

DEPT	AVG(SALARY)
====	=====
.	.
.	.
E21	23053.33

(Average of nonnull salaries)

With default value of 0

EMPNO	DEPT	SALARY
000320	E21	19950.00
000330	E21	25370.00
200340	E21	23840.00
200440	E21	0.00

↓

DEPT	AVG(SALARY)
====	=====
.	.
.	.
E21	17290.00


Figure 7.1

When nulls are preferable to default values

Enforcing validity of column values with check constraints

“Check constraints” on page 57 explains that a check constraint is a rule that specifies the values that are allowed in one or more columns of every row of a table. You can use check constraints to ensure that only values from the domain for the column or attribute are allowed. As a result of using check constraints, programmers don’t need to develop, test, and maintain application code that performs these checks.

You can choose to define check constraints by using the SQL CREATE TABLE statement or ALTER TABLE statement. For example, you might want to ensure that each value in the SALARY column of the EMP table contains more than a certain minimum amount.

DB2 enforces a check constraint by applying the relevant search condition to each row that is inserted, updated, or loaded. An error occurs if the result of the search condition is false for any row. 

Inserting rows into tables with check constraints

When you use the `INSERT` statement to add a row to a table, DB2 automatically enforces all check constraints for that table. If the data violates any check constraint that is defined on that table, DB2 does not insert the row.

Example: Assume that the `NEWEMP` table has the following two check constraints:

- Employees cannot receive a commission that is greater than their salary.
- Department numbers must be between '001' and '100,' inclusive.

Consider this `INSERT` statement, which adds an employee who has a salary of \$65,000 and a commission of \$6,000:

```
INSERT INTO NEWEMP
  (EMPNO, FIRSTNME, LASTNAME, DEPT, JOB, SALARY, COMM)
VALUES ('100125', 'MARY', 'SMITH', '055', 'SLS',
  65000.00, 6000.00);
```

The `INSERT` statement in this example succeeds because it satisfies both constraints.

Example: Consider this `INSERT` statement:

```
INSERT INTO NEWEMP
  (EMPNO, FIRSTNME, LASTNAME, DEPT, JOB, SALARY, COMM)
VALUES ('120026', 'JOHN', 'SMITH', '055', 'DES',
  5000.00, 55000.00 );
```

The `INSERT` statement in this example fails because the \$55,000 commission is higher than the \$5,000 salary. This `INSERT` statement violates a check constraint on `NEWEMP`.

“Loading the tables” on page 277 provides more information about loading data into tables on which you have defined check constraints.

Updating tables with check constraints

DB2 automatically enforces all check constraints for a table when you use the `UPDATE` statement to change a row in the table. If the intended update violates any check constraint that is defined on that table, DB2 does not update the row.

Example: Consider this `UPDATE` statement:

```
UPDATE NEWEMP
  SET DEPT = '011'
  WHERE FIRSTNME = 'MARY' AND LASTNAME= 'SMITH';
```

This update succeeds because it satisfies the constraints that are defined on the NEWEMP table.

Example: Consider this UPDATE statement:

```
UPDATE NEWEMP
  SET DEPT = '166'
  WHERE FIRSTNAME = 'MARY' AND LASTNAME= 'SMITH' ;
```

This update fails because the value of DEPT is '166,' which violates the check constraint on NEWEMP that DEPT values must be between '001' and '100.'

Designing rows

An important consideration in the design of a table is the record size. In DB2, a *record* is the storage representation of a row. DB2 stores records within pages that are 4 KB, 8 KB, 16 KB, or 32 KB in size. Generally, you cannot create a table with a maximum record size that is greater than the page size. No other absolute limit exists, but you risk wasting storage space if you ignore record size in favor of implementing a good theoretical design.

If the record length is larger than the page size, consider using a large object (LOB) data type (described in “Large object data types” on page 230).

Record length—fixed or varying

In a table whose columns all have fixed-length data types, all rows (and therefore all records) are the same size. Otherwise, the size of records can vary.

Fixed-length records are generally preferable to varying-length records because DB2 processing is most efficient for fixed-length records. A fixed-length record never needs to move from the page on which it is first stored. Updates to varying-length records, however, can cause the record length to grow so that it no longer fits on the original page. In that case, the record moves to another page. Each time that a record is accessed, an additional page reference occurs. Therefore, use varying-length columns only when necessary.

Record lengths and pages

The sum of the lengths of all the columns is the *record length*. The length of data that is physically stored in the table is the record length plus DB2 overhead for each row and each page.

If row sizes are very small, use the 4-KB page size. Use the default of 4-KB page sizes when access to your data is random and typically requires only a few rows from each page.

Some situations require larger page sizes. DB2 provides three larger page sizes of 8 KB, 16 KB, and 32 KB to allow for longer records. For example, when the size of individual rows is greater than 4 KB, you must use a larger page size. In general, you can improve performance by using pages for record lengths that best suit your needs.

Designs that waste space

Space is wasted in a table space that contains only records that are slightly longer than half a page because a page can hold only one record. If you can reduce the record length to just under half a page, you need only half as many pages. Similar considerations apply to records that are just over a third of a page, a quarter of a page, and so on.

Defining a table space

This section provides more detailed information about three different types of table spaces—segmented, partitioned, and LOB. Each type of table space has its own advantages and disadvantages. This information will help you choose the table space that best suits your needs. This section also summarizes the process of defining table spaces.

DB2 divides table spaces into equal-sized units, called *pages*, which are written to or read from disk in one operation. You can specify page sizes for the data; the default page size is 4 KB.



Recommendation: Use partitioned table spaces for all table spaces that are referred to in queries that can take advantage of query parallelism. Use segmented table spaces for other queries. The explanations of the different table space types will help you decide.

General naming guidelines for table spaces

A table space name is an identifier of up to eight characters, which you can qualify with a database name. The default database name is database DSNDB04. The following table space name is typical:

Object	Name
Table space	MYDB.MYTS


Coding guidelines for defining table spaces

DB2 stores the names and attributes of all table spaces in the SYSIBM.SYSTABLESPACE catalog table, regardless of whether you define the table spaces explicitly or implicitly.



Recommendation: For large tables, use a partitioned table space. For small tables, use a segmented table space.

Defining a table space explicitly

Use the CREATE TABLESPACE statement to create a table space explicitly. This statement allows you to specify the attributes of the table space. The following list introduces some of the clauses of the CREATE TABLESPACE statement that you will read about in this section. 

LOB

Indicates that the table space is to be a large object (LOB) table space.

DSSIZE

Indicates the maximum size, in GB, for each partition or, for LOB table spaces, each data set.

FREEPAGE *integer*

Specifies how often DB2 should leave a page of free space when the table space or partition is loaded or reorganized. You specify that DB2 should set aside one free page for every *integer* number of pages. Using free pages can improve performance for applications that perform high-volume inserts or that update variable-length columns.

PCTFREE *integer*

Indicates the percentage (*integer*) of each page that DB2 should leave as free space when the table is loaded or reorganized. Specifying PCTFREE can improve performance for applications that perform high-volume inserts or that update variable-length columns.

COMPRESS

Specifies that data is to be compressed. You can compress data in a table space and thereby store more data on each data page. “Compressing data” on page 297 has information about data compression.

BUFFERPOOL *bpname*

Identifies the buffer pool that this table space is to use and determines the page size of the table space. The buffer pool is a portion of memory in which DB2 temporarily stores data for retrieval. You can read about the effect of buffer pool size on performance in “Caching data: The role of buffer pools” on page 294.

LOCKSIZE

Specifies the size of locks that DB2 is to use within the table space. DB2 uses locks to protect data integrity. Use of locks results in some overhead processing costs, so choose the lock size carefully. You can read about locking in “Improving performance for multiple users: Locking and concurrency” on page 301.

You can create segmented, partitioned, and LOB table spaces.

This section provides an overview of how to implement segmented and partitioned table spaces. “Defining large objects” on page 268 has information about LOB table spaces.

A segmented table space can hold one or more tables. Segmented table spaces hold a maximum of 64 GB of data. They might use one or more VSAM data sets. A table space can be larger if either of the following conditions is true:

- The table space is a partitioned table space that you create with the `DSSIZE` option.
- The table space is a LOB table space.

Table space pages are either 4 KB, 8 KB, 16 KB, or 32 KB in size. As a general rule, each DB2 database should have no more than 50 to 100 table spaces. Following this guideline helps minimize maintenance, increase concurrency, and decrease log volume.

Defining a table space implicitly

For small tables, you implicitly create a segmented table space when you use the `CREATE TABLE` statement to create a table and do not specify an existing table space name. When this occurs, DB2 performs the following tasks:

- Generates a table space for you
- Derives a table space name from the name of your table
- Uses default values for space allocation and other table space attributes

One or more tables are created for segmented table spaces.

For large tables, you need to explicitly create a partitioned table space or a LOB table space before you create a table. One table is created for partitioned and LOB table spaces. If your CREATE TABLE statement does not specify a database name, DB2 uses the default database, DSNDB04, and the default DB2 storage group, SYSDEFLT.

You also need to explicitly create a table space when you define a declared temporary table. (You read about declared temporary tables in “Types of tables” on page 218.)

Segmented table spaces

A segmented table space is ideal for storing more than one table, especially relatively small tables. The pages hold segments, and each segment holds records from only one table.

Each segment contains the same number of pages, which must be a multiple of 4 (from 4 to 64). Each table uses only as many segments as it needs. 📖

To search all the rows for one table, you don't need to scan the entire table space. Instead, you can scan only the segments that contain that table. Figure 7.2 shows a possible organization of segments in a segmented table space.

When you use the INSERT statement or the LOAD utility to insert records into a table, records from the same table are stored in different segments. You can reorganize the table space to move segments of the same table together. You can read more about reorganization and other techniques that influence performance of your DB2 subsystem in “Chapter 8. Managing DB2 performance.”

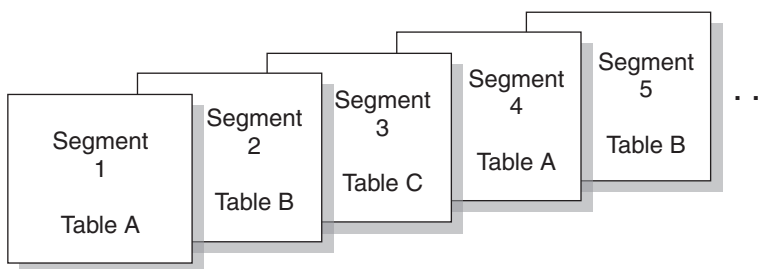


Figure 7.2

A possible organization of segments in a segmented table space

Coding the definition of a segmented table space

A segmented table space consists of segments that hold the records of one table. You define a segmented table space by using the `CREATE TABLESPACE` statement with a `SEGSIZE` clause. If you use this clause, the value that you specify represents the number of pages in each segment. The value must be a multiple of 4 (from 4 to 64). The choice of the value depends on the size of the tables that you store. Table 7.6 summarizes the recommendations for `SEGSIZE`.



Table 7.6 Recommendations for `SEGSIZE`

Number of pages	<code>SEGSIZE</code> recommendation
≤ 28	4 to 28
> 28 < 128 pages	32
≥ 128 pages	64

Another clause of the `CREATE TABLESPACE` statement is `LOCKSIZE TABLE`. This clause is valid only for tables that are in segmented table spaces. DB2, therefore, can acquire locks that lock a single table, rather than the entire table space. You can read about locking in “Improving performance for multiple users: Locking and concurrency” on page 301.

If you want to leave pages of free space in a segmented table space, you must have at least one free page in each segment. Specify the `FREEPAGE` clause with a value that is less than the `SEGSIZE` value.

Example: If you use `FREEPAGE 30` with `SEGSIZE 20`, DB2 interprets the value of `FREEPAGE` as 19, and you get one free page in each segment.



You can read more about free space in “Using free space in data and index storage” on page 298.

If you are creating a segmented table space for use by declared temporary tables, you cannot specify the `FREEPAGE` or `LOCKSIZE` clause.

Characteristics of segmented table spaces


Segmented table spaces share the following characteristics:

- When DB2 scans all the rows for one table, only the segments that are assigned to that table need to be scanned. DB2 doesn’t need to scan the entire table space. Pages of empty segments do not need to be fetched.
- When DB2 locks a table, the lock does not interfere with access to segments of other tables. (You can read more about locking in “Improving performance for multiple users: Locking and concurrency” on page 301.)

- When DB2 drops a table, its segments become available for reuse immediately after the drop is committed without waiting for an intervening REORG utility job. (You can read more about this utility in “Determining when to reorganize data” on page 298.)
- When all rows of a table are deleted, all segments except the first segment become available for reuse immediately after the delete is committed. No intervening REORG utility job is necessary.
- A *mass delete*, which is the deletion of all rows of a table, operates much more quickly and produces much less log information. In some cases, you must delete each individual row. 
- If the table space contains only one table, segmenting it means that the COPY utility does not copy pages that are empty. The pages can be empty as a result of a dropped table or a mass delete.
- Some DB2 utilities, such as LOAD with the REPLACE option, RECOVER, and COPY, operate on only a table space or a partition, not on individual segments. Therefore, for a segmented table space, you must run these utilities on the entire table space. For a large table space, you might notice availability problems. 
- Maintaining the space map creates some additional overhead.

Creating fewer table spaces by storing several tables in one table space can help you avoid reaching the maximum number of concurrently open data sets. Each table space requires at least one data set. A maximum number of concurrently open data sets is determined during installation. Using fewer table spaces means less time spent allocating and deallocating data sets.

Partitioned table spaces

You use a partitioned table space to store a single table. DB2 divides the table space into partitions. The partitions are based on the boundary values defined for specific columns. Utilities and SQL statements can run concurrently on each partition. 

In Figure 7.3, each partition contains one part of a table.

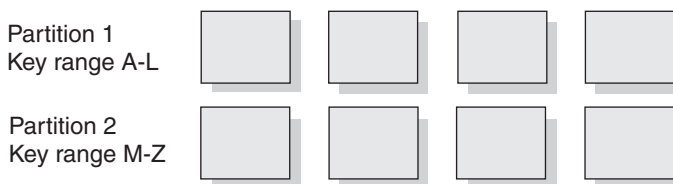



Figure 7.3
Pages in a partitioned table space

Defining partitioned table spaces


In a partitioned table space, you can think of each partition as a unit of storage. You use the `PARTITION` clause of the `CREATE TABLESPACE` statement to define a partitioned table space. For each partition that you specify in the `CREATE TABLESPACE` statement, DB2 creates a separate data set. You assign the number of partitions (from 1 to 4096), and you can assign partitions independently to different storage groups.

The maximum number of partitions in a table space depends on the data set size (`DSSIZE` parameter) and the page size. The size of the table space depends on the data set size and on how many partitions are in the table space. 

Characteristics of partitioned table spaces

Partitioned table spaces share the following characteristics:

- You can plan for growth. When you define a partitioned table space, DB2 usually distributes the data evenly across the partitions. Over time, the distribution of the data might become uneven as inserts and deletes occur. You can rebalance data among the partitions by redefining partition boundaries with no impact to availability. You can also add a partition to the table and to each partitioned index on the table; the new partition becomes available immediately.
- You can spread a large table over several DB2 storage groups or data sets. Not all the partitions of the table need to use the same storage group.
- Partitioned table spaces let a utility job work on part of the data while allowing other applications to concurrently access data on other partitions. In that way, several concurrent utility jobs can, for example, load all partitions of a table space concurrently. Because you can work on part of your data, some of your operations on the data may require less time.
- You can break mass update, delete, or insert operations into separate jobs, each of which works on a different partition. Breaking the job into several smaller jobs that run concurrently can reduce the elapsed time for the whole task.


If your table space uses nonpartitioned indexes, you might need to modify the size of data sets in the indexes to avoid I/O contention among concurrently running jobs. Use the `PIECESIZE` parameter of the `CREATE INDEX` or the `ALTER INDEX` statement to modify the sizes of the index data sets. 

- You can put frequently accessed data on faster devices. Evaluate whether table partitioning or index partitioning can separate more frequently

accessed data from the remainder of the table. You can put the frequently accessed data in a partition of its own. You can also use a different device type. You can read more about table and index partitioning later in this chapter.

- You can take advantage of parallelism for certain read-only queries. When DB2 determines that processing will be extensive, it can begin parallel processing of more than one partition at a time. Parallel processing (for read-only queries) is most efficient when you spread the partitions over different disk volumes and allow each I/O stream to operate on a separate channel. You can take advantage of query parallelism. Use the Parallel Sysplex data sharing technology to process a single read-only query across many DB2 subsystems in a data sharing group. You can optimize Parallel Sysplex query processing by placing each DB2 subsystem on a separate central processor complex. You can read more about Parallel Sysplex processing in “Chapter 12. Data sharing with your DB2 data”.
- Partitioned table space scans are sometimes less efficient than table space scans of segmented table spaces.
- DB2 opens more data sets when you access data in a partitioned table space than when you access data in other types of table spaces.
- Nonpartitioned indexes and data-partitioned secondary indexes are sometimes a disadvantage for partitioned tables spaces. You can read more about these types of indexes later in this chapter.

EA-enabled table spaces and index spaces

You can enable partitioned table spaces for extended addressability (EA), a function of DFSMS. The term for table spaces and index spaces that are enabled for extended addressability is *EA-enabled*. You must use EA-enabled table spaces or index spaces if you specify a maximum partition size (DSSIZE) that is larger than 4 GB in the CREATE TABLESPACE statement. 

Both EA-enabled and non-EA-enabled partitioned table spaces can have only one table and up to 4096 partitions. Table 7.7 summarizes the differences.

Table 7.7 Differences between EA-enabled and non-EA-enabled table spaces

EA-enabled table spaces	Non-EA-enabled table spaces
Holds up to 4096 partitions of 64 GB	Holds up to 4096 partitions of 4 GB
Created with any valid value of DSSIZE	DSSIZE cannot exceed 4 GB
Data sets are managed by SMS	Data sets are managed by VSAM or SMS
Requires setup	No additional setup

You can read more about this topic in “Assignment of table spaces to physical storage” on page 249.

Large object table spaces

LOB table spaces (also known as auxiliary table spaces) are necessary for holding large object data, such as graphics, video, or very large text strings. If your data does not fit entirely within a data page, you can define one or more columns as LOB columns.

LOB objects can do more than store large object data. You can also define LOB columns for infrequently accessed data; the result is faster table space scans on the remaining data in the base table. The table space scan is faster because potentially fewer pages are accessed.

A LOB table space always has a direct relationship with the table space that contains the logical LOB column values. The table space that contains the table with the LOB columns is, in this context, the *base table space*. LOB data is logically associated with the base table, but it is physically stored in an auxiliary table that resides in a LOB table space. Only one auxiliary table can exist in a large object table space. A LOB value can span several pages. However, only one LOB value is stored per page.

You must have a LOB table space for each LOB column that exists in a table. For example, if your table has LOB columns for both resumes and photographs, you need one LOB table space (and one auxiliary table) for each of those columns. If the base table space is a partitioned table space, you need one LOB table space for each LOB in each partition.

If the base table space is not a partitioned table space, each LOB table space is associated with one column of LOBs in a base table. If the base table space is a partitioned table space, each column of LOBs in each partition is associated with a LOB table space.

In a partitioned table space, you can store more LOB data in each column because each partition must have a LOB table space. Table 7.8 shows the approximate amount of data that you can store in one column for the different types of table spaces.

You can read more about the process of defining LOB table spaces in “Defining large objects” on page 268.

Table 7.8 Approximate maximum size of LOB data in a column

Table space type	Maximum (approximate) LOB data in each column
Segmented	16 TB
Partitioned, with NUMPARTS up to 64	1000 TB
Partitioned with DSSIZE, NUMPARTS up to 254	4000 TB
Partitioned with DSSIZE, NUMPARTS up to 4096	64000 TB




Recommendation: Consider defining long string columns as LOB columns when a row does not fit in a 32-KB page. Use the following guidelines to determine if a LOB column is a good choice:

- Defining a long string column as a LOB column might be better if the following factors are true:
 - Table space scans are normally run on the table.
 - The long string column is not referenced often.
 - Removing the long string column from the base table will considerably increase the performance of table space scans.
- LOBs are physically stored in another table space. Therefore, performance for inserting, updating, and retrieving long strings might be better for non-LOB strings than for LOB strings.

Assignment of table spaces to physical storage

You can store table spaces and index spaces in user-managed storage, in DB2-managed storage groups, or in SMS-managed storage. (A *storage group* is a set of disk volumes.) See the “IBM Storage Management Subsystem” sidebar for more information.

If you don’t use SMS, you need to name the DB2 storage groups when you create table spaces or index spaces. DB2 will allocate space for these objects from the named storage group. You can assign different partitions of the same table space to different storage groups. 



Recommendation: Use products in the IBM Storage Management Subsystem (SMS) family, such as Data Facility SMS (DFSMS), to manage some or all of your data sets. Organizations that use SMS to manage DB2 data sets can define storage groups with the VOLUMES(*) clause. As a result, SMS assigns a volume to the table spaces and index spaces in that storage group.



IBM Storage Management Subsystem

IBM offers the Storage Management Subsystem (SMS) family of products. A key product in the SMS family is the Data Facility Storage Management Subsystem (DFSMS). DFSMS can automatically manage all the data sets that DB2 uses and requires. If you use DFSMS to manage your data sets, the result is a reduced workload for DB2 database administrators and storage administrators.

You can experience the following benefits by using DFSMS:

- Simplified data set allocation
- Improved allocation control
- Improved performance management
- Automated disk space management
- Improved management of data availability
- Simplified data movement

DB2 database administrators can use DFSMS to achieve all their objectives for data set placement and design. To successfully use DFSMS, DB2 database administrators and storage administrators need to work together to ensure that the needs of both groups are satisfied.

Figure 7.4 shows how storage groups work together with the various DB2 data structures.

To create a DB2 storage group, use the SQL statement `CREATE STOGROUP`. This statement provides a list of volumes that DB2 can use.

After you define a storage group, DB2 stores information about it in the DB2 catalog. The catalog table `SYSIBM.SYSSTOGROUP` has a row for each storage group, and `SYSIBM.SYSVOLUMES` has a row for each volume in the group.

The process of installing DB2 includes the definition of a default storage group, `SYSDEFLT`. If you have authorization, you can define tables, indexes, table spaces, and databases. DB2 uses `SYSDEFLT` to allocate the necessary auxiliary storage. DB2 stores information about `SYSDEFLT` and all other storage groups in the catalog tables `SYSIBM.SYSSTOGROUP` and `SYSIBM.SYSVOLUMES`.

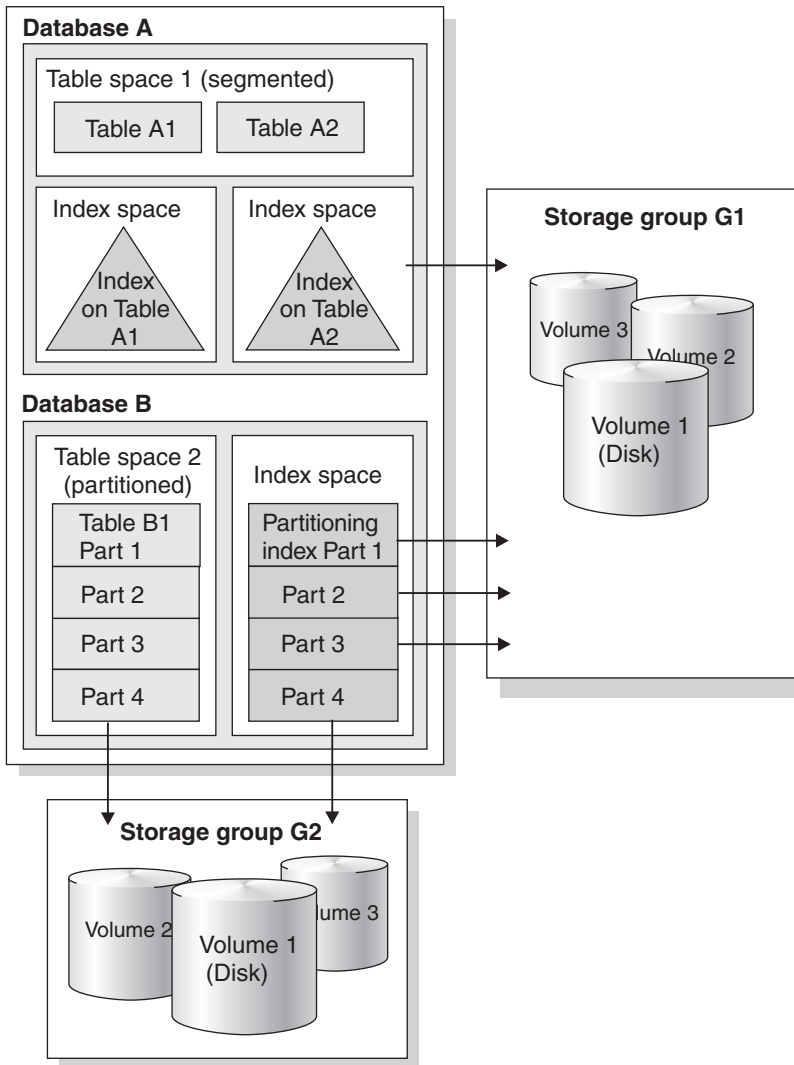



Figure 7.4
Hierarchy of DB2 structures



Recommendation: Use storage groups whenever you can, either explicitly or implicitly, by using the default storage group. In some cases, organizations need to maintain closer control over the physical storage of tables and indexes. These organizations choose to manage their own user-defined data sets rather than to use storage groups. Because this process is complex, this book does not describe the details. 

Example: Consider the following CREATE STOGROUP statement:

```
CREATE STOGROUP MYSTOGRP
  VOLUMES (*)
  VCAT ALIASICF;
```

This statement creates storage group MYSTOGRP. The * on the VOLUMES clause indicates that SMS is to manage your storage group. The VCAT clause identifies ALIASICF as the name or alias of the catalog of the integrated catalog facility that the storage group is to use. The catalog of the integrated catalog facility stores entries for all data sets that DB2 creates on behalf of a storage group.

A few examples of table space definitions

You have read about different types of table spaces. This section provides two examples of table space definitions, which use the following clauses:

IN

Identifies the database in which DB2 should create the table space.

USING STOGROUP

Indicates that you want DB2 to define and manage the data sets for this table space. If you specify the DEFINE NO clause, you can defer allocation of data sets until data is inserted or loaded into a table in the table space.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. This parameter applies only to table spaces that are using storage groups. The *integer* represents the number of kilobytes.

SECQTY *integer*

Specifies the minimum secondary space allocation for a DB2-managed data set. This parameter applies only to table spaces that are using storage groups. The *integer* represents the number of kilobytes.

Example definition for a segmented table space

The following CREATE TABLESPACE statement creates a segmented table space with 32 pages in each segment:

```
CREATE TABLESPACE MYTS
  IN MYDB
  USING STOGROUP MYSTOGRP
    PRIQTY 30720
    SECQTY 10240
  SEGSIZE 32
  LOCKSIZE TABLE
  BUFFERPOOL BP0
  CLOSE NO;
```

Example definition for an EA-enabled partitioned table space

The following CREATE TABLESPACE statement creates an EA-enabled table space, SALESHX. Assume that a large query application uses this table space to record historical sales data for marketing statistics. The first USING clause establishes the MYSTOGRP storage group and space allocations for all partitions:


```
CREATE TABLESPACE SALESHX
  IN MYDB
  USING STOGROUP MYSTOGRP
    PRIQTY 4000
    SECQTY 130
    ERASE NO
  DSSIZE 16G
  Numparts 48
    (PARTITION 46
      COMPRESS YES,
    PARTITION 47
      COMPRESS YES,
    PARTITION 48
      COMPRESS YES)
  LOCKSIZE PAGE
  BUFFERPOOL BP1
  CLOSE NO;
```

Defining indexes

You read about the main purposes of an index in “Determining what columns to index” on page 105. Indexes provide efficient access to data. When you create a table that contains a primary key, you must create a unique index for that table on the primary key. DB2 marks the table as unavailable until the explicit creation of the required indexes.

You can also choose to use indexes because of access requirements.

Be aware that using indexes involves a tradeoff. A greater number of indexes can simultaneously improve the performance of a particular transaction’s access and require additional processing for inserting, updating, and deleting index keys.

After you create an index, DB2 maintains the index, but you can perform necessary maintenance, such as reorganizing it or recovering it, as necessary. 

Index keys

All index keys do not need to be unique. For example, an index on the SALARY column of the EMP table allows duplicates because several employees can earn the same salary.

The usefulness of an index depends on its key. Columns that you use frequently in performing selection, join, grouping, and ordering operations are good key candidates.

A table can have more than one index, and an index key can use one or more columns. An *index key* is a column or an ordered collection of columns on which you define an index. A *composite key* is a key that is built on 2 to 64 columns.



Recommendation: In general, the more selective an index is, the more efficient it is. An efficient index contains multiple columns, is ordered in the same sequence as the SQL statement, and is used often in SQL statements.

The following list identifies some things you should remember when you are defining index keys.

- Column updates require index updates.
- Define as few indexes as possible on a column that is updated frequently because every change must be reflected in each index.
- A composite key might be more useful than a key on a single column when the comparison is for equality. A single multicolumn index is more efficient when the comparison is for equality and the initial columns are available.

However, for more general comparisons, such as $A > \textit{value}$ AND $B > \textit{value}$, multiple indexes might be more efficient.

- Indexes are important tools for improving performance. You can read about the use of indexes during access path selection in “Query and application performance analysis” on page 312.

Example: The following example creates a unique index the EMPPROJECT table. A composite key is defined on two columns, PROJNO and STDATE.

```
CREATE UNIQUE INDEX XPROJAC1
  ON EMPPROJECT
  (PROJNO ASC,
   STDATE ASC)
  :
```

This composite key is useful when you need to find project information by start date. Consider a SELECT statement that has the following WHERE clause:

```
WHERE PROJNO='MA2100' AND STDATE='2004-01-01'
```

This SELECT statement can execute more efficiently than if separate indexes are defined on PROJNO and on STDATE.

General index attributes

You typically determine which type of index you need to define after you define a table space. An index can have many different attributes. Index attributes fall into two broad categories: general attributes that apply to indexes on all tables and specific attributes that apply to indexes on partitioned tables only. Table 7.9 summarizes these categories.

Table 7.9 Index attributes

Table or table space type	Index attribute
Any	<ul style="list-style-type: none"> • Unique or nonunique (See “Unique indexes” on page 256 and “Nonunique indexes” on page 257.) • Clustering or nonclustering (See “Clustering indexes” on page 257.) • Padded or not padded (See “Not padded or padded indexes” on page 259.)
Partitioned	<ul style="list-style-type: none"> • Partitioning (See “Partitioning indexes” on page 260.) • Secondary (See “Secondary indexes” on page 262.)

This section explains the types of indexes that apply to all tables. Indexes that apply to partitioned tables only are covered separately.

Unique indexes

DB2 uses a unique index to ensure that data values are unique.

Example: A good candidate for a unique index is the EMPNO column of the EMP table. Figure 7.5 shows a small set of rows from the EMP table and illustrates the unique index on EMPNO.

DB2 uses this index to prevent the insertion of a row to the EMP table if its EMPNO value matches that of an existing row. The figure illustrates the relationship between each EMPNO value in the index and the corresponding page number and row. DB2 uses the index to locate the row for employee 000030, for example, in row 3 of page 1.

If you do not want duplicate values in the key column, create a unique index by using the UNIQUE clause of the CREATE INDEX statement.

Example: The DEPT table does not allow duplicate department IDs. Creating a unique index, as the following example shows, prevents duplicate values.

```
CREATE UNIQUE INDEX MYINDEX
  ON DEPT (DEPTNO) ;
```

The index name is MYINDEX, and the indexed column is DEPTNO.

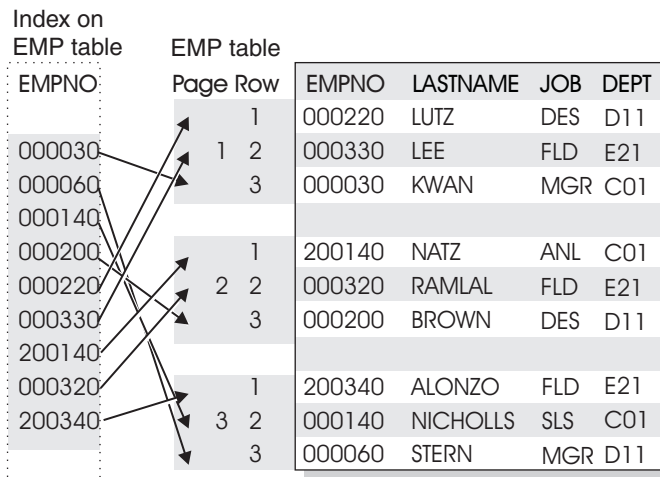


Figure 7.5

A unique index on the EMPNO column

If a table has a primary key (as the DEPT table has), its entries must be unique. You must enforce this uniqueness by defining a unique index on the primary key columns, with the index columns in the same order as the primary key columns.

Before you create a unique index on a table that already contains data, ensure that no pair of rows has the same key value. If DB2 finds a duplicate value in a set of key columns for a unique index, DB2 issues an error message and does not create the index.

Nonunique indexes

You can use nonunique indexes to improve the performance of data access when the values of the columns in the index are not necessarily unique.



Recommendation: Do not create nonunique indexes on very small tables, because scans of the tables are more efficient than using indexes.

To create nonunique indexes, use the SQL CREATE INDEX statement. For nonunique indexes, DB2 allows users and programs to enter duplicate values in a key column.

Example: Assume that more than one employee is named David Brown. Consider an index defined on the FIRSTNAME and LASTNAME columns of the EMP table.

```
CREATE INDEX EMPNAME  
  ON EMP (FIRSTNAME, LASTNAME);
```

This index is an example of a nonunique index that can contain duplicate entries.

Clustering indexes

You can define a clustering index on a partitioned table space or on a segmented table space. On a partitioned table space, a clustering index can be a partitioning index or a secondary index.

When a table has a clustering index, an INSERT statement inserts the records as nearly as possible in the order of their index values. Clustered inserts can provide significant performance advantages in some operations, particularly those that involve many records, such as grouping, ordering, and comparisons other than equal. Although a table can have several indexes, only one can be a clustering index.

If you don't define a clustering index for a table, DB2 recognizes the first index that is created on the table as the implicit clustering index when it orders data rows.



Recommendations:

- Always define a clustering index. Otherwise, DB2 might not choose the key that you would prefer for the index.
- Define the sequence of a clustering index to support high-volume processing of data.

The `CLUSTER` clause of the `CREATE INDEX` or `ALTER INDEX` statement defines a clustering index.

Example: Assume that you often need to gather employee information by department. In the `EMP` table, you can create a clustering index on the `DEPTNO` column.

```
CREATE INDEX DEPT_IX
ON EMP
(DEPTNO ASC)
CLUSTER;
```

As a result, all rows for the same department will probably be close together. DB2 can generally access all the rows for that department in a single read. (Using a clustering index does not guarantee that all rows for the same department are stored on the same page. The actual storage of rows depends on the size of the rows, the number of rows, and the amount of available freespace. Likewise, some pages may contain rows for more than one department.)

Figure 7.6 shows a clustering index on the `DEPT` column of the `EMP` table; only a subset of the rows is shown.

If a clustering index is not defined, DB2 uses the first index created on the table to order the data rows. The result might be similar to the distribution of rows in Figure 7.5.

Suppose that you subsequently create a clustering index on the same table. In this case, DB2 identifies it as the clustering index but does not rearrange the data that is already in the table. The organization of the data remains as it was with the original nonclustering index that you created. However, when the `REORG` utility reorganizes the table space, DB2 clusters the data according to the sequence of the new clustering index. Therefore, if you know that you want a clustering index, you should define the clustering index before you load the table. If that is not possible, you must define the index and then reorganize the table. If you create or drop and re-create a clustering index after loading the table, those changes take effect after a subsequent reorganization.

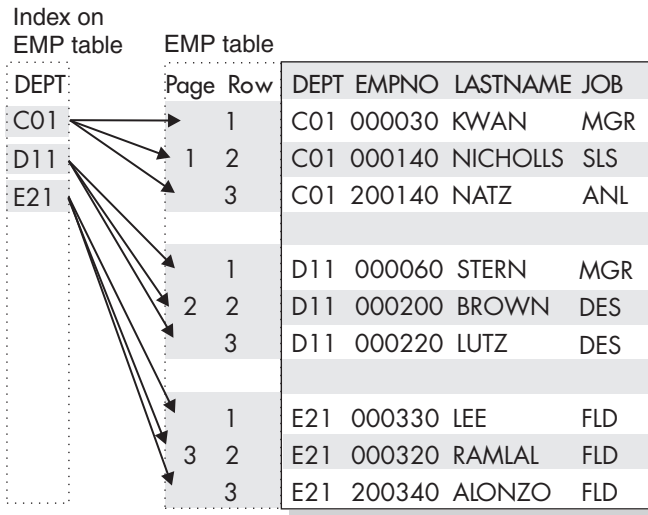


Figure 7.6
A clustering index on the EMP table

Not padded or padded indexes

The NOT PADDED and PADDED options of the CREATE INDEX and ALTER INDEX statements specify how varying-length string columns are stored in an index. You can choose not to pad varying-length string columns in the index to their maximum length (the default), or you can choose to pad them.



Recommendation: Use the NOT PADDED option to implement index-only access if your application typically accesses varying-length columns.

Partitioned table index attributes

Before Version 8, when you created a table in a partitioned table space, you defined a partitioning index and one or more secondary indexes. The partitioning index was also the clustering index, and the only partitioned index. Nonpartitioning indexes, referred to as *secondary indexes*, were not partitioned.

In Version 8, you can define the partitioning scheme of the table by using the PARTITION BY clause of the CREATE TABLE statement “Defining a table with table-controlled partitioning” on page 222 describes this method.

Version 8 introduces new features for indexes on partitioned tables:

- Indexes that are defined on a partitioned table are classified according to their logical attributes and physical attributes.
 - The logical attribute of an index on a partitioned table pertains to whether or not the index can be seen as a logically partitioning index.
 - The physical attribute of an index on a partitioned table pertains to whether or not the index is physically partitioned.
- A partitioning index can be partitioned or nonpartitioned.
- Any index can be a clustering index.
You can define only one clustering on a table.

Figure 7.7 illustrates the difference between a partitioned and a nonpartitioned index.

Based on logical index attributes, indexes on a partitioned table can be divided into two categories:

- Partitioning indexes, described in “Partitioning indexes”
- Secondary indexes, described in “Secondary indexes” on page 262

Partitioning indexes

A partitioning index is an index that defines the partitioning scheme of a table space according to the PARTITION clause for each partition in the CREATE INDEX statement.

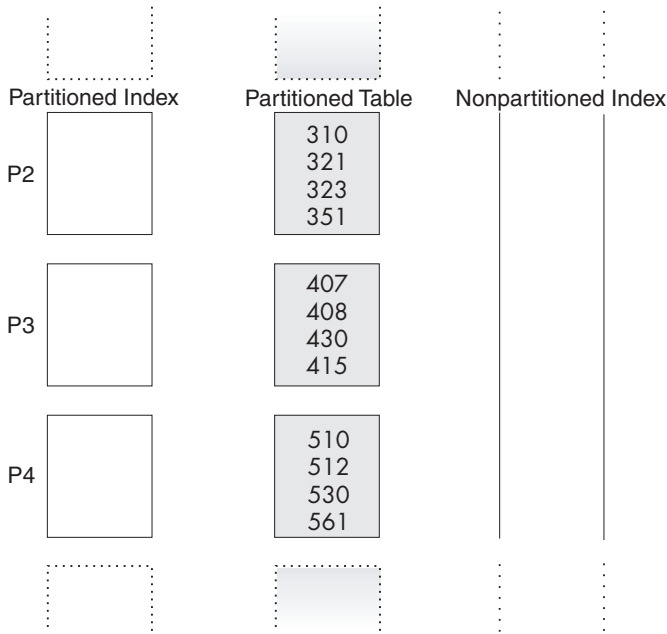


Figure 7.7
Comparison of partitioned and nonpartitioned index

The columns that you specify for the partitioning index are the key columns. The PARTITION clause for each partition defines ranges of values for the key columns. These ranges partition the table space and the corresponding partitioning index space.

Example: Partitioning index: Assume that a table contains state area codes and you need to create a partitioning index to sequence the area codes across partitions. You can use the following SQL statements to create the table and the partitioning index:

```
CREATE TABLE AREA_CODES
  (AREACODE_NO    INTEGER      NOT NULL,
   STATE         CHAR (2)     NOT NULL,
   ...
   PARTITION BY (AREACODE_NO ASC)
   ...

CREATE INDEX AREACODE_IX1 ON AREA_CODES (AREACODE_NO)
  CLUSTER (...
  PARTITION 2 ENDING AT (400),
  PARTITION 3 ENDING AT (500),
  PARTITION 4 ENDING AT (600)),
  ...);
```

Figure 7.8 illustrates the partitioning index on the AREA_CODES table.

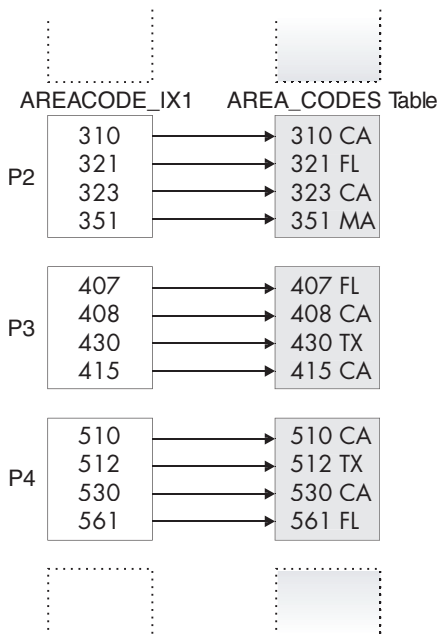


Figure 7.8
Partitioning index on the AREA_CODES table

Secondary indexes

An index that is not a partitioning index is a secondary index. The two types of secondary indexes are data-partitioned secondary indexes and nonpartitioned secondary indexes.

Data-partitioned secondary indexes

A *data-partitioned secondary index (DPSI)* is a nonpartitioning index that is physically partitioned according to the partitioning scheme of the table. Characteristics of DPSIs include:

- A DPSI has as many partitions as the number of partitions in the table space.
- Each DPSI partition contains keys for the rows of the corresponding table space partition only. For example, if the table space has three partitions, the keys in the DPSI partition 1 reference only the rows in table space partition 1; the keys in the DPSI partition 2 reference only the rows in table space partition 2, and so on.

You define a DPSI with the `PARTITIONED` keyword. If the left-most columns of the index that you specify with the `PARTITIONED` keyword coincide with the partitioning columns, DB2 does not create the index as a DPSI.

Nonpartitioned secondary indexes

A *nonpartitioned secondary index (NPSI)* is a nonpartitioning index that is nonpartitioned. A NPSI has one index space that contains keys for the rows of all partitions of the table space.

Example: Data-partitioned secondary index and nonpartitioned secondary index: This example creates a data-partitioned secondary index (DPSIIX2) and a nonpartitioned secondary index (NPSIIX3) on the `AREA_CODES` table. You can use the following SQL statements to create these secondary indexes:

```
CREATE INDEX DPSIIX2 ON AREA_CODES (STATE) PARTITIONED;
CREATE INDEX NPSIIX3 ON AREA_CODES (STATE);
```

Figure 7.9 illustrates what the data-partitioned secondary index and nonpartitioned secondary index indexes on the `AREA_CODES` table look like.

Data-partitioned secondary indexes provide advantages over nonpartitioned secondary indexes for utility processing. For example, utilities such as `COPY`, `REBUILD INDEX`, and `RECOVER INDEX` can operate on physical partitions rather than logical partitions because the keys for a given data partition reside in a single data-partitioned secondary index DPSI partition. This can provide greater availability.

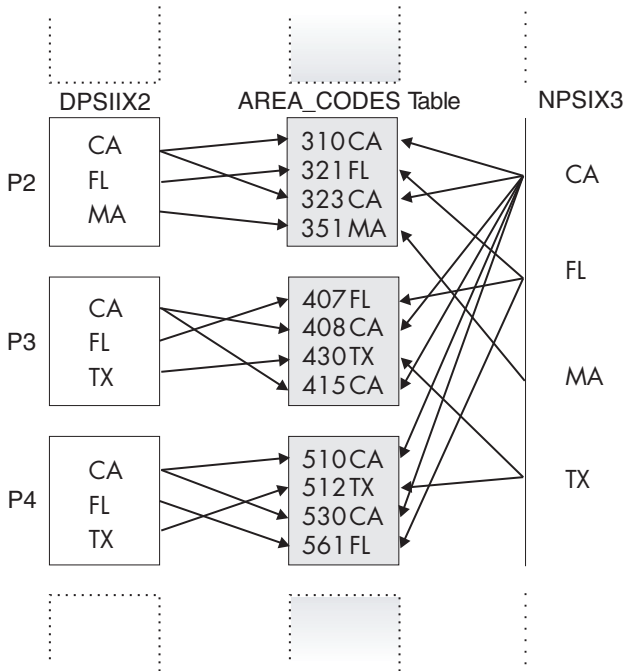


Figure 7.9

Data-partitioned secondary index and nonpartitioned secondary index on AREA_CODES table

Data-partitioned secondary indexes can also provide performance advantages for queries that meet the following criteria:

- The query has predicates on DPSI columns.
- The query contains additional predicates on the partitioning columns of the table that limit the query to a subset of the partitions in the table.

Example: Consider the following SELECT statement:

```
SELECT STATE FROM AREA_CODES
WHERE AREACODE_NO <= 300 AND STATE = 'CA';
```

This query makes efficient use of the data-partitioned secondary index. The number of key values that need to be searched is limited to the key values of the qualifying partitions. In the case of a nonpartitioned secondary index, the query searches all of the key values.

Guidelines for defining indexes

This section provides additional coding guidelines and considerations for working with indexes.

Naming the index

The name for an index is an identifier of up to 128 characters. You can qualify this name with an identifier, or schema, of up to 128 characters. The following example shows an index name:

Object	Name
Index	MYINDEX

The index space name is an eight-character name, which must be unique among names of all index spaces and table spaces in the database.

Sequencing index entries

The sequence of the index entries can be in ascending order or descending order. The ASC and DESC keywords of the CREATE INDEX statement indicate ascending and descending order. ASC is the default.

Using indexes on tables with large objects

You can use indexes on tables with LOBs the same way you use them on other tables, but consider the following factors:

- A LOB column cannot be a column in an index.
- An auxiliary table can have only one index. (An auxiliary table, which you create by using the SQL CREATE AUXILIARY TABLE statement, holds the data for a column that a base table defines. You can read more about auxiliary tables in “Defining large objects” on page 268.)
- Indexes on auxiliary tables are different than indexes on base tables.


Creating an index

If the table that you are indexing is empty, DB2 creates the index. However, DB2 does not actually create index entries until the table is loaded or rows are inserted. If the table is not empty, you can choose to have DB2 build the index when the CREATE INDEX statement is executed. Alternatively, you can defer the index build until later. Optimally, you should create the indexes on a table before loading the table. However, if your table already has data, choosing the DEFER option is preferred; you can build the index later by using the REBUILD INDEX utility.


Copying an index

If your index is fairly large and needs the benefit of high availability, consider copying it for faster recovery. Specify the `COPY YES` clause on a `CREATE INDEX` or `ALTER INDEX` statement to allow the indexes to be copied. DB2 can then track the ranges of log records to apply during recovery, after the image copy of the index is restored. (The alternative to copying the index is to use the `REBUILD INDEX` utility, which might increase the amount of time that the index is unavailable to applications.)


Deferring the allocation of index space data sets

When you execute a `CREATE INDEX` statement with the `USING STOGROUP` clause, DB2 generally defines the necessary VSAM data sets for the index space. In some cases, however, you might want to define an index without immediately allocating the data sets for the index space. 

Example: You might be installing a software program that requires creation of many indexes, but your company might not need some of those indexes. You might prefer not to allocate data sets for indexes that you will not be using.

To defer the physical allocation of DB2-managed data sets, use the `DEFINE NO` clause of the `CREATE INDEX` statement. When you specify the `DEFINE NO` clause, DB2 defines the index but defers the allocation of data sets. The DB2 catalog table contains a record of the created index and an indication that the data sets are not yet allocated. DB2 allocates the data sets for the index space as needed when rows are inserted into the table on which the index is defined. 

Defining views

When you design your database, you might need to give users access to only certain pieces of data. You can give users access by designing and using views. “Using views to customize what data a user sees” on page 104 explains the issues to consider when you design views. This section provides examples of defining views on one or more tables and the effects of modifying view information. 

Coding the view definitions

The name for a view is an identifier of up to 128 characters. The following example shows a view name:

Object	Name
View	MYVIEW

Use the `CREATE VIEW` statement to define and name a view. Unless you specifically list different column names after the view name, the column names of the view

are the same as those of the underlying table. (An example of this is in “Defining a view that combines information from several tables.”) When creating different column names for your view, remember the naming conventions that you established when designing the relational database.

As the examples in this section illustrate, a `SELECT` statement describes the information in the view. The `SELECT` statement can name other views and tables, and it can use the `WHERE`, `GROUP BY`, and `HAVING` clauses. It cannot use the `ORDER BY` clause or name a host variable.

Defining a view on a single table

Example: Assume that you want to create a view on the `DEPT` table. Of the four columns in the table, the view needs only three: `DEPTNO`, `DEPTNAME`, and `MGRNO`. The order of the columns that you specify in the `SELECT` clause is the order in which they appear in the view:

```
CREATE VIEW MYVIEW AS
  SELECT DEPTNO,DEPTNAME,MGRNO
  FROM DEPT;
```

In this example, no column list follows the view name, `MYVIEW`. Therefore, the columns of the view have the same names as those of the `DEPT` table on which it is based. You can execute the following `SELECT` statement to see the view contents:


```
SELECT * FROM MYVIEW;
```

The result table looks like this:

DEPTNO	DEPTNAME	MGRNO
=====	=====	=====
A00	CHAIRMANS OFFICE	000010
B01	PLANNING	000020
C01	INFORMATION CENTER	000030
D11	MANUFACTURING SYSTEMS	000060
E21	SOFTWARE SUPPORT	-----

Defining a view that combines information from several tables

You can create a view that contains a union of more than one table. “Merging lists of values: `UNION`” on page 150 describes how to create a union in an SQL operation.

As “Joining data from more than one table” on page 152 explains, DB2 provides two types of joins—an outer join and an inner join. An *outer join* includes rows in which the values in the join columns don’t match, and rows in which the values match. An *inner join* includes only rows in which matching values in the join columns are returned. 

Example: The following example is an inner join of columns from the DEPT and EMP tables. The WHERE clause limits the view to just those columns in which the MGRNO in the DEPT table matches the EMPNO in the EMP table:

```
CREATE VIEW MYVIEW AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
  FROM DEPT, EMP
  WHERE EMP.EMPNO = DEPT.MGRNO;
```

The result of executing this CREATE VIEW statement is an inner join view of two tables, which is shown below:

DEPTNO	MGRNO	LASTNAME	ADMRDEPT
A00	000010	HAAS	A00
B01	000020	THOMPSON	A00
C01	000030	KWAN	A00
D11	000060	STERN	D11

Example: Suppose that you want to create the view in the preceding example, but you want to include only those departments that report to department A00. Suppose also that you prefer to use a different set of column names. Use the following CREATE VIEW statement:

```
CREATE VIEW MYVIEWA00
  (DEPARTMENT, MANAGER, EMPLOYEE_NAME, REPORT_TO_NAME)
AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
  FROM EMP, DEPT
  WHERE EMP.EMPNO = DEPT.MGRNO
  AND ADMRDEPT = 'A00';
```

You can execute the following SELECT statement to see the view contents:

```
SELECT * FROM MYVIEWA00;
```

When you execute this SELECT statement, the result is a view of a subset of the same data, but with different column names, as follows:

DEPARTMENT	MANAGER	EMPLOYEE_NAME	REPORT_TO_NAME
A00	000010	HAAS	A00
B01	000020	THOMPSON	A00
C01	000030	KWAN	A00

Inserting and updating data through views

If you define a view on a single table, you can refer to the name of a view in an INSERT, UPDATE, or DELETE statement. This section explains how DB2 makes an insert or update to the base table.

To ensure that the insert or update conforms to the view definition, specify the WITH CHECK OPTION clause. The following example illustrates some undesirable results of omitting that check.

Example: Suppose that you define a view, V1, as follows:

```
CREATE VIEW V1 AS
  SELECT * FROM EMP
  WHERE DEPT LIKE 'D%';
```

A user with the SELECT privilege on view V1 can see the information from the EMP table for employees in departments whose IDs begin with D. The EMP table has only one department (D11) with an ID that satisfies the condition.

Assume that a user has the INSERT privilege on view V1. A user with both SELECT and INSERT privileges can insert a row for department E01, perhaps erroneously, but cannot select the row that was just inserted.


The following example shows an alternative way to define view V1.

Example: You can avoid the situation in which a value that does not match the view definition is inserted into the base table. To do this, instead define view V1 to include the WITH CHECK OPTION clause:

```
CREATE VIEW V1 AS SELECT * FROM EMP
  WHERE DEPT LIKE 'D%' WITH CHECK OPTION;
```

With the new definition, any insert or update to view V1 must satisfy the predicate that is contained in the WHERE clause: DEPT LIKE 'D%'. The check can be valuable, but it also carries a processing cost; each potential insert or update must be checked against the view definition. Therefore, you must weigh the advantage of protecting data integrity against the disadvantage of performance degradation.

Defining large objects

Defining large objects to DB2 is different than defining other types of data and objects. This section explains the basic steps that you can take to define LOB data to DB2 and to create large objects. 

These are the basic steps for defining LOBs and moving the data into DB2:

1. Define a column of the appropriate LOB type.

When you create a table with a LOB column or alter a table to add a LOB column, defining a ROWID column is optional. If you do not define a ROWID column, DB2 defines a hidden ROWID column for you. Define only one ROWID column, even if multiple LOB columns are in the table.

The LOB column holds information about the LOB, not the LOB data itself. The table that contains the LOB information is called the *base table*. (This is a special kind of base table, different than the one that “Types of tables” on page 218 describes.) DB2 uses the ROWID column to locate LOB data. You can define the LOB column and the ROWID column in a CREATE TABLE or ALTER TABLE statement. If you are adding a LOB column and a ROWID column to an existing table, you must use two ALTER TABLE statements. If you add the ROWID after you add the LOB column, the table has two ROWIDs: a hidden one and the one that you created. DB2 ensures that the values of the two ROWIDs are always the same.

2. Create a table space and table to hold the LOB data.

For LOB data, the table space is called a LOB table space, and a table is called an auxiliary table. If your base table is nonpartitioned, you must create one LOB table space and one auxiliary table for each LOB column. If your base table is partitioned, you must create one LOB table space and one auxiliary table for each LOB column in each partition. For example, you must create three LOB table spaces and three auxiliary tables for each LOB column if your base table has three partitions. Create these objects by using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.

3. Create an index on the auxiliary table.

Each auxiliary table must have exactly one index in which each index entry refers to a LOB. Use the CREATE INDEX statement for this task.

4. Put the LOB data into DB2.

If the total length of a LOB column and the base table row is less than 32 KB, you can use the LOAD utility to put the data in DB2. Otherwise, you must use INSERT or UPDATE statements. Even though the data resides in the auxiliary table, the LOAD utility statement or INSERT statement specifies the base table. Using INSERT can be difficult because your application needs enough storage to hold the entire value that goes into the LOB column.

Example: Assume that you need to define a LOB table space and an auxiliary table to hold employee resumes. You also need to define an index on the auxiliary table. You must define the LOB table space in the same database as the associated base table. Assume that EMP_PHOTO_RESUME is a base table. This base table has a LOB column named EMP_RESUME. You can use statements like this to define the LOB table space, the auxiliary table space, and the index:

```
CREATE LOB TABLESPACE RESUMETS
  IN MYDB
  LOG NO;
COMMIT;
CREATE AUXILIARY TABLE EMP_RESUME_TAB
  IN MYDB.RESUMETS
  STORES EMP_PHOTO_RESUME
  COLUMN EMP_RESUME;
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME_TAB;
COMMIT;
```


You can use the LOG clause to specify whether changes to a LOB column in the table space are to be logged. The LOG NO clause in the preceding CREATE LOB TABLESPACE statement indicates that changes to the RESUMETS table space are not to be logged.

You can use the DB2 UDB Extenders tools with large object data. See the “DB2 UDB Extenders” sidebar.



DB2 UDB Extenders

You can use the DB2 UDB Extenders feature of DB2 UDB for z/OS to store and manipulate image, audio, video, and text objects. The extenders automatically capture and maintain object information and provide a rich body of APIs.

The DB2 UDB Extenders that support large objects comprise a separate Image Extender, Audio Extender, Video Extender, and Text Extender. Each extender defines a distinct type and a set of user-defined functions for use with objects of its distinct type. The extenders automatically capture and maintain a variety of attribute information about each object that you store. 

Defining databases

When you define a DB2 database, you name an eventual collection of tables and associated indexes, as well as the table spaces in which they are to reside. When you decide whether to define a new database for a new set of objects or use an existing database, consider the following factors:

- You can start and stop an entire database as a unit. You can display the status of all objects in the database by using a single command that names only the database. Therefore, place a set of related tables into the same database. (The same database holds all indexes on those tables.)
- If you want to improve concurrency and memory use, keep the number of tables in a single database relatively small (maximum of 20 tables). For example, with fewer tables, DB2 performs a reorganization in a shorter length of time.
- Having separate databases allows data definitions to run concurrently and uses less space for control blocks as well.

To create a database, use the `CREATE DATABASE` statement. A name for a database is an unqualified identifier of up to eight characters. A DB2 database name must not be the same as the name of any other DB2 database.


The following example shows a valid database name:

Object	Name
Database	MYDB

Example: This `CREATE DATABASE` statement creates the database MYDB:

```
CREATE DATABASE MYDB
  STOGROUP MYSTOGRP
  BUFFERPOOL BP8K4
  INDEXBP BP4 ;
```


The `STOGROUP`, `BUFFERPOOL`, and `INDEXBP` clauses that this example shows establish default values. You can override these values on the definitions of the table space or index space.

You do not need to define a database to use DB2; for development and testing, you can use the default database, `DSNDB04`. This means that you can define tables and indexes without specifically defining a database. The catalog table `SYSIBM.SYS-DATABASE` describes the default database and all other databases. 



Recommendation: Do not use the default database for production work.

Defining relationships with referential constraints

“Referential integrity and referential constraints” on page 54 introduces referential integrity. Referential integrity is a condition in which all intended references from data in one table column to data in another table column are valid. By using referential constraints, you can define relationships between entities that you define in DB2. 

Organizations that choose to enforce referential constraints have at least one thing in common. They need to ensure that values in one column of a table are valid with respect to other data values in the database.

Examples:

- A manufacturing company wants to ensure that each part in a PARTS table identifies a product number that equals a valid product number in the PRODUCTS table. (“Appendix A. Example tables in this book” shows the example PARTS and PRODUCTS tables.)
- A company wants to ensure that each value of DEPT in the EMP table equals a valid DEPTNO value in the DEPT table.

If the DBMS did not support referential integrity, then programmers would need to write and maintain application code that validates the relationship between the columns, and some programs might not enforce business rules, even though they should.


This programming task can be complex because of the need to make sure that only valid values are inserted or updated in the columns. When the DBMS supports referential integrity, as DB2 does, programmers avoid some complex programming tasks and can be more productive in their other work.

This section provides guidelines for establishing a referential structure when you create objects.

How DB2 enforces referential constraints

You define referential constraints between a foreign key and its parent key. Before you start to define the referential relationships and constraints, you should understand what DB2 does to maintain referential integrity. You should understand the rules that DB2 follows when users attempt to modify information in columns that are involved in referential constraints.

To maintain referential integrity, DB2 enforces referential constraints in response to any of the following events:

- An insert to a dependent table
- An update to a parent table or dependent table
- A delete from a parent table
- Running the CHECK DATA utility or the LOAD utility on a dependent table with the ENFORCE CONSTRAINTS option 

When you define the constraints, you have the following choices:

CASCADE

DB2 propagates the action to the dependents of the parent table.

NO ACTION


An error occurs, and DB2 takes no action.


RESTRICT

An error occurs, and DB2 takes no action.

SET NULL

DB2 places a null value in each nullable column of the foreign key that is in each dependent of the parent table.

DB2 does not enforce referential constraints in a predefined order. However, the order in which DB2 enforces constraints can affect the result of the operation. Therefore, you should be aware of the restrictions on the definition of delete rules and on the use of certain statements. The restrictions relate to the following SQL statements: CREATE TABLE, ALTER TABLE, INSERT, UPDATE, and DELETE. 

You read about another type of constraint, an informational referential constraint, in “Referential integrity and referential constraints” on page 54. You can use the NOT ENFORCED option of the referential constraint definition in a CREATE TABLE or ALTER TABLE statement to define an informational referential constraint. You should use this type of referential constraint only when an application process verifies the data in a referential integrity relationship. 

Insert rules

The following insert rules for referential integrity apply to parent and dependent tables:

- **For parent tables:** You can insert a row at any time into a parent table without taking any action in the dependent table. For example, you can create a new department in the DEPT table without making any change to the EMP table. If you are inserting rows into a parent table that is involved in a referential constraint, the following restrictions apply:
 - A unique index must exist on the parent key.
 - You cannot enter duplicate values for the parent key.
 - You cannot insert a null value for any column of the parent key.
- **For dependent tables:** You cannot insert a row into a dependent table unless a row in the parent table has a parent key value that equals the foreign key value that you want to insert. You can insert a foreign key with a null value into a dependent table (if the referential constraint allows this), but no logical connection exists if you do so. If you insert rows into a dependent table, the following restrictions apply:
 - Each nonnull value that you insert into a foreign key column must be equal to some value in the parent key.
 - If any field in the foreign key is null, the entire foreign key is null.
 - If you drop the index that enforces the parent key of the parent table, you cannot insert rows into either the parent table or the dependent table.

Example: Your company doesn't want to have a row in the PARTS table unless the PROD# column value in that row matches a valid PROD# in the PRODUCTS table. The PRODUCTS table has a primary key on PROD#. The PARTS table has a foreign key on PROD#. The constraint definition specifies a RESTRICT constraint. Every inserted row of the PARTS table must have a PROD# that matches a PROD# in the PRODUCTS table.

Update rules

The following update rules for referential integrity apply to parent and dependent tables:

- **For parent tables:** You cannot change a parent key column of a row that has a dependent row. If you do, the dependent row no longer satisfies the referential constraint, so DB2 prohibits the operation.
- **For dependent tables:** You cannot change the value of a foreign key column in a dependent table unless the new value exists in the parent key of the parent table.

Example: When an employee transfers from one department to another, the department number for that employee must change. The new value must be the number of an existing department, or it must be null. You should not be able to assign an employee to a department that does not exist. However, in the event of a company reorganization, employees might temporarily not report to a valid department. In this case, a null value would be a possibility.

If an update to a table with a referential constraint fails, DB2 rolls back all changes that were made during the update.

Delete rules

The following delete rules for referential integrity apply to parent and dependent tables:

- **For parent tables:** For any particular relationship, DB2 enforces delete rules according to the choices that you specify when you define the referential constraint. See “How DB2 enforces referential constraints” on page 272 for descriptions of the choices that you have.
- **For dependent tables:** At any time, you can delete rows from a dependent table without taking any action on the parent table.

Example: Consider the parent table in the department-employee relationship. Suppose that you delete the row for department C01 from the DEPT table. That deletion should affect the information in the EMP table about Sally Kwan, Heather Nicholls, and Kim Natz, who work in department C01.

Example: Consider the dependent in the department-employee relationship. Assume that an employee retires and that a program deletes the row for that employee from the EMP table. The DEPT table is not affected.

To delete a row from a table that has a parent key and dependent tables, you must obey the delete rules for that table. To succeed, the DELETE must satisfy all delete rules of all affected relationships. The DELETE fails if it violates any referential constraint.

Building a referential structure

When you build a referential structure, you need to create a set of tables and indexes in the correct order. “Defining entities for different types of relationships” on page 88 explains the different kinds of relationships. During logical design, you

express one-to-one relationships and one-to-many relationships as if the relationships are bidirectional. For example:

- An employee has a resume, and a resume belongs to an employee (one-to-one relationship).
- A department has many employees, and each employee reports to a department (one-to-many relationship).

During physical design, you restate the relationship so that it is unidirectional; one entity becomes an implied parent of the other. In this case, the employee is the parent of the resume, and the department is the parent of the assigned employees.


During logical design, you express many-to-many relationships as if the relationships are both bidirectional and multivalued. During physical design, database designers resolve many-to-many relationships by using an associative table (described in “Denormalizing tables to improve performance” on page 102). The relationship between employees and projects is a good example of how referential integrity is built. This is a many-to-many relationship because employees work on more than one project and a project can have more than one employee assigned.

Example: To resolve the many-to-many relationship between employees (in the EMP table) and projects (in the PROJ table), designers create a new associative table, EMP_PROJ, during physical design. EMP and PROJ are both parent tables to the child table, EMP_PROJ.

When you establish referential constraints, you must create parent tables with their primary keys and corresponding indexes before you can define any corresponding foreign keys on dependent tables.

Defining the tables in the referential structure

You can use the following procedure as a model to create a referential structure. This procedure uses the DEPT and EMP tables.

You can create table spaces in any order. However, you need to create the table spaces before you perform the following steps. 

1. Create the DEPT table and define its primary key on the DEPTNO column. The PRIMARY KEY clause of the CREATE TABLE statement defines the primary key.

Example:

```
CREATE TABLE DEPT
:
:
PRIMARY KEY (DEPTNO);
```

2. Create the primary index for the DEPT table.

Example:

```
CREATE UNIQUE INDEX DEPT
ON DEPT (DEPTNO);
```

3. Create the EMP table and define its primary key as EMPNO and its foreign key as DEPT. The FOREIGN KEY clause of the CREATE TABLE statement defines the foreign key.

Example:


```
CREATE TABLE EMP
:
:
PRIMARY KEY (EMPNO)
FOREIGN KEY (DEPT)
REFERENCES DEPT (DEPTNO)
ON DELETE SET NULL;
```

4. Alter the DEPT table to add the definition of its foreign key, DEPT.

Example:

```
ALTER TABLE DEPT
FOREIGN KEY (DEPTNO)
REFERENCES EMP (DEPT)
ON DELETE RESTRICT;
```

Loading the tables

Before you load tables that are involved in a referential constraint or check constraint, you need to create exception tables. An *exception table* contains the rows found by the CHECK DATA utility that violate referential constraints or check constraints. 

Defining other business rules

DB2 provides two additional mechanisms that you can use to enforce your business rules—triggers and user-defined functions.

Defining triggers

You read about triggers in “Triggers” on page 57.

Triggers automatically execute a set of SQL statements whenever a specified event occurs. These statements validate and edit database changes, read and modify the database, and invoke functions that perform operations inside and outside the database. A trigger is a powerful mechanism. You can use triggers to define and enforce business rules that involve different states of the data.

Example: Assume that the majority of your organization’s salary increases are less than or equal to 10 percent. Assume also that you need to receive notification of any attempts to increase a value in the salary column by more than that amount. To enforce this requirement, DB2 compares the value of a salary before a salary increase to the value that would exist after a salary increase. You can use a trigger in this case. Whenever a program updates the salary column, DB2 activates the trigger. In the triggered action, you can specify that DB2 is to perform the following actions:

- Update the value in the salary column with a valid value, rather than preventing the update altogether.
- Notify an administrator of the attempt to make an invalid update.

As a result of using a trigger, the notified administrator can decide whether to override the original salary increase and allow a larger-than-normal salary increase.




Recommendation: For rules that involve only one condition of the data, consider using referential constraints and check constraints rather than triggers.

Triggers also move the application logic that is required to enforce business rules into the database, which can result in faster application development and easier maintenance. In the previous example, which limits salary increases, the logic is in the database, rather than in an application. DB2 checks the validity of the changes that any application makes to the salary column. In addition, if the logic ever changes (for example, to allow 12 percent increases), you don’t need to change the application programs.

Triggers are optional. You define triggers by using the CREATE TRIGGER statement. 

Defining user-defined functions

You read about user-defined functions in “Using user-defined functions” on page 135.

User-defined functions can be sourced, external, or SQL functions. *Sourced* means that they are based on existing functions. *External* means that users develop them. SQL means that the function is defined to the database by only SQL statements. 

External user-defined functions can return a single value or a table of values.

- External functions that return a single value are called *user-defined scalar functions*.
- External functions that return a table are called *user-defined table functions*.

User-defined functions, like built-in functions or operators, support the manipulation of distinct types. “Defining and using distinct types” on page 232 introduces distinct types.

The following two examples demonstrate how to define and use both a user-defined function and a distinct type.

Example: Suppose that you define a table called EUROEMP. One column of this table, EUROSAL, has a distinct type of EURO, which is based on DECIMAL(9,2). You cannot use the built-in AVG function to find the average value of EUROSAL because AVG operates on built-in data types only. You can, however, define an AVG function that is sourced on the built-in AVG function and accepts arguments of type EURO:

```
CREATE FUNCTION AVG (EURO)
  RETURNS EURO
  SOURCE SYSIBM.AVG (DECIMAL) ;
```

Example: You can then use this function to find the average value of the EUROSAL column:

```
SELECT AVG (EUROSAL) FROM EUROEMP ;
```

The next two examples demonstrate how to define and use an external user-defined function.

Example: Suppose that you define and write a function, called REVERSE, to reverse the characters in a string. The definition looks like this:

```
CREATE FUNCTION REVERSE (CHAR)
  RETURNS CHAR
  EXTERNAL NAME 'REVERSE'
  PARAMETER STYLE DB2SQL
  LANGUAGE C ;
```

Example: You can then use the REVERSE function in an SQL statement wherever you would use any built-in function that accepts a character argument. For example:

```
SELECT REVERSE (: CHARSTR)
FROM SYSDUMMY1 ;
```

Although you cannot write user-defined aggregate functions, you can define sourced user-defined aggregate functions that are based on built-in aggregate functions. This capability is useful in cases where you want to refer to an existing user-defined function by another name or to pass in a distinct type.

The next two examples demonstrate how to define and use a user-defined table function.

Example: You can define and write a user-defined table function that users can invoke in the FROM clause of a SELECT statement. For example, suppose that you define and write a function called BOOKS. This function returns a table of information about books on a given subject. The definition looks like this:

```
CREATE FUNCTION BOOKS                (VARCHAR(40))
  RETURNS TABLE (TITLE_NAME        VARCHAR(25),
                 AUTHOR_NAME       VARCHAR(25),
                 PUBLISHER_NAME    VARCHAR(25),
                 ISBNNO            VARCHAR(20),
                 PRICE_AMT         DECIMAL(5,2),
                 CHAP1_TXT         CLOB(50K))
LANGUAGE COBOL
PARAMETER STYLE DB2SQL
EXTERNAL NAME BOOKS ;
```

Example: You can then include the BOOKS function in the FROM clause of a SELECT statement to retrieve the book information. For example:

```
SELECT B.TITLE_NAME, B.AUTHOR_NAME, B.PUBLISHER_NAME,
B.ISBNNO
FROM TABLE(BOOKS('Computers')) AS B
WHERE B.TITLE_NAME LIKE '%COBOL%' ;
```

For more information

Table 7.10 lists additional information sources about topics that this chapter introduces.

Table 7.10 More information about topics in Chapter 7

For more information about...	Introduced in section that begins on page...	See...
Check constraints	237	Volume 1 of <i>DB2 Application Programming and SQL Guide</i>
CREATE SQL statements for defining objects: Databases, indexes, storage groups, tables, table spaces, views	217	Volume 2 of <i>DB2 SQL Reference</i>
Data sets	246	<ul style="list-style-type: none"> • Volume 1 of <i>DB2 Administration Guide</i> • <i>z/OS DFSMS: Using Data Sets</i>
DB2 catalog tables	218	Volume 2 of <i>DB2 SQL Reference</i>
DB2 data types	224	Volume 1 of <i>DB2 SQL Reference</i>
DB2 Extenders	268	<ul style="list-style-type: none"> • <i>DB2 Image, Audio, and Video Extender Administration and Programming</i> • <i>DB2 Net Search Extender Administration and Programming</i> • <i>DB2 Text Extender Administration and Programming</i> • www.ibm.com/software/data/db2/extenders
DB2 utilities	244	<i>DB2 Utility Guide and Reference</i>
Distinct types	232	Volume 1 of <i>DB2 SQL Reference</i>
EA-enabled table spaces	247	Volume 1 of <i>DB2 Administration Guide</i>
ENFORCE CONSTRAINTS option on DB2 utilities	272	<i>DB2 Utility Guide and Reference</i>

Table 7.10 More information about topics in Chapter 7 (Continued)

For more information about...	Introduced in section that begins on page...	See...
IBM Storage Management Subsystem (SMS)	249	<i>Storage Management with DB2 for OS/390</i> (viewable at www.redbooks.ibm.com)
Index space data sets, deferring allocation of	265	Volume 2 of <i>DB2 SQL Reference</i>
Informational referential constraints	272	Volume 1 of <i>DB2 Application Programming and SQL Guide</i>
Joining tables	266	Volume 1 of <i>DB2 SQL Reference</i>
Large objects	268	<i>Large Objects with DB2 for z/OS and OS/390</i> (viewable at www.redbooks.ibm.com)
Loading tables that are involved in referential relationships	277	<i>DB2 Utility Guide and Reference</i>
Materialized query tables	222	Volume 1 of <i>DB2 Administration Guide</i>
Naming conventions	216	Volume 1 of <i>DB2 SQL Reference</i>
Partitioned table spaces	245	Volume 2 of <i>DB2 SQL Reference</i>
Referential integrity and constraints	272	<ul style="list-style-type: none"> • Volume 1 of <i>DB2 Application Programming and SQL Guide</i> • Volume 1 of <i>DB2 SQL Reference</i>
Referential integrity, restrictions on SQL statements related to	272	Volume 2 of <i>DB2 SQL Reference</i>
Referential structures, defining	276	Volume 2 of <i>DB2 SQL Reference</i>
Relational concepts as implemented in other DB2 products	216	<ul style="list-style-type: none"> • <i>A Complete Guide to DB2 Universal Database</i> by Don Chamberlin • <i>DB2 Universal Database for iSeries Database Programming</i> • <i>DB2 Universal Database for Linux, UNIX, and Windows Administration Guide: Implementation</i> • <i>DB2 for VSE & VM Database Administration</i>

Table 7.10 More information about topics in Chapter 7 (Continued)

For more information about...	Introduced in section that begins on page...	See...
Rows, conditions for deleting	244	<ul style="list-style-type: none"> • Volume 2 of <i>DB2 Administration Guide</i> • Volume 2 of <i>DB2 SQL Reference</i>
Segment size	243	Volume 2 of <i>DB2 Administration Guide</i>
Storage groups, managing	249	Volume 1 of <i>DB2 Administration Guide</i>
Tables, differences among	218	Volume 1 of <i>DB2 Administration Guide</i>
Triggers, defining	278	Volume 1 of <i>DB2 Application Programming and SQL Guide</i>
Unicode in DB2	226	Volume 1 of <i>DB2 SQL Reference</i>
User-defined default values, restrictions	235	Volume 1 of <i>DB2 SQL Reference</i>
User-defined functions, defining	278	Volume 1 of <i>DB2 Application Programming and SQL Guide</i>
User-defined functions, samples that come with DB2	229	Volume 1 of <i>DB2 Application Programming and SQL Guide</i>
VSAM	265	<i>DFSMS/MVS: Access Method Services for the Integrated Catalog</i>

Practice exam questions

The following practice exam questions test your knowledge of material that this chapter covers.

1. Which statement is false?

- A. You can put data into a table that you create by using the DB2 LOAD utility or SQL INSERT statement.
- B. When you define a base table, you define all the columns of the table.
- C. When you create a base table, DB2 records the table's definition in the DB2 directory.
- D. Creating a table does not store the application data.

2. Which statement is valid?

- A.

```
CREATE TABLE EMP
  (EMPNO CHAR(6) NOT NULL,
   FIRSTNME VAR CHAR(12) NOT NULL,
   LASTNAME VAR CHAR(15) NOT NULL,
   PRIMARY KEY (EMPNO))
IN MYDB.MYTS;
```
- B.

```
CREATE TABLE EMP
  (EMPNO CHAR(6) NOT NULL,
   FIRSTNME VAR CHAR(12) NOT NULL,
   LASTNAME VAR CHAR(15) NOT NULL,
   PRIMARY KEY (EMPNO))
IN MYDB.MYTS
```
- C.

```
CREATE TABLE EMP
  (EMPNO CHAR(6) NOT NULL,
   FIRSTNME VARCHAR(12) NOT NULL,
   LASTNAME VARCHAR(15) NOT NULL,
   PRIMARY KEY (EMPNO))
IN MYDB.MYTS;
```
- D.

```
CREATE TABLE NAME EMP
  (EMPNO CHAR(6) NOT NULL,
   FIRSTNME VARCHAR(12) NOT NULL,
   LASTNAME VARCHAR(15) NOT NULL,
   PRIMARY KEY (EMPNO))
IN MYDB.MYTS;
```

3. For a column that will hold the last names of people, what data type would work best?
- A. CHAR
 - B. VARCHAR
 - C. DBCLOB
 - D. ROWID
4. If a check constraint specifies that a part cost cannot exceed the cost of the product that contains it, which statement results in a violation of the check constraint?
- A. INSERT INTO PRODPART
(PARTNUM, PRODNUM, PARTCOST, PRODCOST)
VALUES ('1256', 'F1109', '42.99', '45.99');
 - B. INSERT INTO PRODPART
(PARTNUM, PRODNUM, PARTCOST, PRODCOST)
VALUES ('1256', 'F1109', '45.99', '42.99');
 - C. INSERT INTO PRODPART
(PARTNUM, PRODNUM, PARTCOST, PRODCOST)
VALUES ('1256', 'F1109', '4.29', '45.99');
 - D. INSERT INTO PRODPART
(PARTNUM, PRODNUM, PARTCOST, PRODCOST)
VALUES ('1256', 'F1109', '45.99', '459.99');
5. Which scenario is not well suited for referential integrity?
- A. A manufacturing company wants to make sure that each part in a PARTS table identifies a product number that equals a valid product number in the PRODUCTS table.
 - B. A company wants to ensure that each value of DEPT in the EMP table equals a valid DEPTNO value in the DEPT table.
 - C. A payroll manager wants to make sure that no user or application program can increase the value of the SALARY column by more than 10%.
 - D. A human resources manager wants to ensure that every value of MGRNAME in the DEPT table equals a valid EMPNO value in the EMP table.

6. Which statement is false?

- A. Indexes provide efficient access to data.
- B. When you create a table that contains a primary key, you must create a unique index for that table on the primary key.
- C. Each index that you define on a table must include the column or columns that comprise the primary key.
- D. After you create the index, DB2 maintains it.

7. Which statement is false?

- A. If you attempt to create an index on a table that has no data, DB2 does not create the index.
- B. You can choose to have DB2 build an index when the CREATE INDEX statement is executed.
- C. You can choose to have DB2 defer the building of the index until later.
- D. Optimally, you should create the indexes on a table before loading the table.

Answers to practice exam questions

1. **Answer: C.** The statements in the other options are true. Option C is false because DB2 records the definition of a base table in the DB2 catalog, not in the DB2 directory.
2. **Answer: C.** Option A is not valid because it is missing a closing parenthesis after (EMPNO). Both option A and option B are not valid because VARCHAR is spelled incorrectly as VAR CHAR. Option D is not valid because the CREATE TABLE syntax includes “NAME,” which is not a valid keyword.
3. **Answer: B.** VARCHAR is the best data type of people’s last names because they are character data of varying lengths.
4. **Answer: B.** Option B is the only INSERT statement that has a part cost (\$45.99) that exceeds the product cost (\$42.99).
5. **Answer: C.** Options A, B, and D all identify scenarios for which referential integrity is a good solution. The scenario in option C could use a trigger to enforce this business rule.
6. **Answer: C.** The statements in the other options are true. Option C is false because DB2 does not require that each index in a table must include the columns that comprise the primary key. Many indexes do not contain the primary key columns.
7. **Answer: A.** Options B, C, and D are true. Option A is false because in this case, DB2 creates the index but does not create actual index entries until the table is loaded or rows are inserted into the table.