Chapter 4

Designing objects and relationships

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.

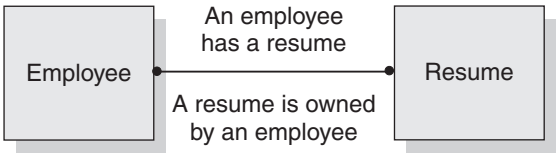

Figure 4.1
Assigning one-to-one facts to an entity

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

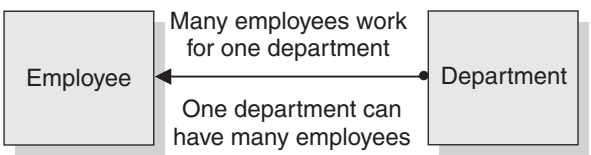

Figure 4.2
Assigning many-to-one facts to an entity

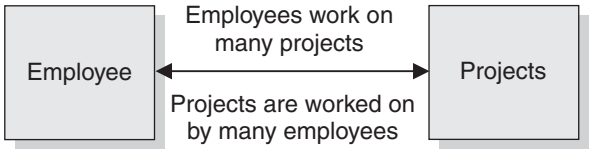

Figure 4.3
Assigning many-to-many facts to an entity

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 DECMAL(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.

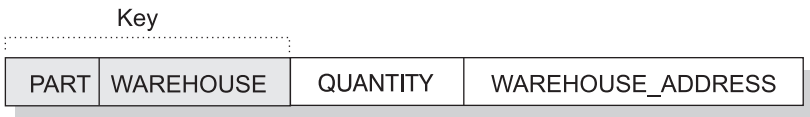

Figure 4.5
A primary key that violates second normal form

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

EMPLOYEE_NUMBER	EMPLOYEE_FIRST_NAME	EMPLOYEE_LAST_NAME
000200	DAVID	BROWN
000329	RAMLAL	MEHTA
000220	JENNIFER	LUTZ

Department table

Key

DEPARTMENT_NUMBER	DEPARTMENT_NAME
D11	MANUFACTURING SYSTEMS
E21	SOFTWARE SUPPORT

Employee_Department table

Key

DEPARTMENT_NUMBER	EMPLOYEE_NUMBER
D11	000200
D11	000220
E21	000329

Figure 4.8
Normalized entities: EMPLOYEE, DEPARTMENT, and EMPLOYEE_DEPARTMENT

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.

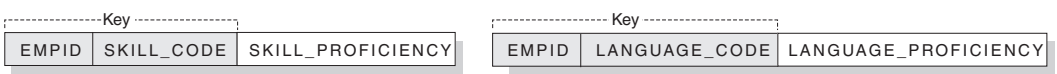

Figure 4.10
Entities that are in fourth normal form

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 subsecond 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?

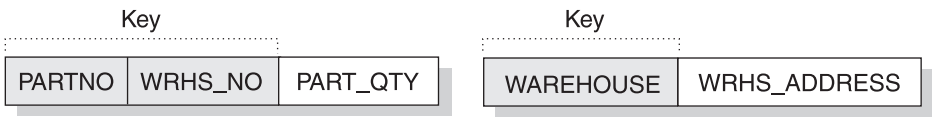

Figure 4.11
Two tables that satisfy second normal form

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).

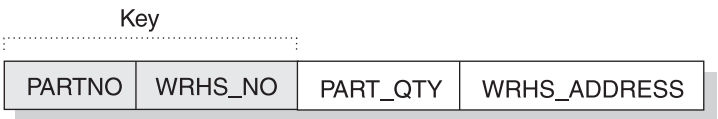

Figure 4.12
Denormalized table

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	• *Handbook of Relational Database Design* by Candace C. Fleming and Barbara von Halle • *Data Modeling Essentials: Analysis, Design, and Innovation* by Graeme C. Simsion • *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models* by Michael Reingruber and William Gregory • *Introduction to Information Engineering: From Strategic Planning to Information Systems* by Clive Finkelstein • *DB2 for z/OS and OS/390 Development for Performance Volume I, DB2 for z/OS and OS/390 Development for Performance Volume II* by Gabrielle Wiorkowski • *DB2 Developer's Guide* by Craig Mullins • *DB2 High Performance Design and Tuning* by Susan Lawson, Richard Yevich, and Warwick Ford • *DB2 Universal Database for Linux, UNIX, and Windows Administration Guide: Planning*

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	• *The Capability Maturity Model: Guidelines for Improving the Software Process* by Software Engineering Institute, Carnegie Mellon University • *Managing the Software Process* by Watts S. Humphrey
UML modeling	99	*Database Design for Smarties: Using UML for Data Modeling* 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.