CPE-307 RDBMS USING PL/SQL

L	T	P	Cr
3	1	0	3.5

Section A

Introduction of DBMS:
DBMS architecture, Enhanced-ER (EER) Model Concepts: Specialization and Generalization, Union type, Constraints on Specialization and Generalization, Concept of Hierarchy and Lattice, EER-to-Relational Mapping.

Distributed Databases and Client-Server Architecture: Introduction to Distributed DBMS Concepts, Client-Server Architecture Overview, Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design, Types of Distributed Database Systems.

PL/SQL: Block Structure, Data Types, Creation of Variable, Scope, Nested Blocks, Control Structures. Records and Collections. Using SQL with PL/SQL: Cursors and its types. Subprograms: Stored and Local Procedures and Functions, Procedure vs Function.

Section B

Database Security: Types of Security, Control Measures, DB security and DBA, Access protection, Discretionary Access Control based on Granting and Revoking privileges. User Creation and Management in SQL: Creating a user, Assigning and Removing User Privileges, Creating and Assigning Roles.

Transaction processing: Introduction, Concurrency, Problems due to concurrency, ACID Properties, Schedule, Serializability. Serial, Non-serial and Conflict-Serializable Schedule

Concurrency control: Locks, Types of Locks: Binary and Two Phase Locking, Variations of Two Phase Locking. Deadlock: Deadlock Prevention Techniques, Deadlock Detection and Recovery. Database Recovery Concepts.

Packages: Specification and Body, Triggers and its types. Introduction to Objects: Creating, Storing and Manipulating Objects.

Recommended Books :

1. Navathe and Elmasri, Fundamentals of Database Systems, Pearson education
2. Korth and Silberschatz Abraham, Database Concepts, McGraw Hall, 1991.
3. An introduction to database system by C.J.Date (Addison Welsey, Publishing house) Latest edition.
4. Bipin Desai, Database System, TMG
5. Prateek Bhatia, Database Management system, Kalyani Publishers

chapter 2

Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client/server system architecture. This evolution mirrors the trends in computing, where large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks to various types of server machines—Web servers, database servers, file servers, application servers, and so on.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules.[1] A **client module** is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms- or menu-based GUIs (graphical user interfaces). The other kind of module, called a **server module**, typically handles data storage, access, search, and other functions. We discuss client/server architectures in more detail in Section 2.5. First, we must study more basic concepts that will give us a better understanding of modern database architectures.

In this chapter we present the terminology and basic concepts that will be used throughout the book. Section 2.1 discusses data models and defines the concepts of schemas and instances, which are fundamental to the study of database systems. Then, we discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3 we describe the types of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment.

[1]As we shall see in Section 2.5, there are variations on this simple *two-tier* client/server architecture.

Section 2.5 gives an overview of various types of client/server architectures. Finally, Section 2.6 presents a classification of the types of DBMS packages. Section 2.7 summarizes the chapter.

The material in Sections 2.4 through 2.6 provides more detailed concepts that may be considered as supplementary to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.[2] By *structure of a database* we mean the data types, relationships, and constraints that apply to the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.[3] An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapter 11) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 11) extend the basic relational model to include such concepts, among others. In the basic relational data model, there is a provision to attach behavior to the relations in the form of persistent stored modules, popularly known as stored procedures (see Chapter 13).

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically

[2]Sometimes the word *model* is used to denote a specific database description, or schema—for example, *the marketing data model*. We will not use this interpretation.

[3]The inclusion of concepts to describe behavior reflects a trend whereby database design and software design activities are increasingly being combined into a single activity. Traditionally, specifying behavior is associated with software design.

magnetic disks. Concepts provided by low-level data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational** (or **implementation**) **data models**,[4] which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project from the miniworld that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project. Chapter 7 presents the **Entity-Relationship model**—a popular high-level conceptual data model. Chapter 8 describes additional abstractions used for advanced modeling, such as generalization, specialization, and categories (union types).

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part 2 is devoted to the relational data model, and its constraints, operations and languages.[5] The SQL standard for relational databases is described in Chapters 4 and 5. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

We can regard the **object data model** as an example of a new family of higher-level implementation data models that are closer to conceptual data models. A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG). We describe the general characteristics of object databases and the object model proposed standard in Chapter 11. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a structure that makes the search for particular database records efficient. We discuss physical storage techniques and access structures in Chapters 17 and 18. An **index** is an example of an access path that allows direct access to data using an index term or a keyword. It is similar to the index at the end of this book, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

---

[4]The term *implementation data model* is not a standard term; we have introduced it to refer to the available data models in commercial database systems.

[5]A summary of the hierarchical and network data models is included in Appendices D and E. They are accessible from the book's Web site.

2.1.2 Schemas, Instances, and Database State

In any data model, it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.[6] Most data models have certain conventions for displaying schemas as diagrams.[7] A displayed schema is called a **schema diagram**. Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 2.1 shows neither the data type of each data item, nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as *students majoring in computer science must take CS1310 before the end of their sophomore year* is quite difficult to represent diagrammatically.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 changes every time we add a new student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the *current* set of **occurrences** or **instances** in the

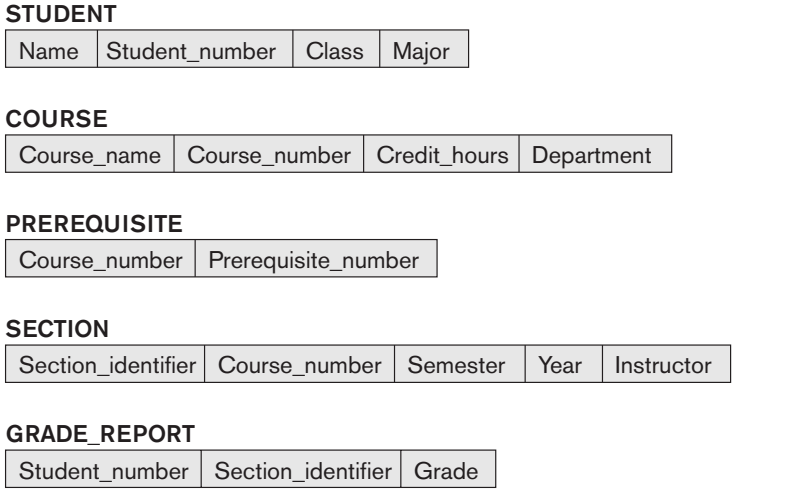


Figure 2.1
Schema diagram for the database in Figure 1.2.

[6]Schema changes are usually needed as the requirements of the database applications change. Newer database systems include operations for allowing schema changes, although the schema change process is more involved than simple database updates.

[7]It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is also sometimes used to refer to a schema.

database. In a given database state, each schema construct has its own *current set* of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.[8] The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the Date_of_birth to the STUDENT schema in Figure 2.1. This is known as **schema evolution**. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

2.2 Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) use of a catalog to store the database description (schema) so as to make it self-describing, (2) insulation of programs and data (program-data and program-operation independence), and (3) support of multiple user views. In this section we specify an architecture for database systems, called the **three-schema architecture**,[9] that was proposed to help achieve and visualize these characteristics. Then we discuss the concept of data independence further.

---

[8]The current state is also called the *current snapshot* of the database. It has also been called a *database instance*, but we prefer to use the term *instance* to refer to individual records.

[9]This is also known as the ANSI/SPARC architecture, after the committee that proposed it (Tsichritzis and Klug 1978).

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.
3. The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level data model.

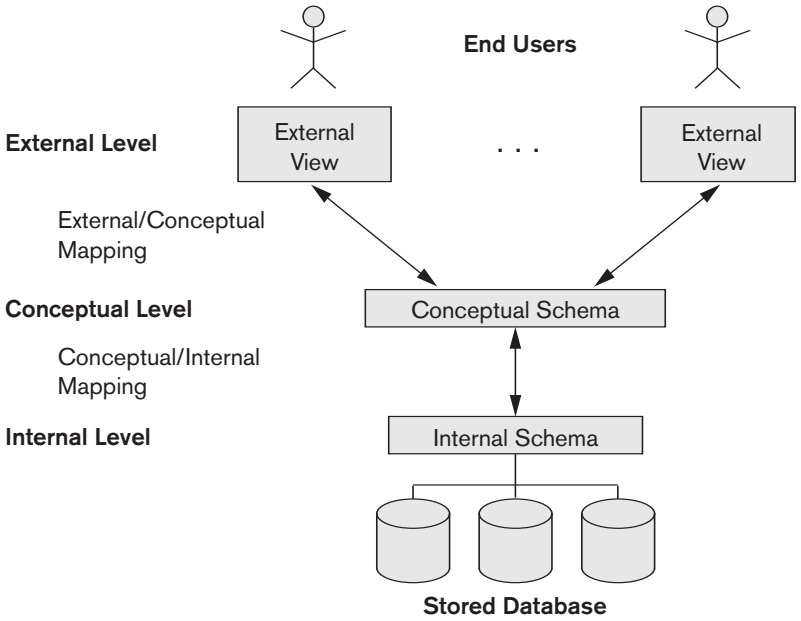


Figure 2.2
The three-schema architecture.

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely and explicitly, but support the three-schema architecture to some extent. Some older DBMSs may include physical-level details in the conceptual schema. The three-level ANSI architecture has an important place in database technology development because it clearly separates the users' external level, the database's conceptual level, and the internal storage level for designing a database. It is very much applicable in the design of DBMSs, even today. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information (for example, a relational DBMS like Oracle uses SQL for this). Some DBMSs allow different data models to be used at the conceptual and external levels. An example is Universal Data Base (UDB), a DBMS from IBM, which uses the relational model to describe the conceptual schema, but may use an object-oriented model to describe an external schema.

Notice that the three schemas are only *descriptions* of data; the stored data that *actually* exists is at the physical level only. In a DBMS based on the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence**, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the GRADE_REPORT file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.

Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of section records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Generally, physical data independence exists in most databases and file environments where physical details such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical data independence is harder to achieve because it allows structural and constraint changes without affecting application programs—a much stricter requirement.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

2.3 Database Languages and Interfaces

In Section 1.4 we discussed the variety of users supported by a DBMS. The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database

and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language** (**DDL**), is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language** (**SDL**), is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. In most relational DBMSs today, there *is no specific language* that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage. These permit the DBA staff to control indexing choices and mapping of data to storage. For a true three-schema architecture, we would need a third language, the **view definition language** (**VDL**), to specify user views and their mappings to the conceptual schema, but in most DBMSs *the DDL is used to define both conceptual and external schemas*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries (see Chapters 4 and 5).

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language** (**DML**) for these purposes.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, which is usually done by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapters 4 and 5), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification, schema evolution, and other features. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level** or **nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a precompiler and processed by the DBMS. A **low-level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming

language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. DL/1, a DML designed for the hierarchical model, is a low-level DML that uses commands such as GET UNIQUE, GET NEXT, or GET NEXT WITHIN PARENT to navigate from record to record within a hierarchy of records in the database. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.[10] On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.[11]

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-Based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called **menus**) that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step-by-step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also often used in **browsing interfaces**, which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages**,

[10] In object databases, the host and data sublanguages typically form one integrated language—for example, C++ with some extensions to support database functionality. Some relational systems also provide integrated languages—for example, Oracle's PL/SQL.

[11] According to the English meaning of the word *query*, it should really be used to describe retrievals only, not updates.

which are special languages that help programmers specify such forms. SQL*Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema. Oracle Forms is a component of the Oracle product suite that provides an extensive set of features to design and build applications using forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces. A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device**, such as a mouse, to select certain parts of the displayed schema diagram.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to *understand* them. A natural language interface usually has its own *schema*, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request. The capabilities of natural language interfaces have not advanced rapidly. Today, we see search engines that accept strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask). They use predefined indexes on words and use ranking functions to retrieve and present resulting documents in a decreasing degree of match. Such "free form" textual query interfaces are not yet common in structured relational or legacy model databases, although a research area called **keyword-based querying** has emerged recently for relational databases.

Speech Input and Output. Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. Applications with limited vocabularies such as inquiries for telephone directory, flight arrival/departure, and credit card account information are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Systems analysts and programmers design and implement a special interface for each known class of naive users. Usually a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example,

function keys in a terminal can be programmed to initiate various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. In this section we discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

2.4.1 DBMS Component Modules

Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system** (**OS**), which schedules disk read/write. Many DBMSs have their own **buffer management** module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

Let us consider the top part of Figure 2.3 first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints. In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.

Casual users and persons with occasional need for information from the database interact using some form of interface, which we call the **interactive query** interface in Figure 2.3. We have not explicitly shown any menu-based or form-based interaction that may be used to generate the interactive query automatically. These queries are parsed and validated for correctness of the query syntax, the names of files and

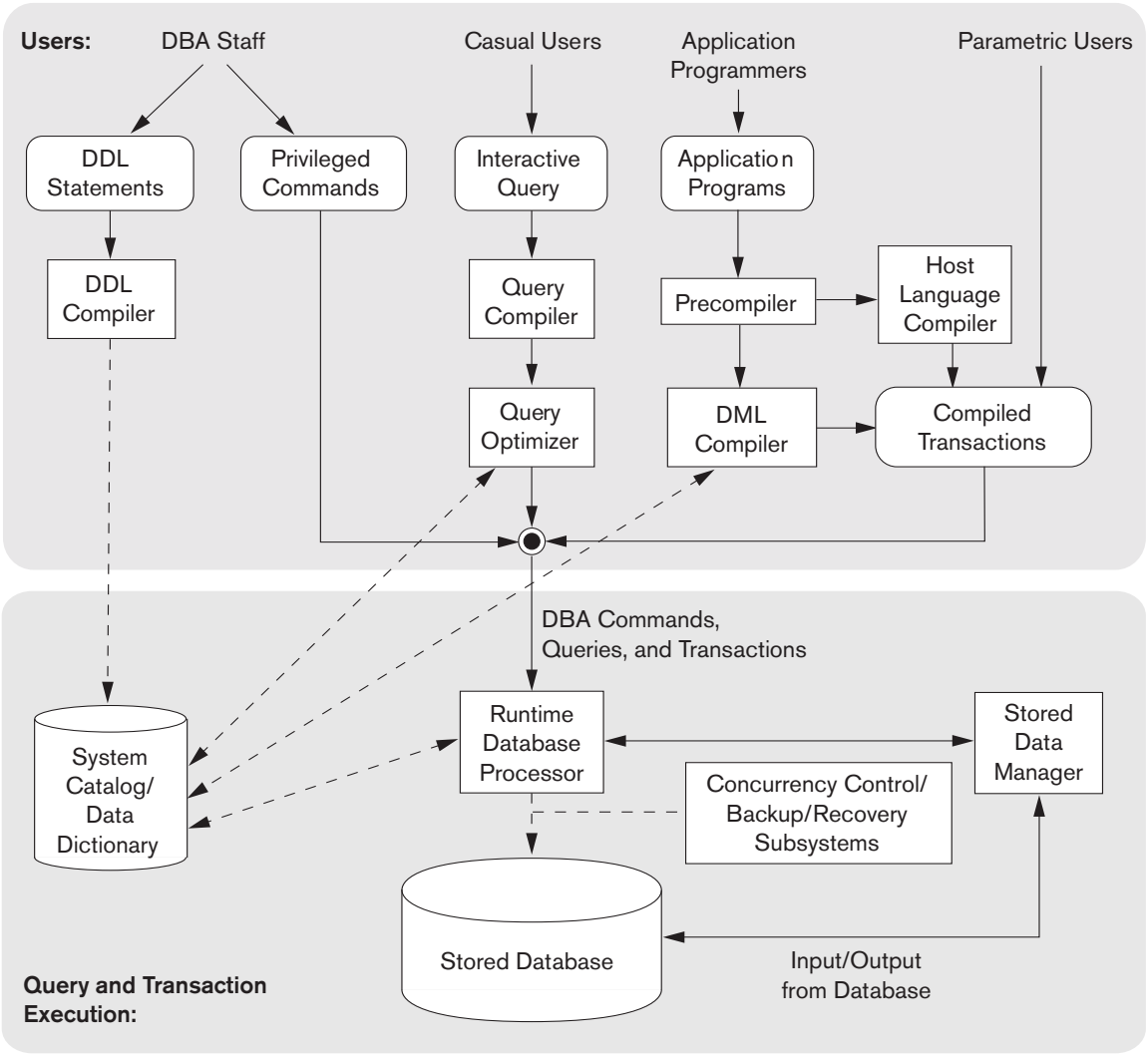


Figure 2.3
Component modules of a DBMS and their interactions.

data elements, and so on by a **query compiler** that compiles them into an internal form. This internal query is subjected to query optimization (discussed in Chapters 19 and 20). Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. Canned transactions are executed repeatedly by parametric users, who simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank withdrawal transaction where the account number and the amount may be supplied as parameters.

In the lower part of Figure 2.3, the **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module while others depend on the OS for buffer management. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.

It is now common to have the **client program** that accesses the DBMS running on a separate computer from the computer on which the database resides. The former is called the **client computer** running a DBMS client software and the latter is called the **database server**. In some cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server. We elaborate on this topic in Section 2.5.

Figure 2.3 is not meant to describe a specific DBMS; rather, it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory buffering of disk pages. The DBMS also interfaces with compilers for general-purpose host programming languages, and with application servers and client programs running on separate machines through the system network interface.

2.4.2 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have **database utilities** that help the DBA manage the database system. Common utilities have the following types of functions:

- **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) for-

mat of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called **conversion tools**. For the hierarchical DBMS called IMS (IBM) and for many network DBMSs including IDMS (Computer Associates), SUPRA (Cincom), and IMAGE (HP), the vendors or third-party companies are making a variety of conversion tools available (e.g., Cincom's SUPRA Server SQL) to transform data into the relational model.

- **Backup.** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex, but saves storage space.
- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations, and create new access paths to improve performance.
- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and the DBMS. CASE tools[12] are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**) **system**. In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**. This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

[12]Although CASE stands for computer-aided software engineering, many CASE tools are used primarily for database design.

Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

The DBMS also needs to interface with **communications software**, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers. These are connected to the database site through data communications hardware such as Internet routers, phone lines, long-haul networks, local networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a **DB/DC** system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often **local area networks (LANs)**, but they can also be other types of networks.

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

2.5.2 Basic Client/Server Architectures

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, data-

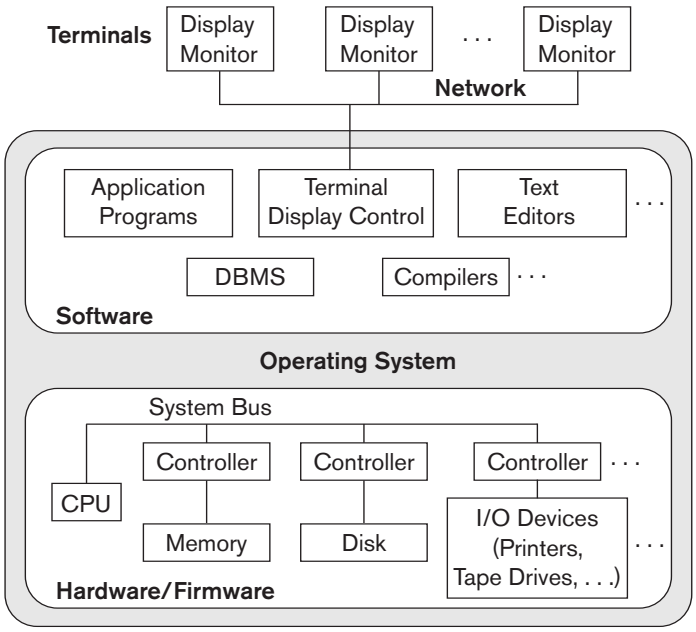


Figure 2.4
A physical centralized architecture.

base servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless workstations or workstations/PCs with disks that have only client software installed).

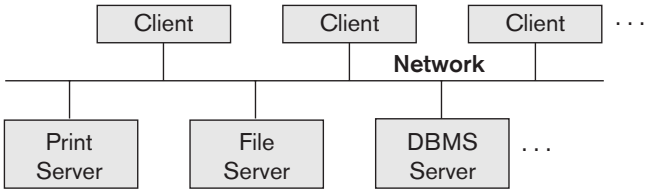


Figure 2.5
Logical two-tier client/server architecture.

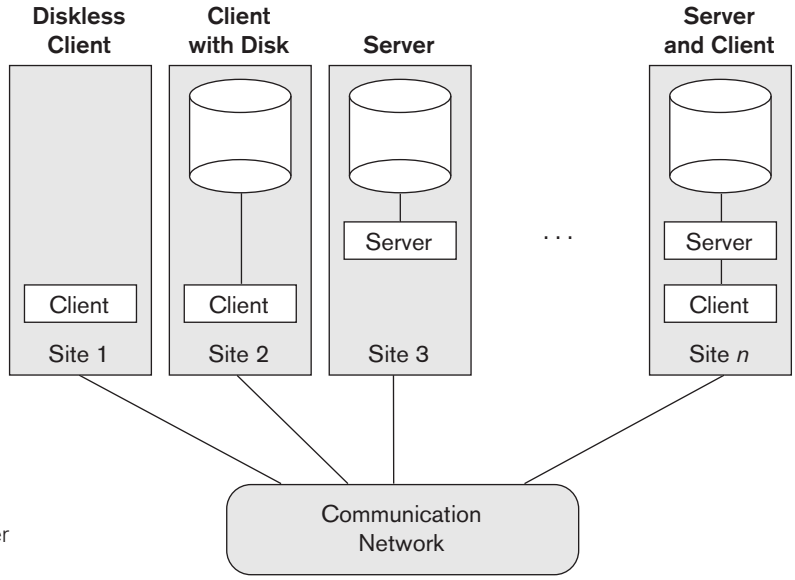


Figure 2.6
Physical two-tier client/server architecture.

Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.[13] We discuss them next.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 4 and 5) provided a standard language for RDBMSs, this created a logical dividing point

[13]There are many other variations of client/server architectures. We discuss the two most basic ones here.

between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity** (**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The different approach to two-tier client/server architecture was taken by some object-oriented DBMSs, where the software modules of the DBMS were divided between client and server in a more integrated way. For example, the **server level** may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the **client level** may handle the user interface; data dictionary functions; DBMS interactions with programming language compilers; global query optimization, concurrency control, and recovery across multiple servers; structuring of complex objects from the data in the buffers; and other such functions. In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users/programmers. The exact division of functionality can vary from system to system. In such a client/server architecture, the server has been called a **data server** because it provides data in disk pages to the client. This data can then be structured into objects for the client programs by the client-side DBMS software.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

2.5.4 Three-Tier and n-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

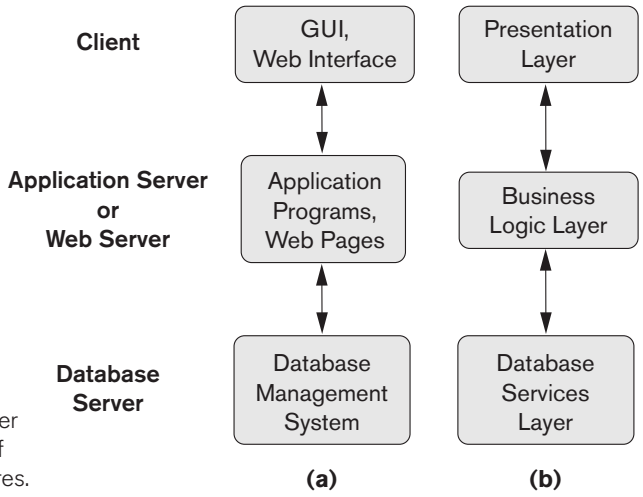


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface, application rules,* and *data access* act as the three tiers. Figure 2.7(b) shows another architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to *n*-tier architectures, where *n* may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, *n*-tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer,* which accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

2.6 Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**. The **object data model** has been implemented in some commercial systems but has not had widespread use. Many legacy applications still run on database systems based on the **hierarchical** and **network data models**. Examples of hierarchical DBMSs include IMS (IBM) and some other systems like System 2K (SAS Inc.) and TDMS. IMS is still used at governmental and industrial installations, including hospitals and banks, although many of its users have converted to relational systems. The network data model was used by many vendors and the resulting products like IDMS (Cullinet—now Computer Associates), DMS 1100 (Univac—now Unisys), IMAGE (Hewlett-Packard), VAX-DBMS (Digital—then Compaq and now HP), and SUPRA (Cincom) still have a following and their user groups have their own active organizations. If we add IBM's popular VSAM file system to these, we can easily say that a reasonable percentage of worldwide-computerized data is still in these so-called **legacy database systems**.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called **object-relational DBMS**s. We can categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

More recently, some experimental DBMSs are based on the XML (eXtended Markup Language) model, which is a tree-structured (hierarchical) data model. These have been called **native XML DBMSs.** Several commercial relational DBMSs have added XML interfaces and storage to their products.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with PCs. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous** DDBMSs use the same DBMS software at all the sites, whereas

heterogeneous DDBMSs can use different DBMS software at each site. It is also possible to develop **middleware software** to access several autonomous preexisting databases stored under heterogeneousDBMSs. This leads to a **federated** DBMS (or **multidatabase system**), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use client-server architecture, as we described in Section 2.5.

The fourth criterion is cost. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services. The main RDBMS products are available as free examination 30-day copy versions as well as personal versions, which may cost under $100 and allow a fair amount of functionality. The giant systems are being sold in modular form with components to handle distribution, replication, parallel processing, mobile capability, and so on, and with a large number of parameters that must be defined for the configuration. Furthermore, they are sold in the form of licenses—site licenses allow unlimited use of the database system with any number of copies running at the customer site. Another type of license limits the number of concurrent users or the number of user seats at a location. Standalone single user versions of some systems like Microsoft Access are sold per copy or included in the overall configuration of a desktop or laptop. In addition, data warehousing and mining features, as well as support for additional data types, are made available at extra cost. It is possible to pay millions of dollars for the installation and maintenance of large database systems annually.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **online transaction processing** (**OLTP**) systems, which must support a large number of concurrent transactions without imposing excessive delays.

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The basic **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 resembles a relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational model and its languages and operations in Chapters 3 through 6, and techniques for programming relational applications in Chapters 13 and 14.

The **object data model** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class**, and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of each class are specified in terms of predefined procedures called **methods**. Relational DBMSs have been extending their models to incorporate object database

concepts and other capabilities; these systems are referred to as **object-relational** or **extended relational systems**. We discuss object databases and object-relational systems in Chapter 11.

The **XML model** has emerged as a standard for exchanging data over the Web, and has been used as a basis for implementing several prototype native XML systems. XML uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex hierarchical structures. This model conceptually resembles the object model but uses different terminology. XML capabilities have been added to many commercial DBMS products. We present an overview of XML in Chapter 12.

Two older, historically important data models, now known as **legacy data models**, are the network and hierarchical models. The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type**. A 1:N, or one-to-many, relationship relates one instance of a record to many record instances using some pointer linking mechanism in these models. Figure 2.8 shows a network schema diagram for the database of Figure 2.1, where record types are shown as rectangles and set types are shown as labeled directed arrows.

The network model, also known as the CODASYL DBTG model,[14] has an associated record-at-a-time language that must be embedded in a host programming language. The network DML was proposed in the 1971 Database Task Group (DBTG) Report as an extension of the COBOL language. It provides commands for locating records directly (e.g., FIND ANY <record-type> USING <field-list>, or FIND DUPLICATE <record-type> USING <field-list>). It has commands to support traversals within set-types (e.g., GET OWNER, GET {FIRST, NEXT, LAST} MEMBER WITHIN <set-type> WHERE <condition>). It also has commands to store new data

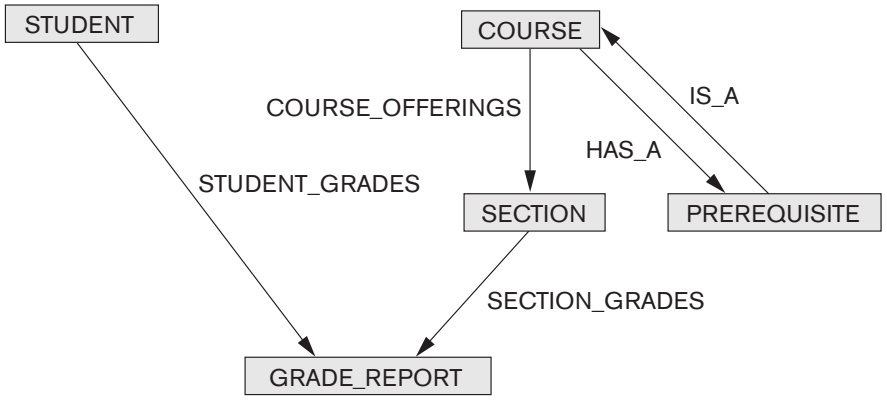


Figure 2.8
The schema of Figure 2.1 in network model notation.

[14]CODASYL DBTG stands for Conference on Data Systems Languages Database Task Group, which is the committee that specified the network model and its language.

(e.g., STORE <record-type>) and to make it part of a set type (e.g., CONNECT <record-type> TO <set-type>). The language also handles many additional considerations, such as the currency of record types and set types, which are defined by the current position of the navigation process within the database. It is prominently used by IDMS, IMAGE, and SUPRA DBMSs today.

The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model. A popular hierarchical DML is DL/1 of the IMS system. It dominated the DBMS market for over 20 years between 1965 and 1985 and is still a widely used DBMS worldwide, holding a large percentage of data in governmental, health care, and banking and insurance databases. Its DML, called DL/1, was a de facto industry standard for a long time. DL/1 has commands to locate a record (e.g., GET { UNIQUE, NEXT} <record-type> WHERE <condition>). It has navigational facilities to navigate within hierarchies (e.g., GET NEXT WITHIN PARENT or GET {FIRST, NEXT} PATH <hierarchical-path-specification> WHERE <condition>). It has appropriate facilities to store and update records (e.g., INSERT <record-type>, REPLACE <record-type>). Currency issues during navigation are also handled with additional features in the language.[15]

2.7 Summary

In this chapter we introduced the main concepts used in database systems. We defined a data model and we distinguished three main categories:

- High-level or conceptual data models (based on entities and relationships)
- Low-level or physical data models
- Representational or implementation data models (record-based, object-oriented)

We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. Then we described the three-schema DBMS architecture, which allows three schema levels:

- An internal schema describes the physical storage structure of the database.
- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

A DBMS that cleanly separates the three levels must have mappings between the schemas to transform requests and query results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.

---

[15]The full chapters on the network and hierarchical models from the second edition of this book are available from this book's Companion Website at http://www.aw.com/elmasri.

chapter 8

The Enhanced Entity-Relationship (EER) Model

The ER modeling concepts discussed in Chapter 7 are sufficient for representing many database schemas for *traditional* database applications, which include many data-processing applications in business and industry. Since the late 1970s, however, designers of database applications have tried to design more accurate database schemas that reflect the data properties and constraints more precisely. This was particularly important for newer applications of database technology, such as databases for engineering design and manufacturing (CAD/CAM),[1] telecommunications, complex software systems, and Geographic Information Systems (GIS), among many other applications. These types of databases have more complex requirements than do the more traditional applications. This led to the development of additional *semantic data modeling* concepts that were incorporated into conceptual data models such as the ER model. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science, such as the **knowledge representation** area of artificial intelligence and the **object modeling** area in software engineering.

In this chapter, we describe features that have been proposed for semantic data models, and show how the ER model can be enhanced to include these concepts, leading to the **Enhanced ER** (**EER**) model.[2] We start in Section 8.1 by incorporating the concepts of *class/subclass relationships* and *type inheritance* into the ER model. Then, in Section 8.2, we add the concepts of *specialization* and *generalization*. Section 8.3

[1]CAD/CAM stands for computer-aided design/computer-aided manufacturing.

[2]EER has also been used to stand for *Extended* ER model.

discusses the various types of *constraints* on specialization/generalization, and Section 8.4 shows how the UNION construct can be modeled by including the concept of *category* in the EER model. Section 8.5 gives a sample UNIVERSITY database schema in the EER model and summarizes the EER model concepts by giving formal definitions. We will use the terms *object* and *entity* interchangeably in this chapter, because many of these concepts are commonly used in object-oriented models.

We present the UML class diagram notation for representing specialization and generalization in Section 8.6, and briefly compare these with EER notation and concepts. This serves as an example of alternative notation, and is a continuation of Section 7.8, which presented basic UML class diagram notation that corresponds to the basic ER model. In Section 8.7, we discuss the fundamental abstractions that are used as the basis of many semantic data models. Section 8.8 summarizes the chapter.

For a detailed introduction to conceptual modeling, Chapter 8 should be considered a continuation of Chapter 7. However, if only a basic introduction to ER modeling is desired, this chapter may be omitted. Alternatively, the reader may choose to skip some or all of the later sections of this chapter (Sections 8.4 through 8.8).

8.1 Subclasses, Superclasses, and Inheritance

The EER model includes *all the modeling concepts of the ER model* that were presented in Chapter 7. In addition, it includes the concepts of **subclass** and **superclass** and the related concepts of **specialization** and **generalization** (see Sections 8.2 and 8.3). Another concept included in the EER model is that of a **category** or **union type** (see Section 8.4), which is used to represent a collection of objects (entities) that is the *union* of objects of different entity types. Associated with these concepts is the important mechanism of **attribute and relationship inheritance**. Unfortunately, no standard terminology exists for these concepts, so we use the most common terminology. Alternative terminology is given in footnotes. We also describe a diagrammatic technique for displaying these concepts when they arise in an EER schema. We call the resulting schema diagrams **enhanced ER** or **EER diagrams**.

The first Enhanced ER (EER) model concept we take up is that of a **subtype** or **subclass** of an entity type. As we discussed in Chapter 7, an entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database. For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database. In many cases an entity type has numerous subgroupings or subtypes of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, the entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these

subgroupings is also an employee. We call each of these subgroupings a **subclass** or **subtype** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** or **supertype** for each of these subclasses. Figure 8.1 shows how to represent these concepts diagramatically in EER diagrams. (The circle notation in Figure 8.1 will be explained in Section 8.2.)

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or **supertype/subtype** or simply **class/subclass relationship**.[3] In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass represents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'Joan Logano' is also the EMPLOYEE 'Joan Logano.' Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role.* When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 9.2, we discuss various options for representing superclass/subclass relationships in relational databases.

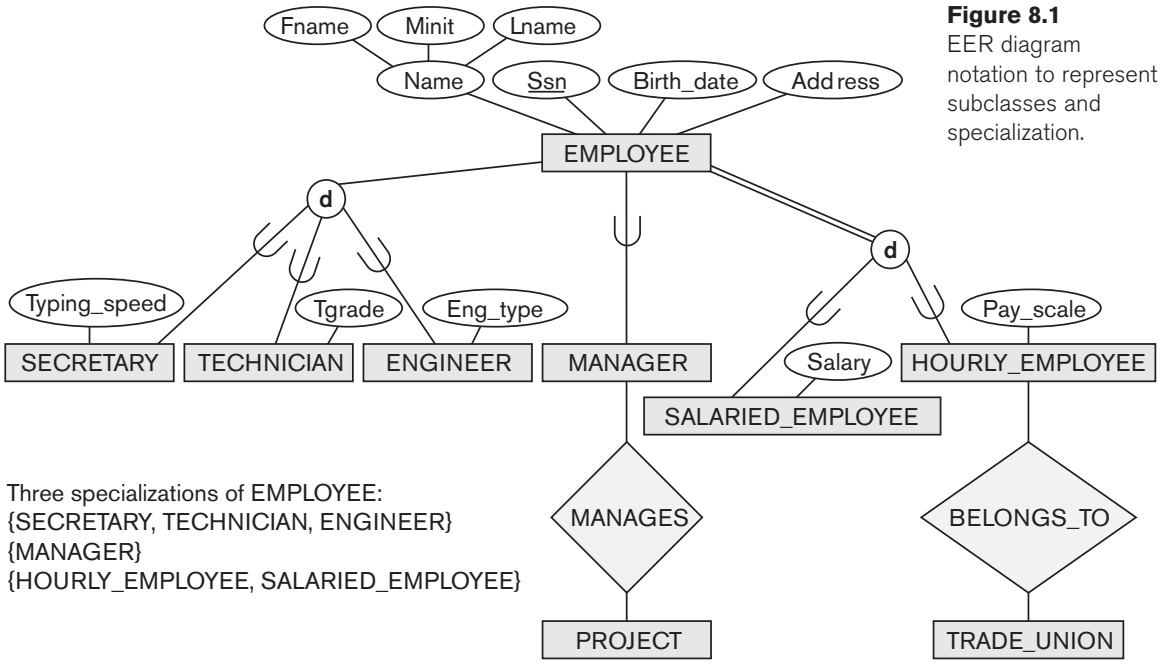


Figure 8.1
EER diagram notation to represent subclasses and specialization.

[3]A class/subclass relationship is often called an **IS-A** (or **IS-AN**) **relationship** because of the way we refer to the concept. We say a SECRETARY *is an* EMPLOYEE, a TECHNICIAN *is an* EMPLOYEE, and so on.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass is a member of some subclass.

An important concept associated with subclasses (subtypes) is that of **type inheritance**. Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.[4]

8.2 Specialization and Generalization

8.2.1 Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the *job type* of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay.*

Figure 8.1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.[5] Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local**

[4]In some object-oriented programming languages, a common restriction is that an entity (or object) has *only one type*. This is generally too restrictive for conceptual database modeling.

[5]There are many alternative notations for specialization; we present the UML notation in Section 8.6 and other proposed notations in Appendix A.

attributes) of the subclass. Similarly, a subclass can participate in **specific relationship types**, such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Figure 8.1. We will explain the **d** symbol in the circles in Figure 8.1 and additional EER diagram notation shortly.

Figure 8.2 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e_1 is shown in both EMPLOYEE and SECRETARY in Figure 8.2. As the figure suggests, a superclass/subclass relationship such as EMPLOYEE/SECRETARY somewhat resembles a 1:1 relationship *at the instance level* (see Figure 7.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.

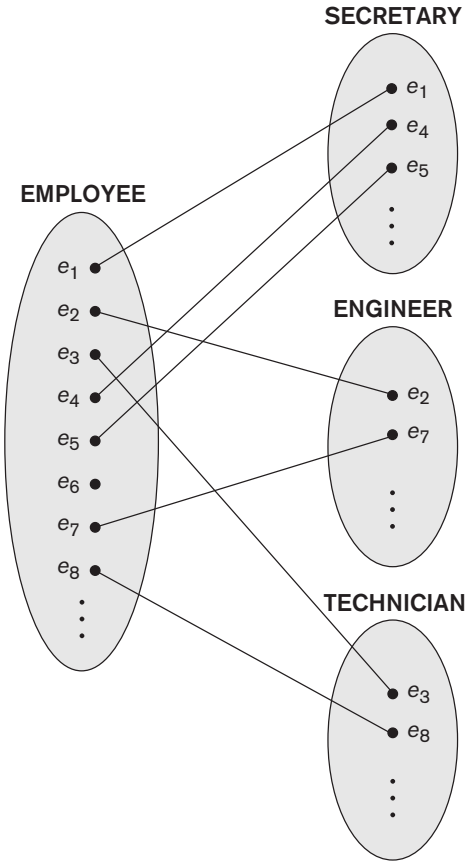


Figure 8.2
Instances of a specialization.

There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, in Figure 8.1 the SECRETARY subclass has the specific attribute Typing_speed, whereas the ENGINEER subclass has the specific attribute Eng_type, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 8.1.

In summary, the specialization process allows us to do the following:

- Define a set of subclasses of an entity type
- Establish additional specific attributes with each subclass
- Establish additional specific relationship types between each subclass and other entity types or other subclasses

8.2.2 Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses**. For example, consider the entity types CAR and TRUCK shown in Figure 8.3(a). Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 8.3(b). Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 8.3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 8.1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation because the decision as to which process is followed in a particular situation is often subjective. Appendix A gives some of the suggested alternative diagrammatic notations for schema diagrams and class diagrams.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a

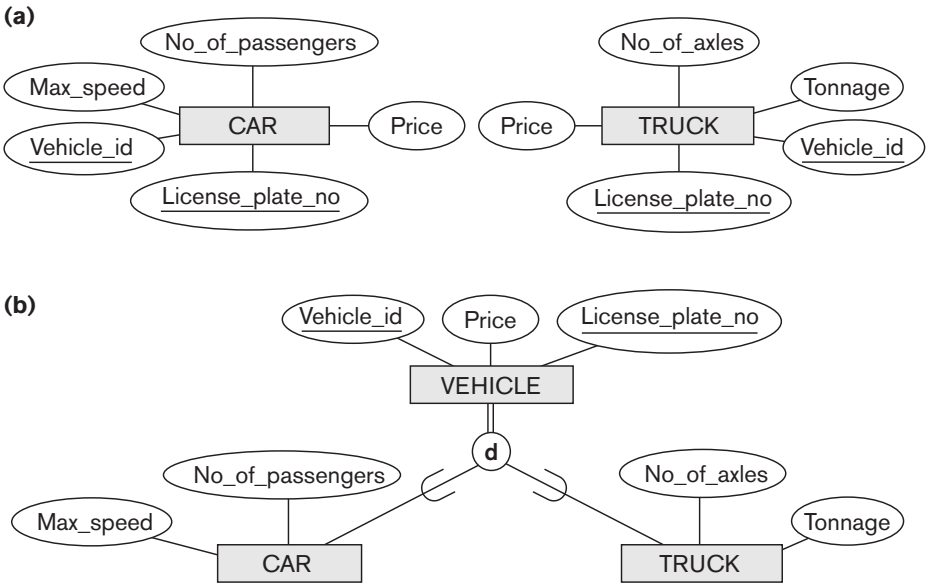


Figure 8.3
Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are all shown in rectangles in EER diagrams, like entity types. Next, we discuss the properties of specializations and generalizations in more detail.

8.3 Constraints and Characteristics of Specialization and Generalization Hierarchies

First, we discuss constraints that apply to a single specialization or a single generalization. For brevity, our discussion refers only to *specialization* even though it applies to *both* specialization and generalization. Then, we discuss differences between specialization/generalization *lattices* (*multiple inheritance*) and *hierarchies* (*single inheritance*), and elaborate on the differences between the specialization and generalization processes during conceptual database schema design.

8.3.1 Constraints on Specialization and Generalization

In general, we may have several specializations defined on the same entity type (or superclass), as shown in Figure 8.1. In such a case, entities may belong to subclasses

in each of the specializations. However, a specialization may also consist of a *single* subclass only, such as the {MANAGER} specialization in Figure 8.1; in such a case, we do not use the circle notation.

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called **predicate-defined** (or **condition-defined**) **subclasses**. For example, if the EMPLOYEE entity type has an attribute Job_type, as shown in Figure 8.4, we can specify the condition of membership in the SECRETARY subclass by the condition (Job_type = 'Secretary'), which we call the **defining predicate** of the subclass. This condition is a *constraint* specifying that exactly those entities of the EMPLOYEE entity type whose attribute value for Job_type is 'Secretary' belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called the **defining attribute** of the specialization.[6] In this case, all the entities with the same value for the attribute belong to the same subclass. We display an attribute-defined specialization by placing the defining attribute name next to the arc from the circle to the superclass, as shown in Figure 8.4.

When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined**. Membership in such a subclass is determined by the database users when they apply the operation to add an entity to the subclass; hence, membership is *specified individually for each entity by the user*, not by any condition that may be evaluated automatically.

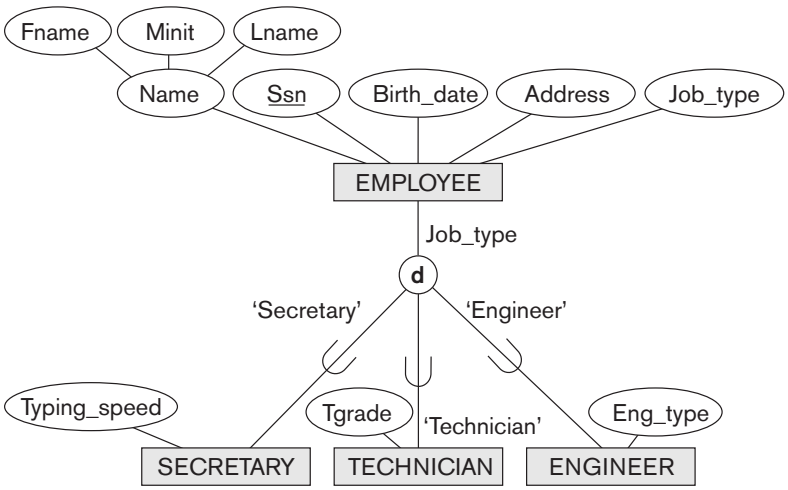


Figure 8.4
EER diagram notation for an attribute-defined specialization on Job_type.

[6]Such an attribute is called a *discriminator* in UML terminology.

Two other constraints may apply to a specialization. The first is the **disjointness (or disjointedness) constraint**, which specifies that the subclasses of the specialization must be disjoint. This means that an entity can be a member of *at most* one of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint (if the attribute used to define the membership predicate is single-valued). Figure 8.4 illustrates this case, where the **d** in the circle stands for *disjoint*. The **d** notation also applies to user-defined subclasses of a specialization that must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 8.1. If the subclasses are not constrained to be disjoint, their sets of entities may be **overlapping**; that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an **o** in the circle, as shown in Figure 8.5.

The second constraint on specialization is called the **completeness** (or **totalness**) **constraint**, which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 8.1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization**, which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} in Figures 8.1 and 8.4, then that specialization is partial.[7]

Notice that the disjointness and completeness constraints are *independent*. Hence, we have the following four possible constraints on specialization:

- Disjoint, total
- Disjoint, partial
- Overlapping, total
- Overlapping, partial

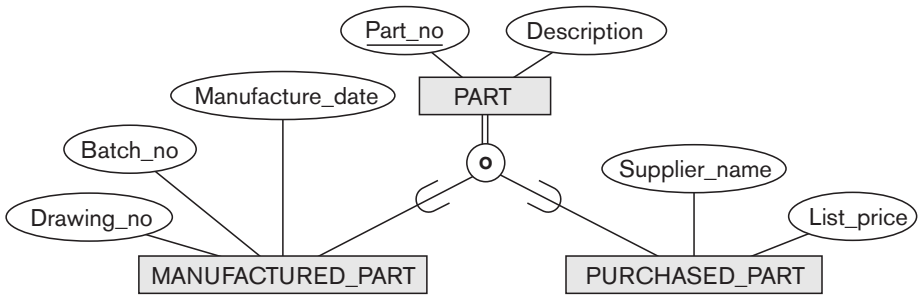


Figure 8.5
EER diagram notation for an overlapping (nondisjoint) specialization.

[7]The notation of using single or double lines is similar to that for partial or total participation of an entity type in a relationship type, as described in Chapter 7.

Of course, the correct constraint is determined from the real-world meaning that applies to each specialization. In general, a superclass that was identified through the *generalization* process usually is **total**, because the superclass is *derived from* the subclasses and hence contains only the entities that are in the subclasses.

Certain insertion and deletion rules apply to specialization (and generalization) as a consequence of the constraints specified earlier. Some of these rules are as follows:

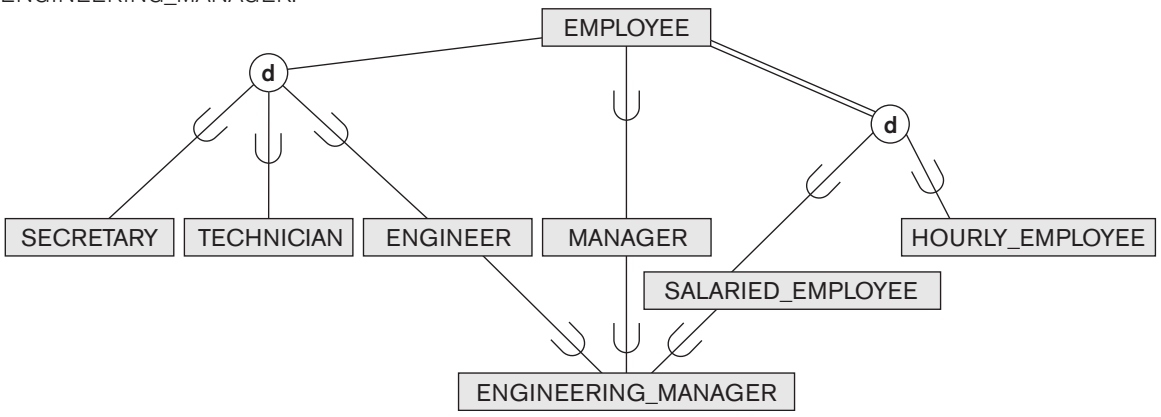
- Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.
- Inserting an entity in a superclass implies that the entity is mandatorily inserted in all *predicate-defined* (or *attribute-defined*) subclasses for which the entity satisfies the defining predicate.
- Inserting an entity in a superclass of a *total specialization* implies that the entity is mandatorily inserted in at least one of the subclasses of the specialization.

The reader is encouraged to make a complete list of rules for insertions and deletions for the various types of specializations.

8.3.2 Specialization and Generalization Hierarchies and Lattices

A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations. For example, in Figure 8.6 ENGINEER is a subclass of EMPLOYEE and is also a superclass of ENGINEERING_MANAGER; this represents the real-world constraint that every engineering manager is required to be an engineer. A **specialization hierarchy** has the constraint that every subclass participates *as a subclass* in *only one* class/subclass relationship; that is, each subclass has only

Figure 8.6
A specialization lattice with shared subclass ENGINEERING_MANAGER.



one parent, which results in a **tree structure** or **strict hierarchy**. In contrast, for a **specialization lattice**, a subclass can be a subclass in *more than one* class/subclass relationship. Hence, Figure 8.6 is a lattice.

Figure 8.7 shows another specialization lattice of more than one level. This may be part of a conceptual schema for a UNIVERSITY database. Notice that this arrangement would have been a hierarchy except for the STUDENT_ASSISTANT subclass, which is a subclass in two distinct class/subclass relationships.

The requirements for the part of the UNIVERSITY database shown in Figure 8.7 are the following:

1. The database keeps track of three types of persons: employees, alumni, and students. A person can belong to one, two, or all three of these types. Each person has a name, SSN, sex, address, and birth date.
2. Every employee has a salary, and there are three types of employees: faculty, staff, and student assistants. Each employee belongs to exactly one of these types. For each alumnus, a record of the degree or degrees that he or she

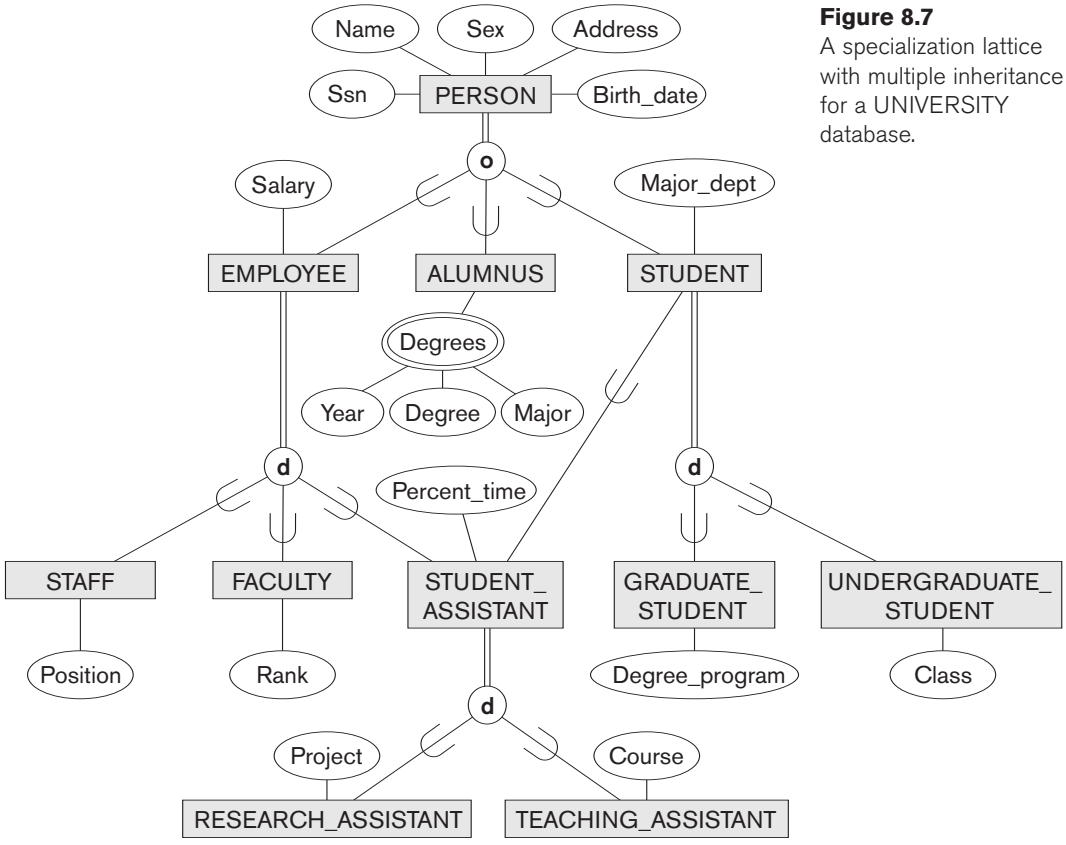


Figure 8.7
A specialization lattice with multiple inheritance for a UNIVERSITY database.

earned at the university is kept, including the name of the degree, the year granted, and the major department. Each student has a major department.

3. Each faculty has a rank, whereas each staff member has a staff position. Student assistants are classified further as either research assistants or teaching assistants, and the percent of time that they work is recorded in the database. Research assistants have their research project stored, whereas teaching assistants have the current course they work on.
4. Students are further classified as either graduate or undergraduate, with the specific attributes degree program (M.S., Ph.D., M.B.A., and so on) for graduate students and class (freshman, sophomore, and so on) for undergraduates.

In Figure 8.7, all person entities represented in the database are members of the PERSON entity type, which is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping; for example, an alumnus may also be an employee and may also be a student pursuing an advanced degree. The subclass STUDENT is the superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}, while EMPLOYEE is the superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}. Notice that STUDENT_ASSISTANT is also a subclass of STUDENT. Finally, STUDENT_ASSISTANT is the superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

In such a specialization lattice or hierarchy, a subclass inherits the attributes not only of its direct superclass, but also of all its predecessor superclasses *all the way to the root* of the hierarchy or lattice if necessary. For example, an entity in GRADUATE_STUDENT inherits all the attributes of that entity as a STUDENT *and* as a PERSON. Notice that an entity may exist in several *leaf nodes* of the hierarchy, where a **leaf node** is a class that has *no subclasses of its own.* For example, a member of GRADUATE_STUDENT may also be a member of RESEARCH_ASSISTANT.

A subclass with *more than one* superclass is called a **shared subclass**, such as ENGINEERING_MANAGER in Figure 8.6. This leads to the concept known as **multiple inheritance**, where the shared subclass ENGINEERING_MANAGER directly inherits attributes and relationships from multiple classes. Notice that the existence of at least one shared subclass leads to a lattice (and hence to *multiple inheritance*); if no shared subclasses existed, we would have a hierarchy rather than a lattice and only **single inheritance** would exist. An important rule related to multiple inheritance can be illustrated by the example of the shared subclass STUDENT_ASSISTANT in Figure 8.7, which inherits attributes from both EMPLOYEE and STUDENT. Here, both EMPLOYEE and STUDENT inherit *the same attributes* from PERSON. The rule states that if an attribute (or relationship) originating in the *same superclass* (PERSON) is inherited more than once via different paths (EMPLOYEE and STUDENT) in the lattice, then it should be included only once in the shared subclass (STUDENT_ASSISTANT). Hence, the attributes of PERSON are inherited *only once* in the STUDENT_ASSISTANT subclass in Figure 8.7.

It is important to note here that some models and languages are limited to **single inheritance** and *do not allow* multiple inheritance (shared subclasses). It is also important to note that some models do not allow an entity to have multiple types, and hence an entity can be a member *of only one leaf class.*[8] In such a model, it is necessary to create additional subclasses as leaf nodes to cover all possible combinations of classes that may have some entity that belongs to all these classes simultaneously. For example, in the overlapping specialization of PERSON into {EMPLOYEE, ALUMNUS, STUDENT} (or {E, A, S} for short), it would be necessary to create seven subclasses of PERSON in order to cover all possible types of entities: E, A, S, E_A, E_S, A_S, and E_A_S. Obviously, this can lead to extra complexity.

Although we have used specialization to illustrate our discussion, similar concepts *apply equally* to generalization, as we mentioned at the beginning of this section. Hence, we can also speak of **generalization hierarchies** and **generalization lattices**.

8.3.3 Utilizing Specialization and Generalization in Refining Conceptual Schemas

Now we elaborate on the differences between the specialization and generalization processes, and how they are used to refine conceptual schemas during conceptual database design. In the specialization process, we typically start with an entity type and then define subclasses of the entity type by successive specialization; that is, we repeatedly define more specific groupings of the entity type. For example, when designing the specialization lattice in Figure 8.7, we may first specify an entity type PERSON for a university database. Then we discover that three types of persons will be represented in the database: university employees, alumni, and students. We create the specialization {EMPLOYEE, ALUMNUS, STUDENT} for this purpose and choose the overlapping constraint, because a person may belong to more than one of the subclasses. We specialize EMPLOYEE further into {STAFF, FACULTY, STUDENT_ASSISTANT}, and specialize STUDENT into {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. Finally, we specialize STUDENT_ASSISTANT into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}. This successive specialization corresponds to a **top-down conceptual refinement process** during conceptual schema design. So far, we have a hierarchy; then we realize that STUDENT_ASSISTANT is a shared subclass, since it is also a subclass of STUDENT, leading to the lattice.

It is possible to arrive at the same hierarchy or lattice from the other direction. In such a case, the process involves generalization rather than specialization and corresponds to a **bottom-up conceptual synthesis**. For example, the database designers may first discover entity types such as STAFF, FACULTY, ALUMNUS, GRADUATE_STUDENT, UNDERGRADUATE_STUDENT, RESEARCH_ASSISTANT, TEACHING_ASSISTANT, and so on; then they generalize {GRADUATE_STUDENT,

[8]In some models, the class is further restricted to be a *leaf node* in the hierarchy or lattice.

UNDERGRADUATE_STUDENT} into STUDENT; then they generalize {RESEARCH_ASSISTANT, TEACHING_ASSISTANT} into STUDENT_ASSISTANT; then they generalize {STAFF, FACULTY, STUDENT_ASSISTANT} into EMPLOYEE; and finally they generalize {EMPLOYEE, ALUMNUS, STUDENT} into PERSON.

In structural terms, hierarchies or lattices resulting from either process may be identical; the only difference relates to the manner or order in which the schema superclasses and subclasses were created during the design process. In practice, it is likely that neither the generalization process nor the specialization process is followed strictly, but that a combination of the two processes is employed. New classes are continually incorporated into a hierarchy or lattice as they become apparent to users and designers. Notice that the notion of representing data and knowledge by using superclass/subclass hierarchies and lattices is quite common in knowledge-based systems and expert systems, which combine database technology with artificial intelligence techniques. For example, frame-based knowledge representation schemes closely resemble class hierarchies. Specialization is also common in software engineering design methodologies that are based on the object-oriented paradigm.

8.4 Modeling of UNION Types Using Categories

All of the superclass/subclass relationships we have seen thus far have a *single superclass.* A shared subclass such as ENGINEERING_MANAGER in the lattice in Figure 8.6 is the subclass in three *distinct* superclass/subclass relationships, where each of the three relationships has a *single* superclass. However, it is sometimes necessary to represent a single superclass/subclass relationship with *more than one* superclass, where the superclasses represent different entity types. In this case, the subclass will represent a collection of objects that is a subset of the UNION of distinct entity types; we call such a *subclass* a **union type** or a **category**.[9]

For example, suppose that we have three entity types: PERSON, BANK, and COMPANY. In a database for motor vehicle registration, an owner of a vehicle can be a person, a bank (holding a lien on a vehicle), or a company. We need to create a class (collection of entities) that includes entities of all three types to play the role of *vehicle owner.* A category (union type) OWNER that is a *subclass of the UNION* of the three entity sets of COMPANY, BANK, and PERSON can be created for this purpose. We display categories in an EER diagram as shown in Figure 8.8. The superclasses COMPANY, BANK, and PERSON are connected to the circle with the ∪ symbol, which stands for the *set union operation.* An arc with the subset symbol connects the circle to the (subclass) OWNER category. If a defining predicate is needed, it is displayed next to the line from the superclass to which the predicate applies. In Figure 8.8 we have two categories: OWNER, which is a subclass of the union of PERSON, BANK, and COMPANY; and REGISTERED_VEHICLE, which is a subclass of the union of CAR and TRUCK.

[9]Our use of the term *category* is based on the ECR (Entity-Category-Relationship) model (Elmasri et al. 1985).

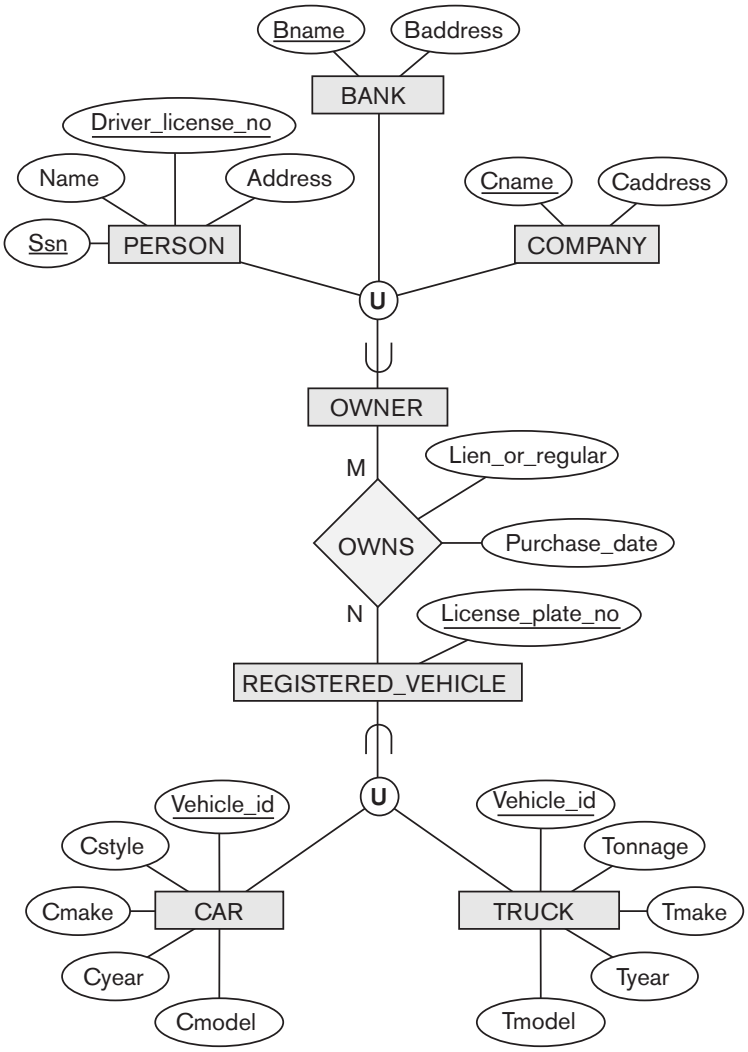


Figure 8.8
Two categories (union types): OWNER and REGISTERED_VEHICLE.

A category has two or more superclasses that may represent *distinct entity types,* whereas other superclass/subclass relationships always have a single superclass. To better understand the difference, we can compare a category, such as OWNER in Figure 8.8, with the ENGINEERING_MANAGER shared subclass in Figure 8.6. The latter is a subclass of *each of* the three superclasses ENGINEER, MANAGER, and SALARIED_EMPLOYEE, so an entity that is a member of ENGINEERING_MANAGER must exist in *all three.* This represents the constraint that an engineering manager must be an ENGINEER, a MANAGER, *and* a SALARIED_EMPLOYEE; that is, ENGINEERING_MANAGER is a subset of the *intersection* of the three classes (sets of entities). On the other hand, a category is a subset of the *union* of its superclasses. Hence, an entity that is a member of OWNER must exist in *only one* of the super-

classes. This represents the constraint that an OWNER may be a COMPANY, a BANK, *or* a PERSON in Figure 8.8.

Attribute inheritance works more selectively in the case of categories. For example, in Figure 8.8 each OWNER entity inherits the attributes of a COMPANY, a PERSON, or a BANK, depending on the superclass to which the entity belongs. On the other hand, a shared subclass such as ENGINEERING_MANAGER (Figure 8.6) inherits *all* the attributes of its superclasses SALARIED_EMPLOYEE, ENGINEER, and MANAGER.

It is interesting to note the difference between the category REGISTERED_VEHICLE (Figure 8.8) and the generalized superclass VEHICLE (Figure 8.3(b)). In Figure 8.3(b), every car and every truck is a VEHICLE; but in Figure 8.8, the REGISTERED_VEHICLE category includes some cars and some trucks but not necessarily all of them (for example, some cars or trucks may not be registered). In general, a specialization or generalization such as that in Figure 8.3(b), if it were *partial*, would not preclude VEHICLE from containing other types of entities, such as motorcycles. However, a category such as REGISTERED_VEHICLE in Figure 8.8 implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE.

A category can be **total** or **partial**. A total category holds the *union* of all entities in its superclasses, whereas a partial category can hold a *subset of the union*. A total category is represented diagrammatically by a double line connecting the category and the circle, whereas a partial category is indicated by a single line.

The superclasses of a category may have different key attributes, as demonstrated by the OWNER category in Figure 8.8, or they may have the same key attribute, as demonstrated by the REGISTERED_VEHICLE category. Notice that if a category is total (not partial), it may be represented alternatively as a total specialization (or a total generalization). In this case, the choice of which representation to use is subjective. If the two classes represent the same type of entities and share numerous attributes, including the same key attributes, specialization/generalization is preferred; otherwise, categorization (union type) is more appropriate.

It is important to note that some modeling methodologies do not have union types. In these models, a union type must be represented in a roundabout way (see Section 9.2).

8.5 A Sample UNIVERSITY EER Schema, Design Choices, and Formal Definitions

In this section, we first give an example of a database schema in the EER model to illustrate the use of the various concepts discussed here and in Chapter 7. Then, we discuss design choices for conceptual schemas, and finally we summarize the EER model concepts and define them formally in the same manner in which we formally defined the concepts of the basic ER model in Chapter 7.

8.5.1 The UNIVERSITY Database Example

For our sample database application, consider a UNIVERSITY database that keeps track of students and their majors, transcripts, and registration as well as of the university's course offerings. The database also keeps track of the sponsored research projects of faculty and graduate students. This schema is shown in Figure 8.9. A discussion of the requirements that led to this schema follows.

For each person, the database maintains information on the person's Name [Name], Social Security number [Ssn], address [Address], sex [Sex], and birth date [Bdate]. Two subclasses of the PERSON entity type are identified: FACULTY and STUDENT. Specific attributes of FACULTY are rank [Rank] (assistant, associate, adjunct, research, visiting, and so on), office [Foffice], office phone [Fphone], and salary [Salary]. All faculty members are related to the academic department(s) with which they are affiliated [BELONGS] (a faculty member can be associated with several departments, so the relationship is M:N). A specific attribute of STUDENT is [Class] (freshman=1, sophomore=2, ..., graduate student=5). Each STUDENT is also related to his or her major and minor departments (if known) [MAJOR] and [MINOR], to the course sections he or she is currently attending [REGISTERED], and to the courses completed [TRANSCRIPT]. Each TRANSCRIPT instance includes the grade the student received [Grade] in a section of a course.

GRAD_STUDENT is a subclass of STUDENT, with the defining predicate Class = 5. For each graduate student, we keep a list of previous degrees in a composite, multivalued attribute [Degrees]. We also relate the graduate student to a faculty advisor [ADVISOR] and to a thesis committee [COMMITTEE], if one exists.

An academic department has the attributes name [Dname], telephone [Dphone], and office number [Office] and is related to the faculty member who is its chairperson [CHAIRS] and to the college to which it belongs [CD]. Each college has attributes college name [Cname], office number [Coffice], and the name of its dean [Dean].

A course has attributes course number [C#], course name [Cname], and course description [Cdesc]. Several sections of each course are offered, with each section having the attributes section number [Sec#] and the year and quarter in which the section was offered ([Year] and [Qtr]).[10] Section numbers uniquely identify each section. The sections being offered during the current quarter are in a subclass CURRENT_SECTION of SECTION, with the defining predicate Qtr = Current_qtr and Year = Current_year. Each section is related to the instructor who taught or is teaching it ([TEACH]), if that instructor is in the database.

The category INSTRUCTOR_RESEARCHER is a subset of the union of FACULTY and GRAD_STUDENT and includes all faculty, as well as graduate students who are supported by teaching or research. Finally, the entity type GRANT keeps track of research grants and contracts awarded to the university. Each grant has attributes grant title [Title], grant number [No], the awarding agency [Agency], and the starting

[10]We assume that the *quarter* system rather than the *semester* system is used in this university.

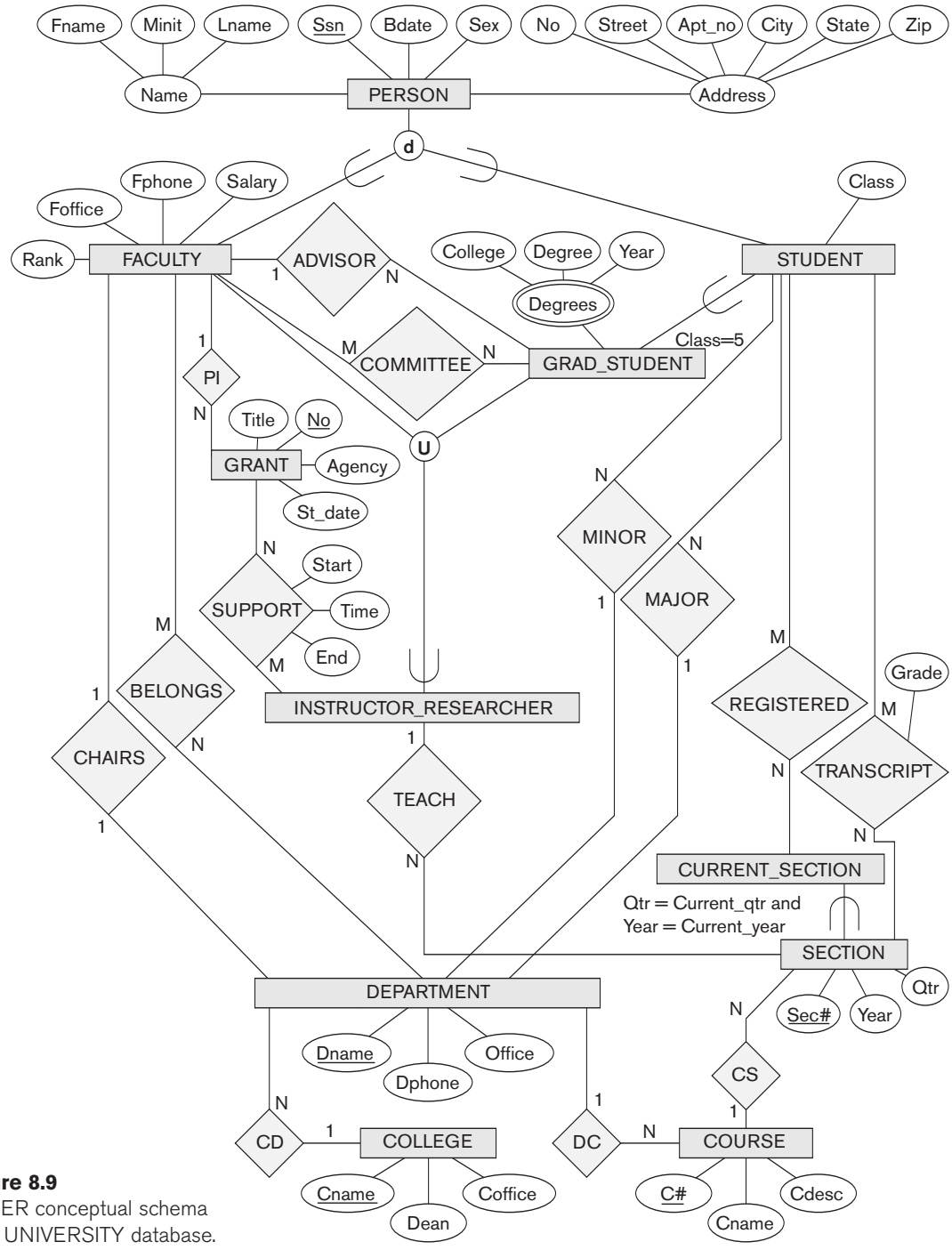


Figure 8.9
An EER conceptual schema for a UNIVERSITY database.

date [St_date]. A grant is related to one principal investigator [PI] and to all researchers it supports [SUPPORT]. Each instance of support has as attributes the starting date of support [Start], the ending date of the support (if known) [End], and the percentage of time being spent on the project [Time] by the researcher being supported.

8.5.2 Design Choices for Specialization/Generalization

It is not always easy to choose the most appropriate conceptual design for a database application. In Section 7.7.3, we presented some of the typical issues that confront a database designer when choosing among the concepts of entity types, relationship types, and attributes to represent a particular miniworld situation as an ER schema. In this section, we discuss design guidelines and choices for the EER concepts of specialization/generalization and categories (union types).

As we mentioned in Section 7.7.3, conceptual database design should be considered as an iterative refinement process until the most suitable design is reached. The following guidelines can help to guide the design process for EER concepts:

- In general, many specializations and subclasses can be defined to make the conceptual model accurate. However, the drawback is that the design becomes quite cluttered. It is important to represent only those subclasses that are deemed necessary to avoid extreme cluttering of the conceptual schema.
- If a subclass has few specific (local) attributes and no specific relationships, it can be merged into the superclass. The specific attributes would hold NULL values for entities that are not members of the subclass. A *type* attribute could specify whether an entity is a member of the subclass.
- Similarly, if all the subclasses of a specialization/generalization have few specific attributes and no specific relationships, they can be merged into the superclass and replaced with one or more *type* attributes that specify the subclass or subclasses that each entity belongs to (see Section 9.2 for how this criterion applies to relational databases).
- Union types and categories should generally be avoided unless the situation definitely warrants this type of construct, which does occur in some practical situations. If possible, we try to model using specialization/generalization as discussed at the end of Section 8.4.
- The choice of disjoint/overlapping and total/partial constraints on specialization/generalization is driven by the rules in the miniworld being modeled. If the requirements do not indicate any particular constraints, the default would generally be overlapping and partial, since this does not specify any restrictions on subclass membership.

As an example of applying these guidelines, consider Figure 8.6, where no specific (local) attributes are shown. We could merge all the subclasses into the EMPLOYEE entity type, and add the following attributes to EMPLOYEE:

- An attribute Job_type whose value set {'Secretary', 'Engineer', 'Technician'} would indicate which subclass in the first specialization each employee belongs to.
- An attribute Pay_method whose value set {'Salaried', 'Hourly'} would indicate which subclass in the second specialization each employee belongs to.
- An attribute Is_a_manager whose value set {'Yes', 'No'} would indicate whether an individual employee entity is a manager or not.

8.5.3 Formal Definitions for the EER Model Concepts

We now summarize the EER model concepts and give formal definitions. A **class**[11] is a set or collection of entities; this includes any of the EER schema constructs of group entities, such as entity types, subclasses, superclasses, and categories. A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass** (or **IS-A**) **relationship**. We denote such a relationship by C/S. For such a superclass/subclass relationship, we must always have

$$S \subseteq C$$

A **specialization** $Z = \{S_1, S_2, \ldots, S_n\}$ is a set of subclasses that have the same superclass G; that is, G/S_i is a superclass/subclass relationship for $i = 1, 2, \ldots, n$. G is called a **generalized entity type** (or the **superclass** of the specialization, or a **generalization** of the subclasses $\{S_1, S_2, \ldots, S_n\}$). Z is said to be **total** if we always (at any point in time) have

$$\bigcup_{i=1}^{n} S_i = G$$

Otherwise, Z is said to be **partial**. Z is said to be **disjoint** if we always have

$$S_i \cap S_j = \varnothing \text{ (empty set) for } i \neq j$$

Otherwise, Z is said to be **overlapping**.

A subclass S of C is said to be **predicate-defined** if a predicate p on the attributes of C is used to specify which entities in C are members of S; that is, $S = C[p]$, where $C[p]$ is the set of entities in C that satisfy p. A subclass that is not defined by a predicate is called **user-defined**.

A specialization Z (or generalization G) is said to be **attribute-defined** if a predicate $(A = c_i)$, where A is an attribute of G and c_i is a constant value from the domain of A,

[11]The use of the word *class* here differs from its more common use in object-oriented programming languages such as C++. In C++, a class is a structured type definition along with its applicable functions (operations).

is used to specify membership in each subclass S_i in Z. Notice that if $c_i \neq c_j$ for $i \neq j$, and A is a single-valued attribute, then the specialization will be disjoint.

A **category** T is a class that is a subset of the union of n defining superclasses $D_1, D_2, \ldots, D_n$, $n > 1$, and is formally specified as follows:

$$T \subseteq (D_1 \cup D_2 \ldots \cup D_n)$$

A predicate p_i on the attributes of D_i can be used to specify the members of each D_i that are members of T. If a predicate is specified on every D_i, we get

$$T = (D_1[p_1] \cup D_2[p_2] \ldots \cup D_n[p_n])$$

We should now extend the definition of **relationship type** given in Chapter 7 by allowing any class—not only any entity type—to participate in a relationship. Hence, we should replace the words *entity type* with *class* in that definition. The graphical notation of EER is consistent with ER because all classes are represented by rectangles.

8.6 Example of Other Notation: Representing Specialization and Generalization in UML Class Diagrams

We now discuss the UML notation for generalization/specialization and inheritance. We already presented basic UML class diagram notation and terminology in Section 7.8. Figure 8.10 illustrates a possible UML class diagram corresponding to the EER diagram in Figure 8.7. The basic notation for specialization/generalization (see Figure 8.10) is to connect the subclasses by vertical lines to a horizontal line, which has a triangle connecting the horizontal line through another vertical line to the superclass. A blank triangle indicates a specialization/generalization with the *disjoint* constraint, and a filled triangle indicates an *overlapping* constraint. The root superclass is called the **base class**, and the subclasses (leaf nodes) are called **leaf classes**.

The above discussion and example in Figure 8.10, and the presentation in Section 7.8 gave a brief overview of UML class diagrams and terminology. We focused on the concepts that are relevant to ER and EER database modeling, rather than those concepts that are more relevant to software engineering. In UML, there are many details that we have not discussed because they are outside the scope of this book and are mainly relevant to software engineering. For example, classes can be of various types:

- Abstract classes define attributes and operations but do not have objects corresponding to those classes. These are mainly used to specify a set of attributes and operations that can be inherited.
- Concrete classes can have objects (entities) instantiated to belong to the class.
- Template classes specify a template that can be further used to define other classes.

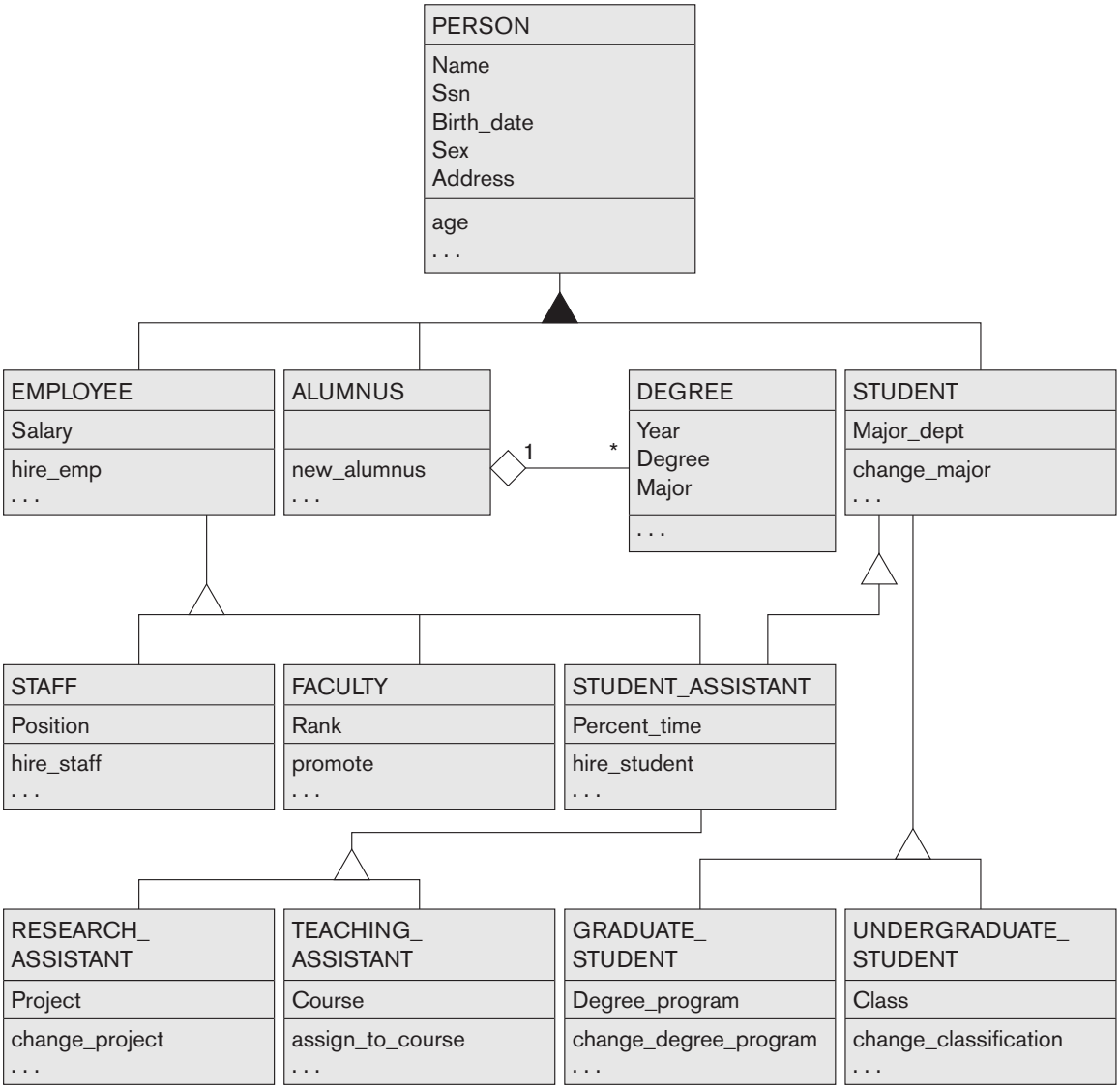


Figure 8.10
A UML class diagram corresponding to the EER diagram in Figure 8.7, illustrating UML notation for specialization/generalization.

In database design, we are mainly concerned with specifying concrete classes whose collections of objects are permanently (or persistently) stored in the database. The bibliographic notes at the end of this chapter give some references to books that describe complete details of UML. Additional material related to UML is covered in Chapter 10.

8.7 Data Abstraction, Knowledge Representation, and Ontology Concepts

In this section we discuss in general terms some of the modeling concepts that we described quite specifically in our presentation of the ER and EER models in Chapter 7 and earlier in this chapter. This terminology is not only used in conceptual data modeling but also in artificial intelligence literature when discussing **knowledge representation** (**KR**). This section discusses the similarities and differences between conceptual modeling and knowledge representation, and introduces some of the alternative terminology and a few additional concepts.

The goal of KR techniques is to develop concepts for accurately modeling some **domain of knowledge** by creating an **ontology**[12] that describes the concepts of the domain and how these concepts are interrelated. Such an ontology is used to store and manipulate knowledge for drawing inferences, making decisions, or answering questions. The goals of KR are similar to those of semantic data models, but there are some important similarities and differences between the two disciplines:

- Both disciplines use an abstraction process to identify common properties and important aspects of objects in the miniworld (also known as *domain of discourse* in KR) while suppressing insignificant differences and unimportant details.
- Both disciplines provide concepts, relationships, constraints, operations, and languages for defining data and representing knowledge.
- KR is generally broader in scope than semantic data models. Different forms of knowledge, such as rules (used in inference, deduction, and search), incomplete and default knowledge, and temporal and spatial knowledge, are represented in KR schemes. Database models are being expanded to include some of these concepts (see Chapter 26).
- KR schemes include **reasoning mechanisms** that deduce additional facts from the facts stored in a database. Hence, whereas most current database systems are limited to answering direct queries, knowledge-based systems using KR schemes can answer queries that involve **inferences** over the stored data. Database technology is being extended with inference mechanisms (see Section 26.5).
- Whereas most data models concentrate on the representation of database schemas, or meta-knowledge, KR schemes often mix up the schemas with the instances themselves in order to provide flexibility in representing exceptions. This often results in inefficiencies when these KR schemes are implemented, especially when compared with databases and when a large amount of data (facts) needs to be stored.

[12]An *ontology* is somewhat similar to a conceptual schema, but with more knowledge, rules, and exceptions.

We now discuss four **abstraction concepts** that are used in semantic data models, such as the EER model as well as in KR schemes: (1) classification and instantiation, (2) identification, (3) specialization and generalization, and (4) aggregation and association. The paired concepts of classification and instantiation are inverses of one another, as are generalization and specialization. The concepts of aggregation and association are also related. We discuss these abstract concepts and their relation to the concrete representations used in the EER model to clarify the data abstraction process and to improve our understanding of the related process of conceptual schema design. We close the section with a brief discussion of *ontology*, which is being used widely in recent knowledge representation research.

8.7.1 Classification and Instantiation

The process of **classification** involves systematically assigning similar objects/entities to object classes/entity types. We can now describe (in DB) or reason about (in KR) the classes rather than the individual objects. Collections of objects that share the same types of attributes, relationships, and constraints are classified into classes in order to simplify the process of discovering their properties. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class. An object instance is related to its object class by the **IS-AN-INSTANCE-OF** or **IS-A-MEMBER-OF** relationship. Although EER diagrams do not display instances, the UML diagrams allow a form of instantiation by permitting the display of individual objects. We *did not* describe this feature in our introduction to UML class diagrams.

In general, the objects of a class should have a similar type structure. However, some objects may display properties that differ in some respects from the other objects of the class; these **exception objects** also need to be modeled, and KR schemes allow more varied exceptions than do database models. In addition, certain properties apply to the class as a whole and not to the individual objects; KR schemes allow such **class properties**. UML diagrams also allow specification of class properties.

In the EER model, entities are classified into entity types according to their basic attributes and relationships. Entities are further classified into subclasses and categories based on additional similarities and differences (exceptions) among them. Relationship instances are classified into relationship types. Hence, entity types, subclasses, categories, and relationship types are the different concepts that are used for classification in the EER model. The EER model does not provide explicitly for class properties, but it may be extended to do so. In UML, objects are classified into classes, and it is possible to display both class properties and individual objects.

Knowledge representation models allow multiple classification schemes in which one class is an *instance* of another class (called a **meta-class**). Notice that this *cannot* be represented directly in the EER model, because we have only two levels—classes and instances. The only relationship among classes in the EER model is a superclass/subclass relationship, whereas in some KR schemes an additional class/instance relationship can be represented directly in a class hierarchy. An instance may itself be another class, allowing multiple-level classification schemes.

8.7.2 Identification

Identification is the abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier**. For example, a class name uniquely identifies a whole class within a schema. An additional mechanism is necessary for telling distinct object instances apart by means of object identifiers. Moreover, it is necessary to identify multiple manifestations in the database of the same real-world object. For example, we may have a tuple <'Matthew Clarke', '610618', '376-9821'> in a PERSON relation and another tuple <'301-54-0836', 'CS', 3.8> in a STUDENT relation that happen to represent the same real-world entity. There is no way to identify the fact that these two database objects (tuples) represent the same real-world entity unless we make a provision *at design time* for appropriate cross-referencing to supply this identification. Hence, identification is needed at two levels:

- To distinguish among database objects and classes
- To identify database objects and to relate them to their real-world counterparts

In the EER model, identification of schema constructs is based on a system of unique names for the constructs in a schema. For example, every class in an EER schema—whether it is an entity type, a subclass, a category, or a relationship type—must have a distinct name. The names of attributes of a particular class must also be distinct. Rules for unambiguously identifying attribute name references in a specialization or generalization lattice or hierarchy are needed as well.

At the object level, the values of key attributes are used to distinguish among entities of a particular entity type. For weak entity types, entities are identified by a combination of their own partial key values and the entities they are related to in the owner entity type(s). Relationship instances are identified by some combination of the entities that they relate to, depending on the cardinality ratio specified.

8.7.3 Specialization and Generalization

Specialization is the process of classifying a class of objects into more specialized subclasses. **Generalization** is the inverse process of generalizing several classes into a higher-level abstract class that includes the objects in all these classes. Specialization is conceptual refinement, whereas generalization is conceptual synthesis. Subclasses are used in the EER model to represent specialization and generalization. We call the relationship between a subclass and its superclass an **IS-A-SUBCLASS-OF** relationship, or simply an **IS-A** relationship. This is the same as the IS-A relationship discussed earlier in Section 8.5.3.

8.7.4 Aggregation and Association

Aggregation is an abstraction concept for building composite objects from their component objects. There are three cases where this concept can be related to the EER model. The first case is the situation in which we aggregate attribute values of

an object to form the whole object. The second case is when we represent an aggregation relationship as an ordinary relationship. The third case, which the EER model does not provide for explicitly, involves the possibility of combining objects that are related by a particular relationship instance into a *higher-level aggregate object.* This is sometimes useful when the higher-level aggregate object is itself to be related to another object. We call the relationship between the primitive objects and their aggregate object **IS-A-PART-OF**; the inverse is called **IS-A-COMPONENT-OF**. UML provides for all three types of aggregation.

The abstraction of **association** is used to associate objects from several *independent classes.* Hence, it is somewhat similar to the second use of aggregation. It is represented in the EER model by relationship types, and in UML by associations. This abstract relationship is called **IS-ASSOCIATED-WITH**.

In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 8.11(a), which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) Cname (company name) and Caddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes Contact_name and Contact_phone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, whereas others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 8.11(b) is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 8.11(c) is *not allowed* because the ER model does not allow relationships among relationships.

One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 8.11(d). Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object**. Other models treat entity types and relationship types uniformly and hence permit relationships among relationships, as illustrated in Figure 8.11(c).

To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 8.11(e), and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of aggregation, as in Figure 8.11(d), or to allow relationships among relationships, as in Figure 8.11(c).

The main structural distinction between aggregation and association is that when an association instance is deleted, the participating objects may continue to exist. However, if we support the notion of an aggregate object—for example, a CAR that is made up of objects ENGINE, CHASSIS, and TIRES—then deleting the aggregate CAR object amounts to deleting all its component objects.

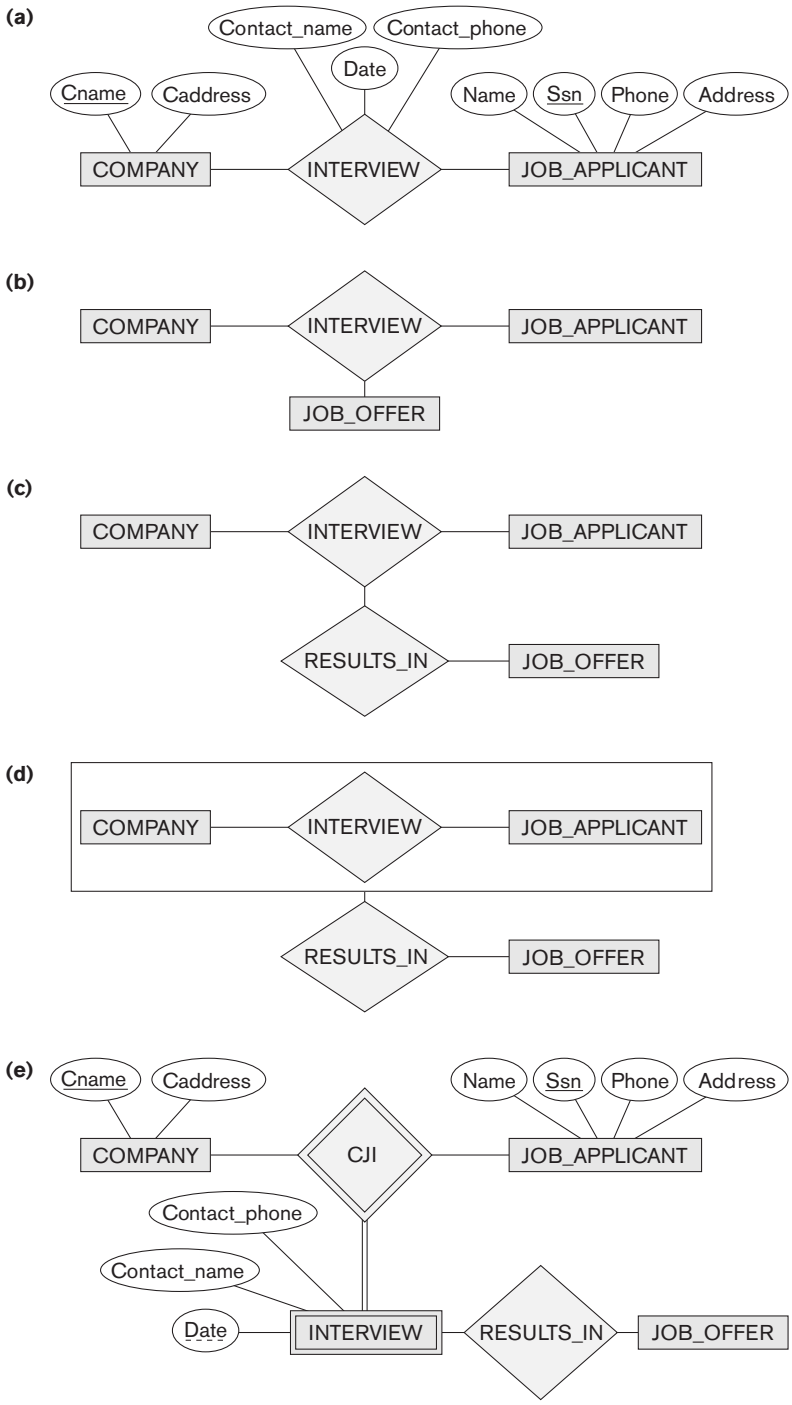


Figure 8.11
Aggregation. (a) The relationship type INTERVIEW. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Having the RESULTS_IN relationship participate in other relationships (not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally not allowed in ER but allowed by some modeling tools). (e) Correct representation in ER.

8.7.5 Ontologies and the Semantic Web

In recent years, the amount of computerized data and information available on the Web has spiraled out of control. Many different models and formats are used. In addition to the database models that we present in this book, much information is stored in the form of **documents**, which have considerably less structure than database information does. One ongoing project that is attempting to allow information exchange among computers on the Web is called the **Semantic Web**, which attempts to create knowledge representation models that are quite general in order to allow meaningful information exchange and search among machines. The concept of *ontology* is considered to be the most promising basis for achieving the goals of the Semantic Web and is closely related to knowledge representation. In this section, we give a brief introduction to what ontology is and how it can be used as a basis to automate information understanding, search, and exchange.

The study of ontologies attempts to describe the structures and relationships that are possible in reality through some common vocabulary; therefore, it can be considered as a way to describe the knowledge of a certain community about reality. Ontology originated in the fields of philosophy and metaphysics. One commonly used definition of **ontology** is *a specification* of a *conceptualization*.[13]

In this definition, a **conceptualization** is the set of concepts that are used to represent the part of reality or knowledge that is of interest to a community of users. **Specification** refers to the language and vocabulary terms that are used to specify the conceptualization. The ontology includes both *specification* and *conceptualization.* For example, the same conceptualization may be specified in two different languages, giving two separate ontologies. Based on this quite general definition, there is no consensus on what an ontology is exactly. Some possible ways to describe ontologies are as follows:

- A **thesaurus** (or even a **dictionary** or a **glossary** of terms) describes the relationships between words (vocabulary) that represent various concepts.
- A **taxonomy** describes how concepts of a particular area of knowledge are related using structures similar to those used in a specialization or generalization.
- A detailed **database schema** is considered by some to be an ontology that describes the concepts (entities and attributes) and relationships of a miniworld from reality.
- A **logical theory** uses concepts from mathematical logic to try to define concepts and their interrelationships.

Usually the concepts used to describe ontologies are quite similar to the concepts we discussed in conceptual modeling, such as entities, attributes, relationships, specializations, and so on. The main difference between an ontology and, say, a database schema, is that the schema is usually limited to describing a small subset of a mini-

[13]This definition is given in Gruber (1995).

chapter 9

Relational Database Design by ER- and EER-to-Relational Mapping

This chapter discusses how to **design a relational database schema** based on a conceptual schema design. Figure 7.1 presented a high-level view of the database design process, and in this chapter we focus on the logical database design or data model mapping step of database design. We present the procedures to create a relational schema from an Entity-Relationship (ER) or an Enhanced ER (EER) schema. Our discussion relates the constructs of the ER and EER models, presented in Chapters 7 and 8, to the constructs of the relational model, presented in Chapters 3 through 6. Many computer-aided software engineering (CASE) tools are based on the ER or EER models, or other similar models, as we have discussed in Chapters 7 and 8. Many tools use ER or EER diagrams or variations to develop the schema graphically, and then convert it automatically into a relational database schema in the DDL of a specific relational DBMS by employing algorithms similar to the ones presented in this chapter.

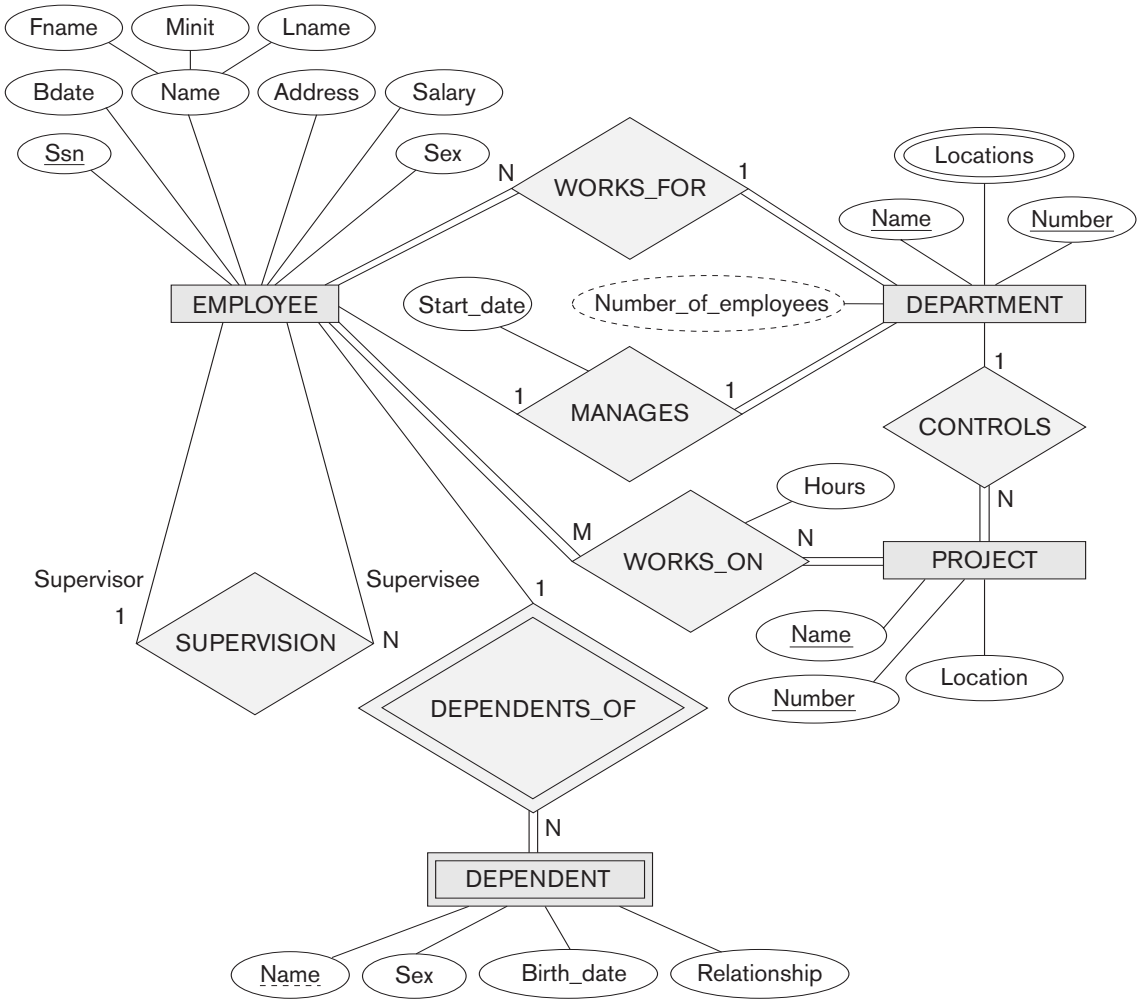
We outline a seven-step algorithm in Section 9.1 to convert the basic ER model constructs—entity types (strong and weak), binary relationships (with various structural constraints), *n*-ary relationships, and attributes (simple, composite, and multivalued)—into relations. Then, in Section 9.2, we continue the mapping algorithm by describing how to map EER model constructs—specialization/generalization and union types (categories)—into relations. Section 9.3 summarizes the chapter.

9.1 Relational Database Design Using ER-to-Relational Mapping

9.1.1 ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 9.1, and the corresponding COMPANY relational database schema is shown in Figure 9.2 to illustrate the map-

Figure 9.1
The ER conceptual schema diagram for the COMPANY database.



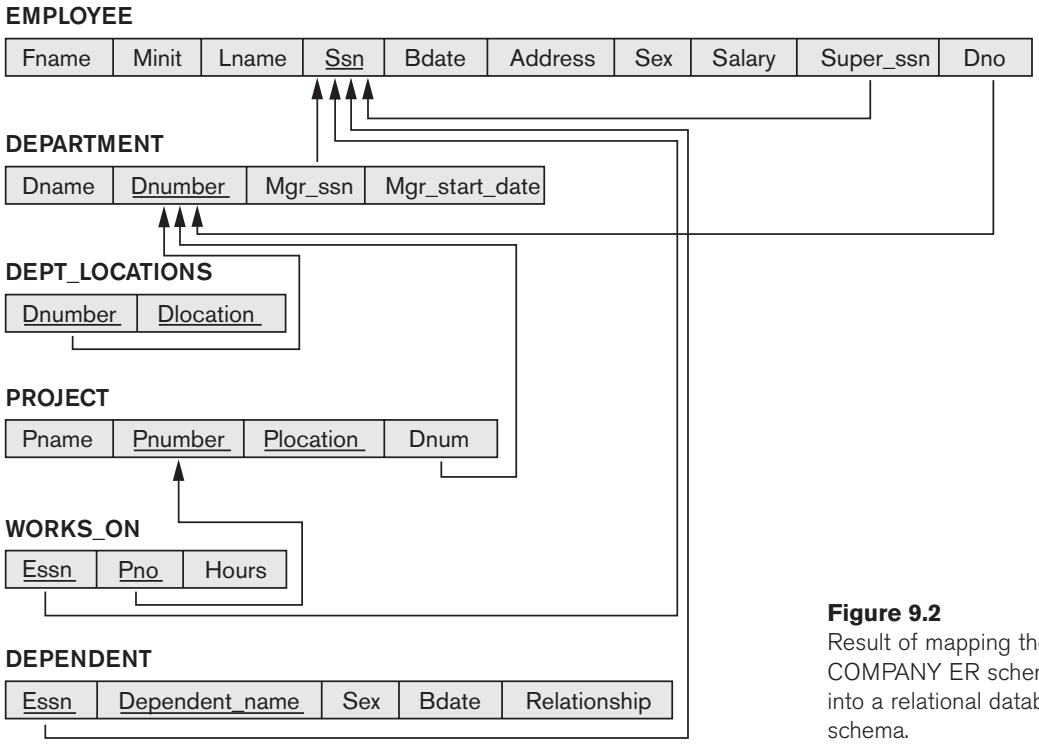


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

ping steps. We assume that the mapping will create tables with simple single-valued attributes. The relational model constraints defined in Chapter 3, which include primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.

If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R. Knowledge about keys is also kept for indexing purposes and other types of analyses.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the

attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are secondary keys is kept for possible use later in the design.

The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance. The result after this mapping step is shown in Figure 9.3(a).

Step 2: Mapping of Weak Entity Types. For each weak entity type *W* in the ER schema with owner entity type *E*, create a relation *R* and include all simple attributes (or simple components of composite attributes) of *W* as attributes of *R*. In addition, include as foreign key attributes of *R*, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of *W*. The primary key of *R* is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type *W*, if any.

If there is a weak entity type E_2 whose owner is also a weak entity type E_1, then E_1 should be mapped before E_2 to determine its primary key first.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT (see Figure 9.3(b)). We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary.

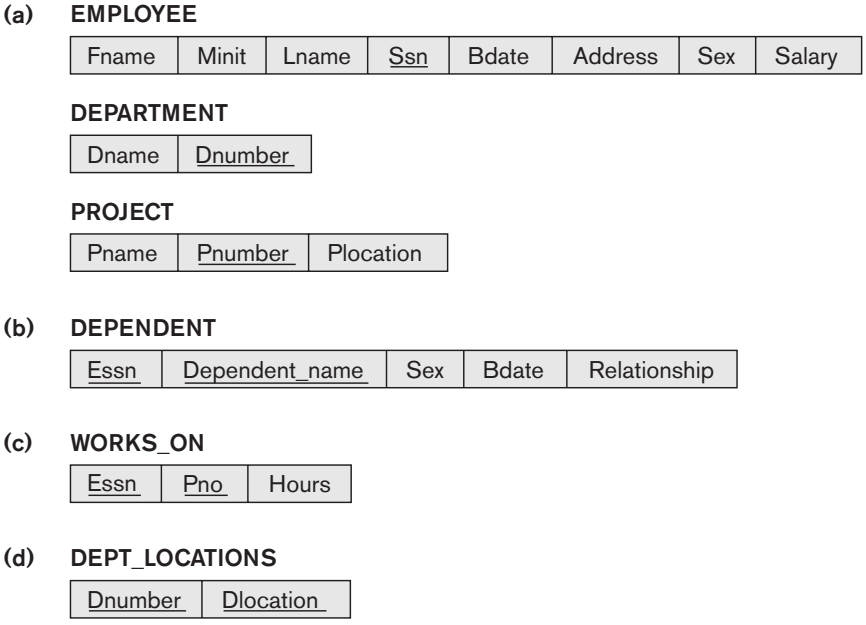


Figure 9.3
Illustration of some mapping steps. (a) *Entity* relations after step 1. (b) Additional *weak entity* relation after step 2. (c) *Relationship* relation after step 5. (d) Relation representing multivalued attribute after step 6.

The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}, because Dependent_name (also renamed from Name in Figure 9.1) is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action (see Section 4.2) on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type *R* in the ER schema, identify the relations *S* and *T* that correspond to the entity types participating in *R*. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross-reference or relationship relation approach. The first approach is the most useful and should be followed unless special conditions exist, as we discuss below.

1. **Foreign key approach:** Choose one of the relations—*S*, say—and include as a foreign key in *S* the primary key of *T*. It is better to choose an entity type with *total participation* in *R* in the role of *S*. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type *R* as attributes of *S*.

   In our example, we map the 1:1 relationship type MANAGES from Figure 9.1 by choosing the participating entity type DEPARTMENT to serve in the role of *S* because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date (see Figure 9.2).

   Note that it is possible to include the primary key of *S* as a foreign key in *T* instead. In our example, this amounts to having a foreign key attribute, say Department_managed in the EMPLOYEE relation, but it will have a NULL value for employee tuples who do not manage a department. If only 2 percent of employees manage a department, then 98 percent of the foreign keys would be NULL in this case. Another possibility is to have foreign keys in both relations *S* and *T* redundantly, but this creates redundancy and incurs a penalty for consistency maintenance.
2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when *both participations are total,* as this would indicate that the two tables will have the exact same number of tuples at all times.
3. **Cross-reference or relationship relation approach:** The third option is to set up a third relation *R* for the purpose of cross-referencing the primary keys of the two relations *S* and *T* representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation *R* is called a **relationship relation** (or sometimes a **lookup table**), because each

tuple in *R* represents a relationship instance that relates one tuple from *S* with one tuple from *T*. The relation *R* will include the primary key attributes of *S* and *T* as foreign keys to *S* and *T*. The primary key of *R* will be one of the two foreign keys, and the other foreign key will be a unique key of *R*. The drawback is having an extra relation, and requiring an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type *R*, identify the relation *S* that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in *S* the primary key of the relation *T* that represents the other entity type participating in *R*; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of *S*.

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 9.1. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation. These foreign keys are shown in Figure 9.2.

An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation *R* whose attributes are the primary keys of *S* and *T*, which will also be foreign keys to *S* and *T*. The primary key of *R* is the same as the primary key of *S*. This option can be used if few tuples in *S* participate in the relationship to avoid excessive NULL values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. For each binary M:N relationship type *R*, create a new relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types; their *combination* will form the primary key of *S*. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of *S*. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation S*.

In our example, we map the M:N relationship type WORKS_ON from Figure 9.1 by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively. We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}. This **relationship relation** is shown in Figure 9.3(c).

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) should be specified on the foreign keys in the relation corresponding to the relationship *R*, since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach, as we discussed earlier. This alternative is particularly useful when few relationship instances exist, in order to avoid NULL values in foreign keys. In this case, the primary key of the relationship relation will be *only one* of the foreign keys that reference the participating entity relations. For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation.

Step 6: Mapping of Multivalued Attributes. For each multivalued attribute *A*, create a new relation *R*. This relation *R* will include an attribute corresponding to *A*, plus the primary key attribute *K*—as a foreign key in *R*—of the relation that represents the entity type or relationship type that has *A* as a multivalued attribute. The primary key of *R* is the combination of *A* and *K*. If the multivalued attribute is composite, we include its simple components.

In our example, we create a relation DEPT_LOCATIONS (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, while Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}. A separate tuple will exist in DEPT_LOCATIONS for each location that a department has.

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) should be specified on the foreign key in the relation *R* corresponding to the multivalued attribute for both ON UPDATE and ON DELETE. We should also note that the key of *R* when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes. In some cases, when a multivalued attribute is composite, only some of the component attributes are required to be part of the key of *R*; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute (see Section 7.5).

Figure 9.2 shows the COMPANY relational database schema obtained with steps 1 through 6, and Figure 3.6 shows a sample database state. Notice that we did not yet discuss the mapping of *n*-ary relationship types ($n > 2$) because none exist in Figure 9.1; these are mapped in a similar way to M:N relationship types by including the following additional step in the mapping algorithm.

Step 7: Mapping of *N*-ary Relationship Types. For each *n*-ary relationship type *R*, where $n > 2$, create a new relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the *n*-ary relationship type (or

simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E (see the discussion in Section 7.9.2 concerning constraints on n-ary relationships).

For example, consider the relationship type SUPPLY in Figure 7.17. This can be mapped to the relation SUPPLY shown in Figure 9.4, whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

9.1.2 Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 summarizes the correspondences between ER and relational model constructs and constraints.

One of the main points to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having two attributes A and B, one a primary key and the other a foreign key (over the same domain) included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN operation (or NATURAL JOIN if the two join attributes have the same name) over $S.A$ and $T.B$, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or 1:N relationship type is involved, a single join operation is usually needed. For a binary M:N relationship type, two join operations are needed, whereas for n-ary relationship types, n joins are needed to fully materialize the relationship instances.

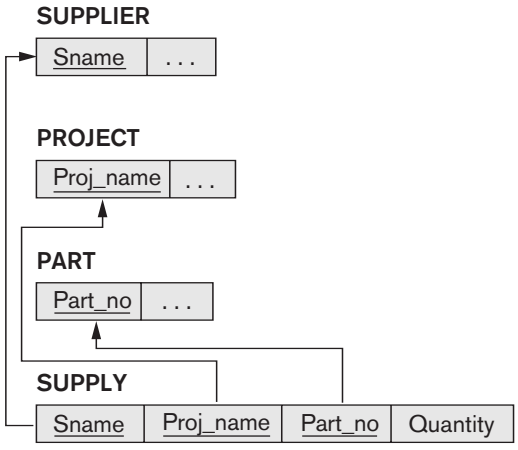


Figure 9.4
Mapping the n-ary relationship type SUPPLY from Figure 7.17(a).

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

For example, to form a relation that includes the employee name, project name, and hours that the employee works on each project, we need to connect each EMPLOYEE tuple to the related PROJECT tuples via the WORKS_ON relation in Figure 9.2. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition Ssn = Essn, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition Pno = Pnumber. In general, when multiple relationships need to be traversed, numerous join operations must be specified. A relational database user must always be aware of the foreign key attributes in order to use them correctly in combining related tuples from two or more relations. This is sometimes considered to be a drawback of the relational data model, because the foreign key/primary key correspondences are not always obvious upon inspection of relational schemas. If an EQUIJOIN is performed among attributes of two relations that do not represent a foreign key/primary key relationship, the result can often be meaningless and may lead to spurious data. For example, the reader can try joining the PROJECT and DEPT_LOCATIONS relations on the condition Dlocation = Plocation and examine the result (see the discussion of spurious tuples in Section 15.1.4).

In the relational schema we create a separate relation for *each* multivalued attribute. For a particular entity with a set of values for the multivalued attribute, the key attribute value of the entity is repeated once for each value of the multivalued attribute in a separate tuple because the basic relational model does *not* allow multiple values (a list, or a set of values) for an attribute in a single tuple. For example, because department 5 has three locations, three tuples exist in the DEPT_LOCATIONS relation in Figure 3.6; each tuple specifies one of the locations. In our example, we apply EQUIJOIN to DEPT_LOCATIONS and DEPARTMENT on the Dnumber attribute to get the values of all locations along with other DEPARTMENT attributes. In the resulting relation, the values of the other DEPARTMENT attributes are repeated in separate tuples for every location that a department has.

The basic relational algebra does not have a NEST or COMPRESS operation that would produce a set of tuples of the form {<'1', 'Houston'>, <'4', 'Stafford'>, <'5', {'Bellaire', 'Sugarland', 'Houston'}>} from the DEPT_LOCATIONS relation in Figure 3.6. This is a serious drawback of the basic normalized or *flat* version of the relational model. The object data model and object-relational systems (see Chapter 11) do allow multivalued attributes.

9.2 Mapping EER Model Constructs to Relations

Next, we discuss the mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm that was presented in Section 9.1.1.

9.2.1 Mapping of Specialization or Generalization

There are several options for mapping a number of subclasses that together form a specialization (or alternatively, that are generalized into a superclass), such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE in Figure 8.4. We can add a further step to our ER-to-relational mapping algorithm from Section 9.1.1, which has seven steps, to handle the mapping of specialization. Step 8, which follows, gives the most common options; other mappings are also possible. We discuss the conditions under which each option should be used. We use Attrs(R) to denote *the attributes of relation R*, and PK(R) to denote the *primary key of R*. First we describe the mapping formally, then we illustrate it with examples.

Step 8: Options for Mapping Specialization or Generalization. Convert each specialization with m subclasses $\{S_1, S_2, ..., S_m\}$ and (generalized) superclass C, where the attributes of C are $\{k, a_1, ...a_n\}$ and k is the (primary) key, into relation schemas using one of the following options:

- **Option 8A: Multiple relations—superclass and subclasses.** Create a relation L for C with attributes Attrs(L) = $\{k, a_1, ..., a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, $1 \le i \le m$, with the attributes Attrs(L_i) = $\{k\} \cup$ {attributes of S_i} and PK(L_i) = k. This option works for any specialization (total or partial, disjoint or overlapping).
- **Option 8B: Multiple relations—subclass relations only.** Create a relation L_i for each subclass S_i, $1 \le i \le m$, with the attributes Attrs(L_i) = {attributes of S_i} $\cup$ $\{k, a_1, ..., a_n\}$ and PK(L_i) = k. This option only works for a specialization whose subclasses are *total* (every entity in the superclass must belong to (at least) one of the subclasses). Additionally, it is only recommended if the specialization has the *disjointedness constraint* (see Section 8.3.1).If the specialization is *overlapping*, the same entity may be duplicated in several relations.
- **Option 8C: Single relation with one type attribute.** Create a single relation L with attributes Attrs(L) = $\{k, a_1, ..., a_n\}$ $\cup$ {attributes of S_1} $\cup$ ... $\cup$ {attributes of S_m} $\cup$ $\{t\}$ and PK(L) = k. The attribute t is called a **type** (or

discriminating) attribute whose value indicates the subclass to which each tuple belongs, if any. This option works only for a specialization whose subclasses are *disjoint,* and has the potential for generating many NULL values if many specific attributes exist in the subclasses.

- **Option 8D: Single relation with multiple type attributes.** Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, ..., a_n\} \cup \{\text{attributes of } S_1\} \cup ... \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, ..., t_m\}$ and $\text{PK}(L) = k$. Each t_i, $1 \leq i \leq m$, is a **Boolean type attribute** indicating whether a tuple belongs to subclass S_i. This option is used for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization).

Options 8A and 8B can be called the **multiple-relation options**, whereas options 8C and 8D can be called the **single-relation options**. Option 8A creates a relation L for the superclass C and its attributes, plus a relation L_i for each subclass S_i; each L_i includes the specific (or local) attributes of S_i, plus the primary key of the superclass C, which is propagated to L_i and becomes its primary key. It also becomes a foreign key to the superclass relation. An EQUIJOIN operation on the primary key between any L_i and L produces all the specific and inherited attributes of the entities in S_i. This option is illustrated in Figure 9.5(a) for the EER schema in Figure 8.4. Option 8A works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{<k>}(L_i) \subseteq \pi_{<k>}(L)$$

must hold for each L_i. This specifies a foreign key from each L_i to L, as well as an *inclusion dependency* $L_i.k < L.k$ (see Section 16.5).

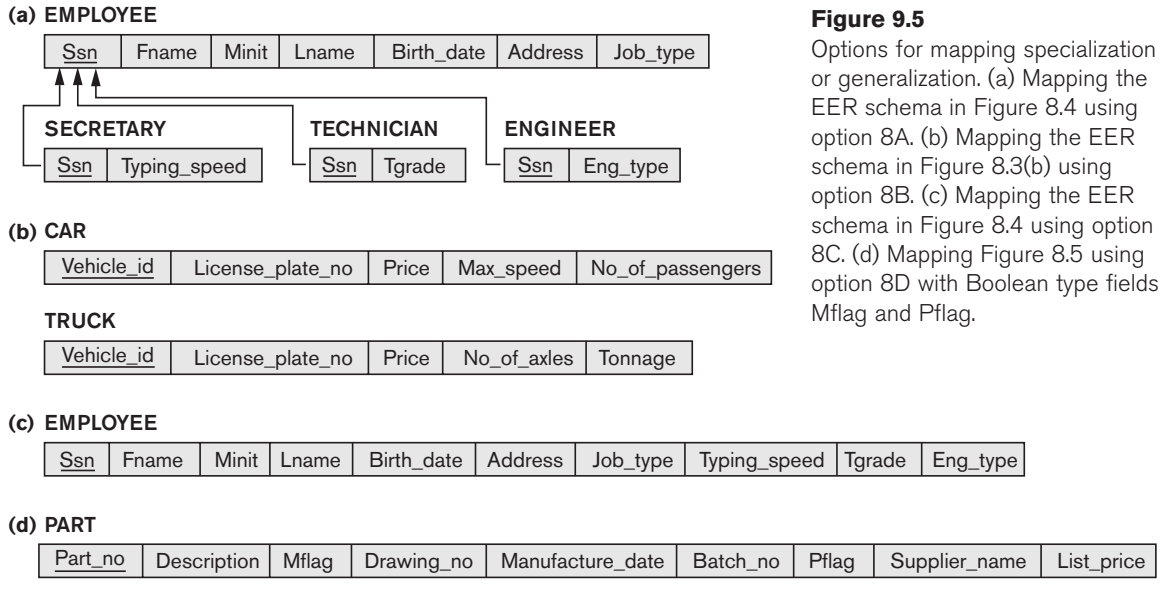


Figure 9.5
Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 8.4 using option 8A. (b) Mapping the EER schema in Figure 8.3(b) using option 8B. (c) Mapping the EER schema in Figure 8.4 using option 8C. (d) Mapping Figure 8.5 using option 8D with Boolean type fields Mflag and Pflag.

In option 8B, the EQUIJOIN operation between each subclass and the superclass is *built into* the schema and the relation L is done away with, as illustrated in Figure 9.5(b) for the EER specialization in Figure 8.3(b). This option works well only when *both* the disjoint and total constraints hold. If the specialization is not total, an entity that does not belong to any of the subclasses S_i is lost. If the specialization is not disjoint, an entity belonging to more than one subclass will have its inherited attributes from the superclass C stored redundantly in more than one L_i. With option 8B, no relation holds all the entities in the superclass C; consequently, we must apply an OUTER UNION (or FULL OUTER JOIN) operation (see Section 6.4) to the L_i relations to retrieve all the entities in C. The result of the outer union will be similar to the relations under options 8C and 8D except that the type fields will be missing. Whenever we search for an arbitrary entity in C, we must search all the m relations L_i.

Options 8C and 8D create a single relation to represent the superclass C and all its subclasses. An entity that does not belong to some of the subclasses will have NULL values for the specific attributes of these subclasses. These options are not recommended if many specific attributes are defined for the subclasses. If few specific subclass attributes exist, however, these mappings are preferable to options 8A and 8B because they do away with the need to specify EQUIJOIN and OUTER UNION operations; therefore, they can yield a more efficient implementation.

Option 8C is used to handle disjoint subclasses by including a single **type** (or **image** or **discriminating**) **attribute** t to indicate to which of the m subclasses each tuple belongs; hence, the domain of t could be {1, 2, ..., m}. If the specialization is partial, t can have NULL values in tuples that do not belong to any subclass. If the specialization is attribute-defined, that attribute serves the purpose of t and t is not needed; this option is illustrated in Figure 9.5(c) for the EER specialization in Figure 8.4.

Option 8D is designed to handle overlapping subclasses by including m *Boolean* **type** (or **flag**) fields, one for *each* subclass. It can also be used for disjoint subclasses. Each type field t_i can have a domain {yes, no}, where a value of yes indicates that the tuple is a member of subclass S_i. If we use this option for the EER specialization in Figure 8.4, we would include three types attributes—Is_a_secretary, Is_a_engineer, and Is_a_technician—instead of the Job_type attribute in Figure 9.5(c). Notice that it is also possible to create a single type attribute of m *bits* instead of the m type fields. Figure 9.5(d) shows the mapping of the specialization from Figure 8.5 using option 8D.

When we have a multilevel specialization (or generalization) hierarchy or lattice, we do not have to follow the same mapping option for all the specializations. Instead, we can use one mapping option for part of the hierarchy or lattice and other options for other parts. Figure 9.6 shows one possible mapping into relations for the EER lattice in Figure 8.6. Here we used option 8A for PERSON/{EMPLOYEE, ALUMNUS, STUDENT}, option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT} by including the type attribute Employee_type, and option 8D for STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ ASSISTANT} by including the type attributes Ta_flag and Ra_flag in EMPLOYEE, STUDENT/

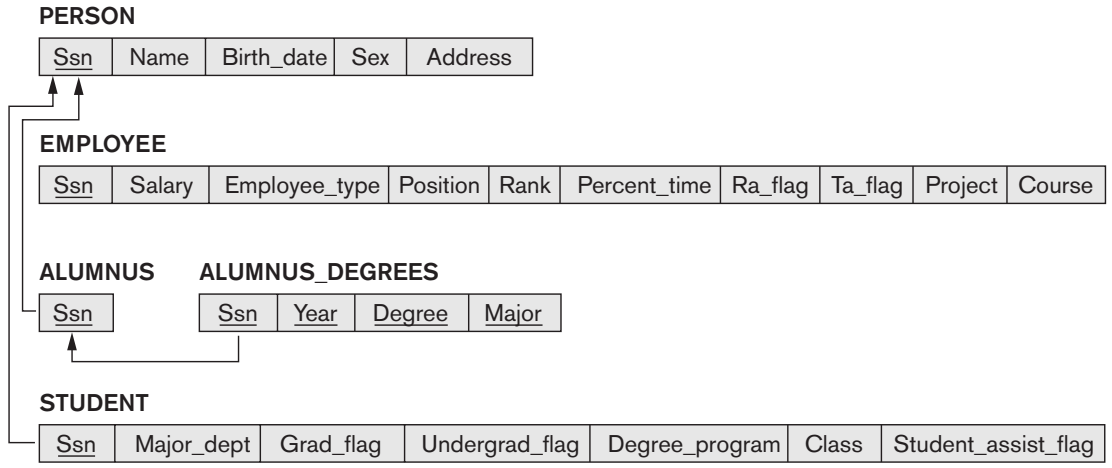


Figure 9.6
Mapping the EER specialization lattice in Figure 8.8 using multiple options.

STUDENT_ASSISTANT by including the type attributes Student_assist_flag in STUDENT, and STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} by including the type attributes Grad_flag and Undergrad_flag in STUDENT. In Figure 9.6, all attributes whose names end with *type* or *flag* are type fields.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

A shared subclass, such as ENGINEERING_MANAGER in Figure 8.6, is a subclass of several superclasses, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category (union type) as we discussed in Section 8.4. We can apply any of the options discussed in step 8 to a shared subclass, subject to the restrictions discussed in step 8 of the mapping algorithm. In Figure 9.6, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (Employee_type attribute) and option 8D is used in the STUDENT relation (Student_assist_flag attribute).

9.2.3 Mapping of Categories (Union Types)

We add another step to the mapping procedure—step 9—to handle categories. A category (or union type) is a subclass of the *union* of two or more superclasses that can have different keys because they can be of different entity types (see Section 8.4). An example is the OWNER category shown in Figure 8.8, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that figure, REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

Step 9: Mapping of Union Types (Categories). For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a **surrogate key**, when creating a relation to correspond to the category. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the category. In our example in Figure 8.8, we create a relation OWNER to correspond to the OWNER category, as illustrated in Figure 9.7, and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called Owner_id. We also include the surrogate key attribute Owner_id as foreign key in each relation corresponding to a superclass of the category, to specify the correspondence in values between the surrogate key and the key of each superclass. Notice that if a particular PERSON (or BANK or COMPANY) entity is not a member of OWNER, it would have a NULL value for its Owner_id attribute in its corresponding tuple in the PERSON (or BANK or COMPANY) relation, and it would not have a tuple in the OWNER relation. It is also recommended to add a type attribute (not shown in Figure 9.7) to the OWNER relation to indicate the particular entity type to which each tuple belongs (PERSON or BANK or COMPANY).

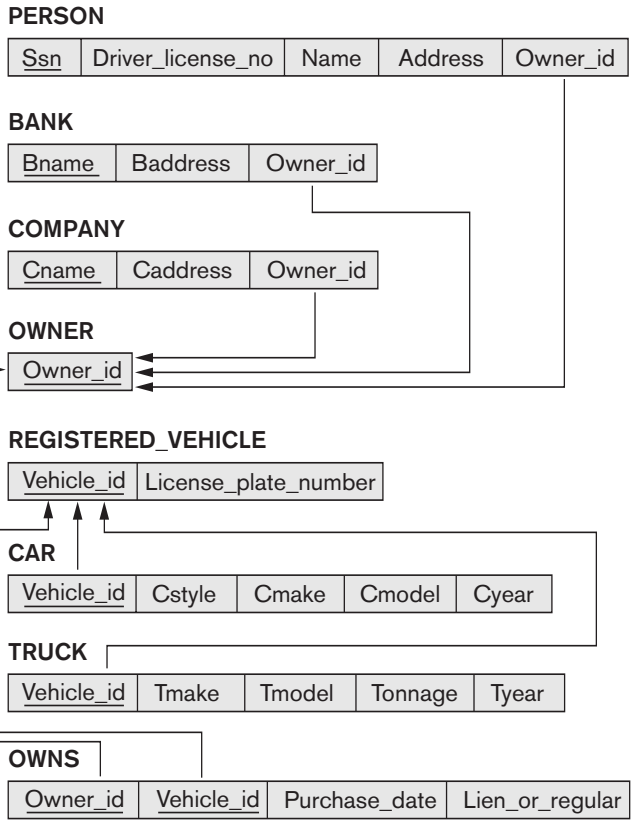


Figure 9.7
Mapping the EER categories (union types) in Figure 8.8 to relations.

chapter 25

Distributed Databases

In this chapter we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. Distributed databases bring the advantages of distributed computing to the database management domain. A **distributed computing system** consists of a number of processing elements, not necessarily homogeneous, that are interconnected by a computer network, and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The economic viability of this approach stems from two reasons: more computing power is harnessed to solve a complex task, and each autonomous processing element can be managed independently to develop its own applications.

DDB technology resulted from a merger of two technologies: database technology, and network and data communication technology. Computer networks allow distributed processing of data. Traditional databases, on the other hand, focus on providing centralized, controlled access to data. Distributed databases allow an integration of information and its processing by applications that may themselves be centralized or distributed.

Several distributed database prototype systems were developed in the 1980s to address the issues of data distribution, distributed query and transaction processing, distributed database metadata management, and other topics. However, a full-scale comprehensive DDBMS that implements the functionality and techniques proposed in DDB research never emerged as a commercially viable product. Most major vendors redirected their efforts from developing a *pure* DDBMS product into developing systems based on client-server concepts, or toward developing technologies for accessing distributed heterogeneous data sources.

Organizations continue to be interested in the *decentralization* of processing (at the system level) while achieving an *integration* of the information resources (at the logical level) within their geographically distributed systems of databases, applications, and users. There is now a general endorsement of the client-server approach to application development, and the three-tier approach to Web applications development (see Section 2.5).

In this chapter we discuss distributed databases, their architectural variations, and concepts central to data distribution and the management of distributed data. Details of the advances in communication technologies facilitating the development of DDBs are outside the scope of this book; see the texts on data communications and networking listed in the Selected Bibliography at the end of this chapter.

Section 25.1 introduces distributed database management and related concepts. Sections 25.2 and 25.3 introduce different types of distributed database systems and their architectures, including federated and multidatabase systems. The problems of heterogeneity and the needs of autonomy in federated database systems are also highlighted. Detailed issues of distributed database design, involving fragmenting of data and distributing it over multiple sites with possible replication, are discussed in Section 25.4. Sections 25.5 and 25.6 introduce distributed database query and transaction processing techniques, respectively. Section 25.7 gives an overview of the concurrency control and recovery in distributed databases. Section 25.8 discusses catalog management schemes in distributed databases. In Section 25.9, we briefly discuss current trends in distributed databases such as cloud computing and peer-to-peer databases. Section 25.10 discusses distributed database features of the Oracle RDBMS. Section 25.11 summarizes the chapter.

For a short introduction to the topic of distributed databases, Sections 25.1, 25.2, and 25.3 may be covered.

25.1 Distributed Database Concepts[1]

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.[2]

Distributed databases are different from Internet Web files. Web pages are basically a very large collection of files stored on different nodes in a network—the Internet—with interrelationships among the files represented via hyperlinks. The common functions of database management, including uniform query processing and transaction processing, *do not* apply to this scenario yet. The technology is, however, moving in a direction such that distributed World Wide Web (WWW) databases will become a reality in the future. We have discussed some of the issues of

[1]The substantial contribution of Narasimhan Srinivasan to this and several other sections in this chapter is appreciated.

[2]This definition and discussions in this section are based largely on Ozsu and Valduriez (1999).

accessing databases on the Web in Chapters 12 and 14. The proliferation of data at millions of Websites in various forms does *not* qualify as a DDB by the definition given earlier.

25.1.1 Differences between DDB and Multiprocessor Systems

We need to distinguish distributed databases from multiprocessor systems that use shared storage (primary memory or disk). For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **communication network** to transmit data and commands among sites, as shown later in Figure 25.3(c).
- **Logical interrelation of the connected databases.** It is essential that the information in the databases be logically related.
- **Absence of homogeneity constraint among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity—say, within the same building or a group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines or satellites. It is also possible to use a combination of networks.

Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of communication network exists among sites, regardless of any particular topology. We will not address any network-specific issues, although it is important to understand that for an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying communication network are invisible to the end user.

25.1.2 Transparency

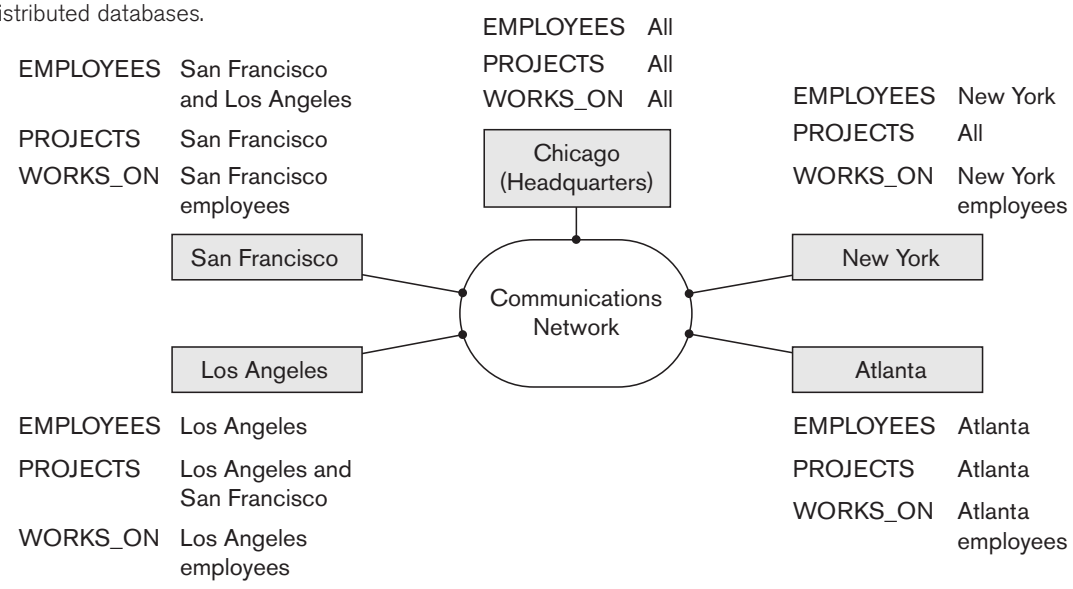
The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple

sites connected by a computer network, so additional types of transparencies are introduced.

Consider the company database in Figure 3.5 that we have been discussing throughout the book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows, as we will discuss in Section 25.4) and stored with possible replication as shown in Figure 25.1. The following types of transparencies are possible:

- **Data organization transparency (also known as *distribution* or *network transparency*).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.
- **Replication transparency.** As we show in Figure 25.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
- **Fragmentation transparency.** Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations

Figure 25.1
Data distribution and replication among distributed databases.



that are subsets of the tuples (rows) in the original relation. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. A global query by the user must be transformed into several fragment queries. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include **design transparency** and **execution transparency**—referring to freedom from knowing how the distributed database is designed and where a transaction executes.

25.1.3 Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.

25.1.4 Reliability and Availability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. **Errors** constitute that subset of system states that causes the failure. **Fault** is the cause of an error.

To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and designs mechanisms that can detect and remove faults before they can result in a system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components and processes user requests so long as database consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Communication failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.

25.1.5 Advantages of Distributed Databases

Organizations resort to distributed database management for various reasons. Some important advantages are listed below.

1. **Improved ease and flexibility of application development**. Developing and maintaining applications at geographically distributed sites of an organization is facilitated owing to transparency of data distribution and control.
2. **Increased reliability and availability.** This is achieved by the isolation of faults to their site of origin without affecting the other databases connected to the network. When the data and DDBMS software are distributed over several sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. This improves both reliability and availability. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site had been replicated at another site prior to the failure, then the user will not be affected at all.
3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. **Easier expansion**. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more processors is much easier.

The transparencies we discussed in Section 25.1.2 lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy**, which gives the users tighter control over local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations. Additionally, transparency impacts the features that must be provided by the operating system and the DBMS.

25.1.6 Additional Functions of Distributed Databases

Distribution leads to increased complexity in the system design and implementation. To achieve the potential advantages listed previously, the DDBMS software must be able to provide the following functions in addition to those of a centralized DBMS:

- **Keeping track of data distribution.** The ability to keep track of the data distribution, fragmentation, and replication by expanding the DDBMS catalog.
- **Distributed query processing.** The ability to access remote sites and transmit queries and data among the various sites via a communication network.
- **Distributed transaction management.** The ability to devise execution strategies for queries and transactions that access data from more than one site and to synchronize the access to distributed data and maintain the integrity of the overall database.
- **Replicated data management.** The ability to decide which copy of a replicated data item to access and to maintain the consistency of copies of a replicated data item.
- **Distributed database recovery.** The ability to recover from individual site crashes and from new types of failures, such as the failure of communication links.
- **Security.** Distributed transactions must be executed with the proper management of the security of the data and the authorization/access privileges of users.
- **Distributed directory (catalog) management.** A directory contains information (metadata) about data in the database. The directory may be global for the entire DDB, or local for each site. The placement and distribution of the directory are design and policy issues.

These functions themselves increase the complexity of a DDBMS over a centralized DBMS. Before we can realize the full potential advantages of distribution, we must find satisfactory solutions to these design issues and problems. Including all this additional functionality is hard to accomplish, and finding optimal solutions is a step beyond that.

25.2 Types of Distributed Database Systems

The term *distributed database management system* can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

Figure 25.2 shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is complete autonomy, but a total lack of distribution and heterogeneity (Point A in the figure). We see that the degree of local autonomy provides further ground for classification into federated and multidatabase systems. At one extreme of the autonomy spectrum, we have a DDBMS that *looks like* a centralized DBMS to the user, with zero autonomy (Point B). A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS—which means that no local autonomy exists. Along the autonomy axis we encounter two types of DDBMSs called *federated database system* (Point C) and *multidatabase system* (Point D). In such systems, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA, and hence has

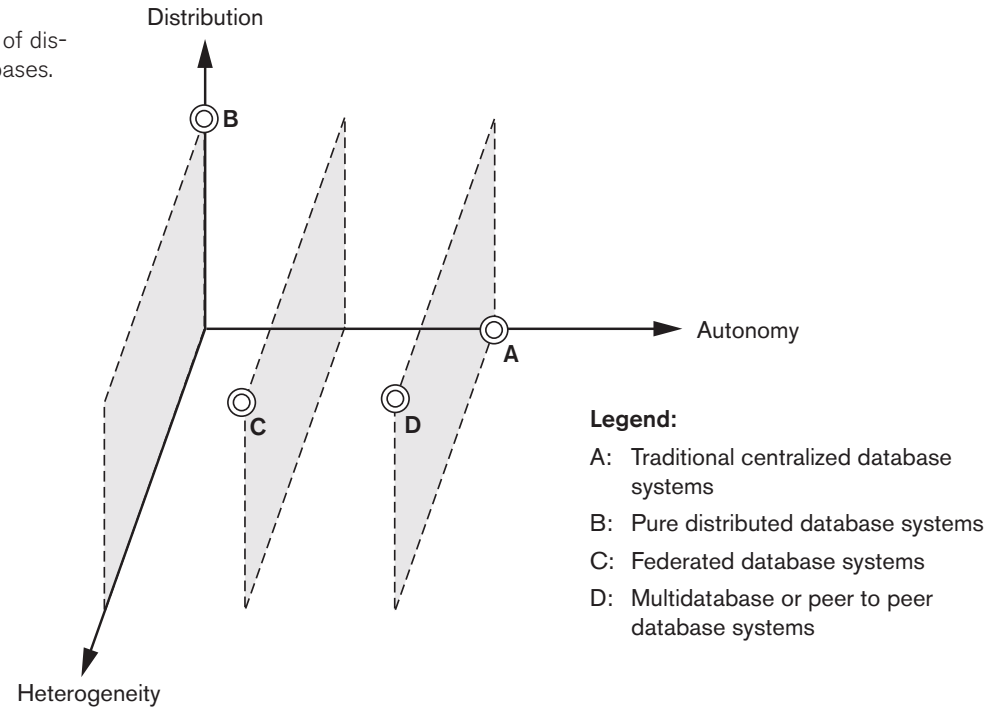


Figure 25.2
Classification of distributed databases.

a very high degree of *local autonomy*. The term **federated database system (FDBS)** is used when there is some global view or schema of the federation of databases that is shared by the applications (Point C). On the other hand, a **multidatabase system** has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (Point D).[3] Both systems are hybrids between distributed and centralized systems, and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense. Point D in the diagram may also stand for a system with full local autonomy and full heterogeneity—this could be a peer-to-peer database system (see Section 25.9.2). In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS (such as Computer Associates' IDMS or HP'S IMAGE/3000), and a third an object DBMS (such as Object Design's ObjectStore) or hierarchical DBMS (such as IBM's IMS); in such a case, it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server.

We briefly discuss the issues affecting the design of FDBSs next.

25.2.1 Federated Database Management Systems Issues

The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FDBS.

- **Differences in data models.** Databases in an organization come from a variety of data models, including the so-called legacy models (hierarchical and network, see Web Appendixes D and E), the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.
- **Differences in constraints.** Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.

[3]The term *multidatabase system* is not easily applicable to most enterprise IT environments. The notion of constructing a global schema as and when the need arises is not very feasible in practice for enterprise databases.

- **Differences in query languages.** Even with the same data model, the languages and their versions vary. For example, SQL has multiple versions like SQL-89, SQL-92, SQL-99, and SQL:2008, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

Semantic Heterogeneity. Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The **design autonomy** of component DBSs refers to their freedom of choosing the following design parameters, which in turn affect the eventual complexity of the FDBS:

- **The universe of discourse from which the data is drawn.** For example, for two customer accounts, databases in the federation may be from the United States and Japan and have entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations would also present a problem. Hence, relations in these two databases that have identical names—CUSTOMER or ACCOUNT—may have some common and some entirely distinct information.
- **Representation and naming.** The representation and naming of data elements and the structure of the data model may be prespecified for each local database.
- **The understanding, meaning, and subjective interpretation of data.** This is a chief contributor to semantic heterogeneity.
- **Transaction and policy constraints.** These deal with serializability criteria, compensating transactions, and other transaction policies.
- **Derivation of summaries.** Aggregation, summarization, and other data-processing features and operations supported by the system.

The above problems related to semantic heterogeneity are being faced by all major multinational and governmental organizations in all application areas. In today's commercial environment, most enterprises are resorting to heterogeneous FDBSs, having heavily invested in the development of individual database systems using diverse data models on different platforms over the last 20 to 30 years. Enterprises are using various forms of software—typically called the **middleware**, or Web-based packages called **application servers** (for example, WebLogic or WebSphere) and even generic systems, called **Enterprise Resource Planning (ERP) systems** (for example, SAP, J. D. Edwards ERP)—to manage the transport of queries and transactions from the global application to individual databases (with possible additional processing for business rules) and the data from the heterogeneous database servers to the global application. Detailed discussion of these types of software systems is outside the scope of this book.

Just as providing the ultimate transparency is the goal of any distributed database architecture, local component databases strive to preserve autonomy. **Communication autonomy** of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy**

refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

25.3 Distributed Database Architectures

In this section, we first briefly point out the distinction between parallel and distributed database architectures. While both are prevalent in industry today, there are various manifestations of the distributed architectures that are continuously evolving among large enterprises. The parallel architecture is more common in high-performance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications. We then introduce a generic architecture of a distributed database. This is followed by discussions on the architecture of three-tier client-server and federated database systems.

25.3.1 Parallel versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

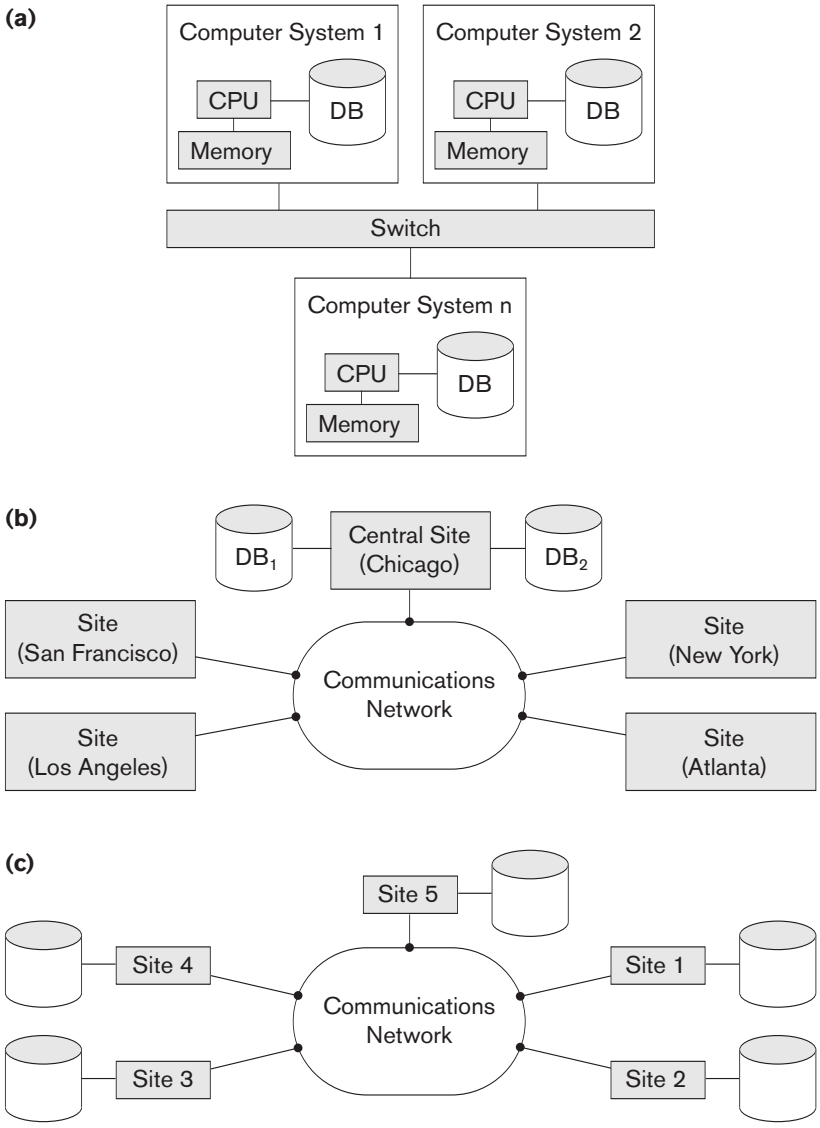
- **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
- **Shared disk (loosely coupled) architecture.** Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.[4] Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared nothing architecture**. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for

[4]If both primary and secondary memories are shared, the architecture is also known as *shared everything architecture*.

parallel databases. Figure 25.3a illustrates a parallel database (shared nothing), whereas Figure 25.3b illustrates a centralized database with distributed access and Figure 25.3c shows a pure distributed database. We will not expand on parallel architectures and related data management issues here.

Figure 25.3
Some different database system architectures. (a) Shared nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.



25.3.2 General Architecture of Pure Distributed Databases

In this section we discuss both the logical and component architectural models of a DDB. In Figure 25.4, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency (see Section 25.1.2). To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency discussed in Section 25.1.2. Figure 25.5 shows the component architecture of a DDB. It is an extension of its centralized counterpart (Figure 2.3) in Chapter 2. For the sake of simplicity, common elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints. The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU,

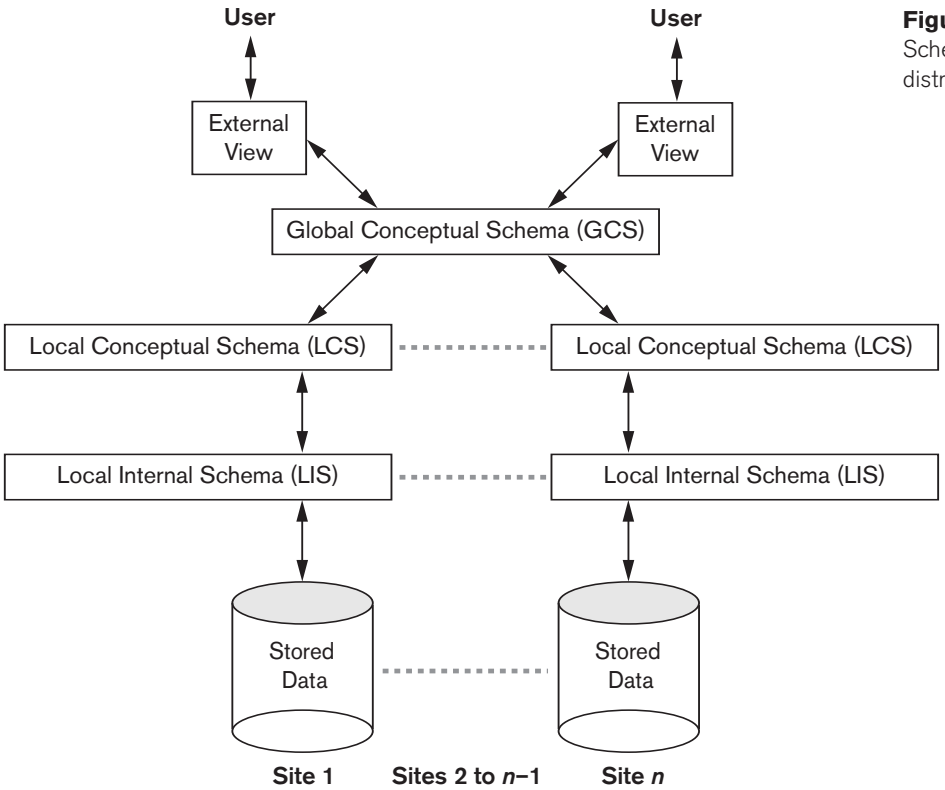


Figure 25.4
Schema architecture of distributed databases.

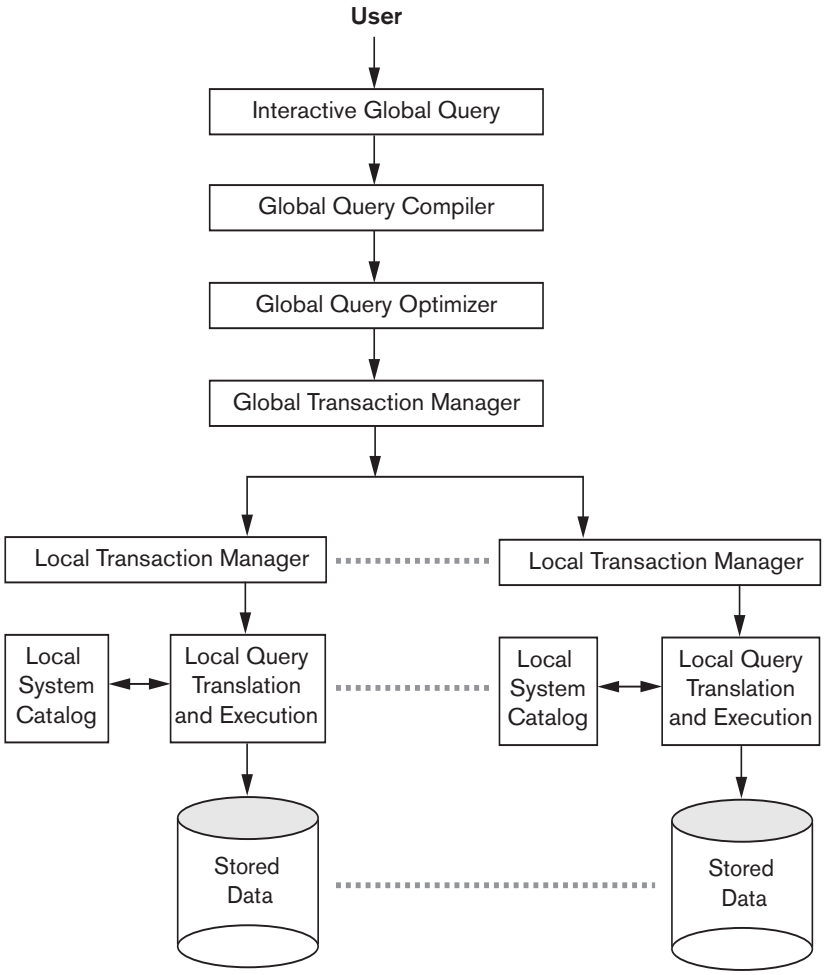


Figure 25.5
Component architecture of distributed databases.

I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution. Each local DBMS would have their local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

25.3.3 Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 25.6. In this architecture, the **local schema** is the

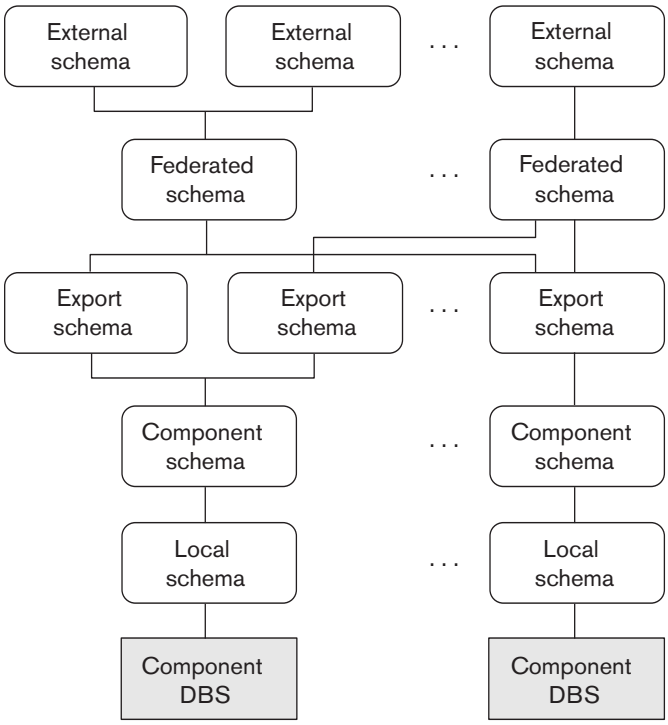


Figure 25.6
The five-level schema architecture in a federated database system (FDBS).

Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the schema for a user group or an application, as in the three-level schema architecture.[5]

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

[5]For a detailed discussion of the autonomies and the five-level architecture of FDBMSs, see Sheth and Larson (1990).

25.3.4 An Overview of Three-Tier Client-Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client-server architectures. We introduced the two-tier client-server architecture in Section 2.5. It is now more common to use a three-tier architecture, particularly in Web applications. This architecture is illustrated in Figure 25.7.

In the three-tier client-server architecture, the following three layers exist:

1. **Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. **Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such

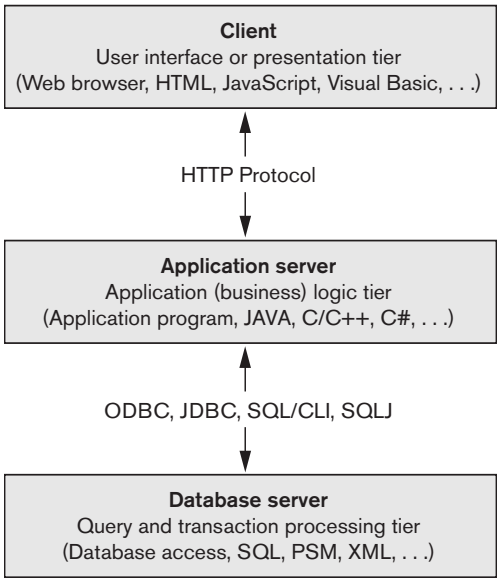


Figure 25.7
The three-tier client-server architecture.

as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

3. **Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 12) when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality between the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI (see Chapter 13).

In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 12), so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

25.4 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

In this section we discuss techniques that are used to break up the database into logical units, called **fragments**, which may be assigned for storage at the various sites. We also discuss the use of **data replication**, which permits certain data to be stored in more than one site, and the process of **allocating** fragments—or replicas of fragments—for storage at the various sites. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

25.4.1 Data Fragmentation

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is *no replication;* that is, each relation—or portion of a relation—is stored at one site only. We discuss replication and its effects later in this section. We also use the terminology of relational databases, but similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites. To illustrate our discussion, we use the relational database schema in Figure 3.5.

Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT in Figure 3.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 3.6, and assume there are three computer sites—one for each department in the company.[6]

We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

[6]Of course, in an actual situation, there will be many more tuples in the relation than those shown in Figure 3.6.

Horizontal Fragmentation. A **horizontal fragment** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved. For example, we may define three horizontal fragments on the EMPLOYEE relation in Figure 3.6 with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1)—each fragment contains the PROJECT tuples controlled by a particular department. **Horizontal fragmentation** divides a relation *horizontally* by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation. Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation "vertically" by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is *no common attribute* between the two fragments. It is necessary to include the primary key or some candidate key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Notice that each horizontal fragment on a relation R can be specified in the relational algebra by a $\sigma_{C_i}(R)$ operation. A set of horizontal fragments whose conditions $C_1, C_2, ..., C_n$ include all the tuples in R—that is, every tuple in R satisfies (C_1 OR C_2 OR ... OR C_n)—is called a **complete horizontal fragmentation** of R. In many cases a complete horizontal fragmentation is also **disjoint**; that is, no tuple in R satisfies (C_i AND C_j) for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a $\pi_{L_i}(R)$ operation in the relational algebra. A set of vertical fragments whose projection lists $L_1, L_2, ..., L_n$ include all the attributes in R but share only the primary key attribute of R is called a

complete vertical fragmentation of R. In this case the projection lists satisfy the following two conditions:

- $L_1 \cup L_2 \cup \ldots \cup L_n = \text{ATTRS}(R)$.
- $L_i \cap L_j = \text{PK}(R)$ for any $i \neq j$, where $\text{ATTRS}(R)$ is the set of attributes of R and $\text{PK}(R)$ is the primary key of R.

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists $L_1 = \{\text{Ssn, Name, Bdate, Address, Sex}\}$ and $L_2 = \{\text{Ssn, Salary, Super\_ssn, Dno}\}$ constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation in Figure 3.5 by the conditions (Salary > 50000) and (Dno = 4); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists $L_1 = \{\text{Name, Address}\}$ and $L_2 = \{\text{Ssn, Name, Salary}\}$; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) Fragmentation. We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If $C = \text{TRUE}$ (that is, all tuples are selected) and $L \neq \text{ATTRS}(R)$, we get a vertical fragment, and if $C \neq \text{TRUE}$ and $L = \text{ATTRS}(R)$, we get a horizontal fragment. Finally, if $C \neq \text{TRUE}$ and $L \neq \text{ATTRS}(R)$, we get a mixed fragment. Notice that a relation can itself be considered a fragment with $C = \text{TRUE}$ and $L = \text{ATTRS}(R)$. In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is

stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

25.4.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication, as we will see in Section 25.7.

The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database.[7] A description of the replication of fragments is sometimes called a **replication schema**.

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required, transactions can be submitted at any site, and most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

[7]For a proposed scalable approach to synchronize partially replicated databases, see Mahajan et al. (1998).

25.4.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database in Figures 3.5 and 3.6. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department.* Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super_ssn attributes of EMPLOYEE. Site 1 is used by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database in Figure 3.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively, in Figure 3.5. We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super_ssn, Dno}. Figure 25.8 shows the mixed fragments EMPD_5 and EMPD_4, which include the EMPLOYEE tuples satisfying the conditions Dno = 5 and Dno = 4, respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at headquarters—site 1.

We must now fragment the WORKS_ON relation and decide which fragments of WORKS_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS_ON relates an employee e to a project *P*. We could fragment WORKS_ON based on the department *D* in which e works *or* based on the department *D′* that controls *P*. Fragmentation becomes easy if we have a constraint stating that $D = D'$ for all WORKS_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database in Figure 3.6. For example, the WORKS_ON tuple <333445555, 10, 10.0> relates an employee who works for department 5 with a project controlled by department 4. In this case, we could fragment WORKS_ON based on the department in which the employee works (which is expressed by the condition *C*) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure 25.9.

In Figure 25.9, the union of fragments G_1, G_2, and G_3 gives all WORKS_ON tuples for employees who work for department 5. Similarly, the union of fragments G_4, G_5, and G_6 gives all WORKS_ON tuples for employees who work for department 4. On the other hand, the union of fragments G_1, G_4, and G_7 gives all WORKS_ON tuples for projects controlled by department 5. The condition for each of the fragments G_1 through G_9 is shown in Figure 25.9 The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution in Figure 25.8, we choose to include all fragments that can be joined to

Figure 25.8
Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

(a)

EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

Data at site 2

(b)

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

Essn	Pno	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

Data at site 3

Figure 25.9
Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=1)]).

(a) Employees in Department 5

G1

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G2

Essn	Pno	Hours
333445555	10	10.0

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

Essn	Pno	Hours
333445555	20	10.0

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(b) Employees in Department 4

G4

Essn	Pno	Hours

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

Essn	Pno	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

Essn	Pno	Hours
987654321	20	15.0

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(c) Employees in Department 1

G7

Essn	Pno	Hours

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

Essn	Pno	Hours

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

Essn	Pno	Hours
888665555	20	Null

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G_1, G_2, G_3, G_4, and G_7 at site 2 and the union of fragments G_4, G_5, G_6, G_2, and G_8 at site 3. Notice that fragments G_2 and G_4 are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.

25.5 Query Processing and Optimization in Distributed Databases

Now we give an overview of how a DDBMS processes and optimizes a query. First we discuss the steps involved in query processing and then elaborate on the communication costs of processing a distributed query. Finally we discuss a special operation, called a *semijoin,* which is used to optimize some types of queries in a DDBMS. A detailed discussion about optimization algorithms is beyond the scope of this book. We attempt to illustrate optimization principles using suitable examples.[8]

25.5.1 Distributed Query Processing

A distributed database query is processed in stages as follows:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.
3. **Global Query Optimization**. Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).

[8]For a detailed discussion of optimization algorithms, see Ozsu and Valduriez (1999).

4. **Local Query Optimization**. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, while the last stage is performed locally.

25.5.2 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 19. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become quite significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations in Figure 3.5 are distributed at two sites as shown in Figure 25.10. We will assume in this example that neither relation is fragmented. According to Figure 25.10, the size of the EMPLOYEE relation is $100 * 10{,}000 = 10^6$ bytes, and the size of the DEPARTMENT relation is $35 * 100 = 3500$ bytes. Consider the query Q: *For each employee, retrieve the employee name and the name of the department for which the employee works.* This can be stated as follows in the relational algebra:

$$Q: \pi_{Fname,Lname,Dname}(EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long.*

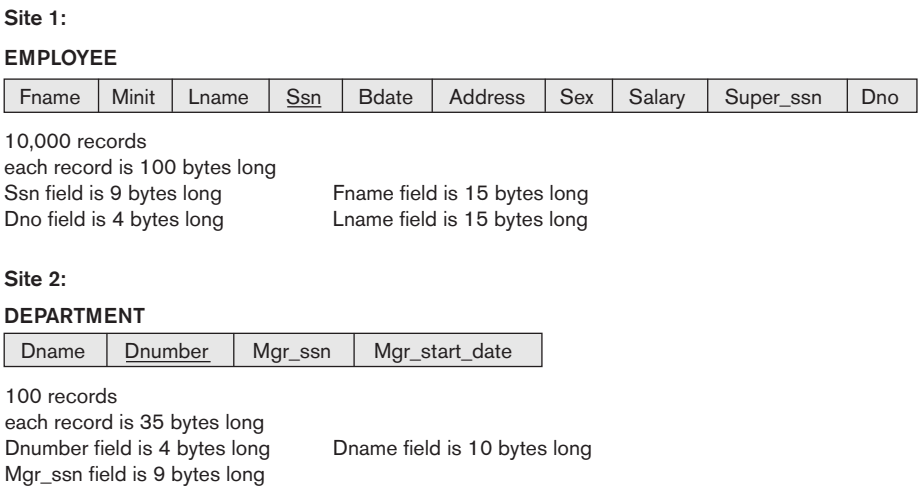


Figure 25.10
Example to illustrate volume of data transferred.

The query is submitted at a distinct site 3, which is called the **result site** because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query Q′: *For each department, retrieve the department name and the name of the department manager.* This can be stated as follows in the relational algebra:

$$Q': \pi_{Fname,Lname,Dname}(\text{DEPARTMENT} \bowtie_{Mgr\_ssn=Ssn} \text{EMPLOYEE})$$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query Q apply to Q′, except that the result of Q′ includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 100 = 4,000 bytes, so 4,000 + 1,000,000 = 1,004,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 4,000 + 3,500 = 7,500 bytes must be transferred.

Again, we would choose strategy 3—this time by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q′.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3,500 = 403,500 bytes must be transferred for Q and 4,000 + 3,500 = 7,500 bytes for Q′.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin**. We introduce this operation and discuss distributed execution using semijoins next.

25.5.3 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation R to the site where the other relation S is located; this column is then joined with S. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R. Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be quite an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q′:

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q, we transfer $F = \pi_{\text{Dnumber}}(\text{DEPARTMENT})$, whose size is $4 * 100 = 400$ bytes, whereas, for Q′, we transfer $F' = \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$, whose size is $9 * 100 = 900$ bytes.
2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q, we transfer $R = \pi_{\text{Dno, Fname, Lname}}(F \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$, whose size is $34 * 10{,}000 = 340{,}000$ bytes, whereas, for Q′, we transfer $R' = \pi_{\text{Mgr\_ssn, Fname, Lname}}(F' \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$, whose size is $39 * 100 = 3{,}900$ bytes.
3. Execute the query by joining the transferred file R or R' with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4,800 bytes for Q′. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q′ only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin operation** $R \ltimes_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi_R(R \bowtie_{A=B} S)$. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_B(S)$ to the site where R resides and then joining F with R, thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

$$R \ltimes S \neq S \ltimes R$$

25.5.4 Query and Update Decomposition

In a DDBMS with *no distribution transparency,* the user phrases a query directly in terms of specific fragments. For example, consider another query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*, which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 25.8, and those at site 1 are shown in Figure 3.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS_5 and WORKS_ON_5 relations at site 2 (Figure 25.8) or the PROJECT and WORKS_ON relations at site 1 (Figure 3.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency.*

On the other hand, a DDBMS that supports *full distribution, fragmentation,* and *replication transparency* allows the user to specify a query or update request on the schema in Figure 3.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms to be discussed in Section 25.7. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 3.6) are TRUE (all tuples), and the attribute lists are * (all attributes). For the fragments shown in Figure 25.8, we have the guard conditions and attribute lists shown in Figure 25.11. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple <'Alex', 'B', 'Coleman', '345671239', '22-APR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> would be decomposed by the DDBMS into two insert requests: the first inserts the preceding tuple in the EMPLOYEE fragment at site 1, and the second inserts the projected tuple <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4> in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard

(a) EMPD5
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno=5
DEP5
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber=5
DEP5_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber=5
PROJS5
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum=5
WORKS_ON5
attribute list: * (all attributes Essn, Pno,Hours)
guard condition: Essn IN (π_{Ssn} (EMPD5)) OR Pno IN ($\pi_{Pnumber}$ (PROJS5))

(b) EMPD4
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno=4
DEP4
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber=4
DEP4_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber=4
PROJS4
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum=4
WORKS_ON4
attribute list: * (all attributes Essn, Pno, Hours)
guard condition: Essn IN (π_{Ssn} (EMPD4))
OR Pno IN ($\pi_{Pnumber}$ (PROJS4))

Figure 25.11
Guard conditions and attributes lists for fragments.
(a) Site 2 fragments. (b) Site 3 fragments.

conditions. For example, consider the query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*. This can be specified in SQL on the schema in Figure 3.5 as follows:

```
Q:  SELECT Fname, Lname, Hours
    FROM   EMPLOYEE, PROJECT, WORKS_ON
    WHERE  Dnum=5 AND Pnumber=Pno AND Essn=Ssn;
```

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (Dnum = 5 AND Pnumber = Pno) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T_1 \leftarrow \pi_{\text{Essn}}(\text{PROJS5} \bowtie_{\text{Pnumber=Pno}} \text{WORKS\_ON5})$$
$$T_2 \leftarrow \pi_{\text{Essn, Fname, Lname}}(T_1 \bowtie_{\text{Essn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Hours}}(T_2 * \text{WORKS\_ON5})$$

This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying (Dnum = 5) and that WORKS_ON5 contains all tuples to be joined with PROJS5; hence, subquery T_1 can be executed at site 2, and the projected column Essn can be sent to site 1. Subquery T_2 can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

25.6 Overview of Transaction Management in Distributed Databases

The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions (see Chapter 21). We discuss distributed transaction management in this section and explore concurrency control in Section 25.7.

As can be seen in Figure 25.5, an additional component called the **global transaction manager** is introduced for supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs. The operations exported by this interface are similar to those covered in Section 21.2.1, namely BEGIN_TRANSACTION, READ or WRITE, END_TRANSACTION, COMMIT_TRANSACTION, and ROLLBACK (or ABORT). The manager stores bookkeeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For READ operations, it returns a local copy if valid and available. For WRITE operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For ABORT operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For COMMIT operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic termination (COMMIT/ ABORT) of distributed transactions is commonly implemented using the two-phase commit protocol. We give more details of this protocol in the following section.

The transaction manager passes to the concurrency controller the database operation and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is delayed until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation. We discuss commonly used distributed concurrency methods in Section 25.7.

25.6.1 Two-Phase Commit Protocol

In Section 23.6, we described the *two-phase commit protocol* (**2PC**), which requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables) The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol, which we discuss next.

25.6.2 Three-Phase Commit Protocol

The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources. Another problematic scenario is when both the coordinator and a participant that has committed crash together. In the two-phase commit protocol, a participant has no way to ensure that all participants got the commit message in the second phase. Hence once a decision to commit has been made by the coordinator in the first phase, participants will commit their transactions in the second phase independent of receipt of a global commit message by other participants. Thus, in the situation that both the coordinator and a committed participant crash together, the result of the transaction becomes uncertain or nondeterministic. Since the transaction has already been committed by one participant, it cannot be aborted on recovery by the coordinator. Also, the transaction cannot be optimistically committed on recovery since the original vote of the coordinator may have been to abort.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have committed and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

25.6.3 Operating System Support for Transaction Management

The following are the main benefits of operating system (OS)-supported transaction management:

- Typically, DBMSs use their own semaphores[9] to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in userspace at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this lock resource get queued. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.
- Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.
- Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

25.7 Overview of Concurrency Control and Recovery in Distributed Databases

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These include the following:

- **Dealing with *multiple copies* of the data items.** The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.

[9]Semaphores are data structures used for synchronized and exclusive access to shared resources for preventing race conditions in a parallel computing system.

- **Failure of individual sites.** The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up-to-date with the rest of the sites before it rejoins the system.
- **Failure of communication links.** The system must be able to deal with the failure of one or more of the communication links that connect the sites. An extreme case of this problem is that **network partitioning** may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.
- **Distributed commit.** Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The **two-phase commit protocol** (see Section 23.6) is often used to deal with this problem.
- **Distributed deadlock.** Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed concurrency control and recovery techniques must deal with these and other problems. In the following subsections, we review some of the techniques that have been suggested to deal with recovery and concurrency control in DDBMSs.

25.7.1 Distributed Concurrency Control Based on a Distinguished Copy of a Data Item

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques for centralized databases. We discuss these techniques in the context of extending centralized *locking.* Similar extensions apply to other concurrency control techniques. The idea is to designate *a particular copy* of each data item as a **distinguished copy**. The locks for this data item are associated *with the distinguished copy,* and all locking and unlocking requests are sent to the site that contains that copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site technique**, all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site**. Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item. We discuss these techniques next.

Primary Site Technique. In this method a single **primary site** is designated to be the **coordinator site** *for all database items.* Hence, all locks are kept at that site, and all requests for locking or unlocking are sent there. This method is thus an extension

of the centralized locking approach. For example, if all transactions follow the two-phase locking protocol, serializability is guaranteed. The advantage of this approach is that it is a simple extension of the centralized approach and thus is not overly complex. However, it has certain inherent disadvantages. One is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. A second disadvantage is that failure of the primary site paralyzes the system, since all locking information is kept at that site. This can limit system reliability and availability.

Although all locks are accessed at the primary site, the items themselves can be accessed at any site at which they reside. For example, once a transaction obtains a Read_lock on a data item from the primary site, it can access any copy of that data item. However, once a transaction obtains a Write_lock and updates a data item, the DDBMS is responsible for updating *all copies* of the data item before releasing the lock.

Primary Site with Backup Site. This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site**. All locking information is maintained at both the primary and the backup sites. In case of primary site failure, the backup site takes over as the primary site, and a new backup site is chosen. This simplifies the process of recovery from failure of the primary site, since the backup site takes over and processing can resume after a new backup site is chosen and the lock status information is copied to that site. It slows down the process of acquiring locks, however, because all lock requests and granting of locks must be recorded at *both the primary and the backup sites* before a response is sent to the requesting transaction. The problem of the primary and backup sites becoming overloaded with requests and slowing down the system remains undiminished.

Primary Copy Technique. This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items *stored at different sites.* Failure of one site affects any transactions that are accessing locks on items whose primary copies reside at that site, but other transactions are not affected. This method can also use backup sites to enhance reliability and availability.

Choosing a New Coordinator Site in Case of Failure. Whenever a coordinator site fails in any of the preceding techniques, the sites that are still running must choose a new coordinator. In the case of the primary site approach with *no* backup site, all executing transactions must be aborted and restarted in a tedious recovery process. Part of the recovery process involves choosing a new primary site and creating a lock manager process and a record of all lock information at that site. For methods that use backup sites, transaction processing is suspended while the backup site is designated as the new primary site and a new backup site is chosen and is sent copies of all the locking information from the new primary site.

If a backup site X is about to become the new primary site, X can choose the new backup site from among the system's running sites. However, if no backup site

existed, or if both the primary and the backup sites are down, a process called **election** can be used to choose the new coordinator site. In this process, any site *Y* that attempts to communicate with the coordinator site repeatedly and fails to do so can assume that the coordinator is down and can start the election process by sending a message to all running sites proposing that *Y* become the new coordinator. As soon as *Y* receives a majority of yes votes, *Y* can declare that it is the new coordinator. The election algorithm itself is quite complex, but this is the main idea behind the election method. The algorithm also resolves any attempt by two or more sites to become coordinator at the same time. The references in the Selected Bibliography at the end of this chapter discuss the process in detail.

25.7.2 Distributed Concurrency Control Based on Voting

The concurrency control methods for replicated items discussed earlier all use the idea of a distinguished copy that maintains the locks for that item. In the **voting method**, there is no distinguished copy; rather, a lock request is sent to all sites that includes a copy of the data item. Each copy maintains its own lock and can grant or deny the request for it. If a transaction that requests a lock is granted that lock by *a majority* of the copies, it holds the lock and informs *all copies* that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain *time-out period,* it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved. Simulation studies have shown that voting has higher message traffic among sites than do the distinguished copy methods. If the algorithm takes into account possible site failures during the voting process, it becomes extremely complex.

25.7.3 Distributed Recovery

The recovery process in distributed databases is quite involved. We give only a very brief idea of some of the issues here. In some cases it is quite difficult even to determine whether a site is down without exchanging numerous messages with other sites. For example, suppose that site *X* sends a message to site *Y* and expects a response from *Y* but does not receive it. There are several possible explanations:

- The message was not delivered to *Y* because of communication failure.
- Site *Y* is down and could not respond.
- Site *Y* is running and sent a response, but the response was not delivered.

Without additional information or the sending of additional messages, it is difficult to determine what actually happened.

Another problem with distributed recovery is distributed commit. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on *every* site cannot be lost. This means that every site must first

have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol is often used to ensure the correctness of distributed commit (see Section 23.6).

25.8 Distributed Catalog Management

Efficient catalog management in distributed databases is critical to ensure satisfactory performance related to site autonomy, view management, and data distribution and replication. Catalogs are databases themselves containing metadata about the distributed database system.

Three popular management schemes for distributed catalogs are *centralized* catalogs, *fully replicated* catalogs, and *partitioned* catalogs. The choice of the scheme depends on the database itself as well as the access patterns of the applications to the underlying data.

Centralized Catalogs. In this scheme, the entire catalog is stored in one single site. Owing to its central nature, it is easy to implement. On the other hand, the advantages of reliability, availability, autonomy, and distribution of processing load are adversely impacted. For read operations from noncentral sites, the requested catalog data is locked at the central site and is then sent to the requesting site. On completion of the read operation, an acknowledgement is sent to the central site, which in turn unlocks this data. All update operations must be processed through the central site. This can quickly become a performance bottleneck for write-intensive applications.

Fully Replicated Catalogs. In this scheme, identical copies of the complete catalog are present at each site. This scheme facilitates faster reads by allowing them to be answered locally. However, all updates must be broadcast to all sites. Updates are treated as transactions and a centralized two-phase commit scheme is employed to ensure catalog consitency. As with the centralized scheme, write-intensive applications may cause increased network traffic due to the broadcast associated with the writes.

Partially Replicated Catalogs. The centralized and fully replicated schemes restrict site autonomy since they must ensure a consistent global view of the catalog. Under the partially replicated scheme, each site maintains complete catalog information on data stored locally at that site. Each site is also permitted to cache entries retrieved from remote sites. However, there are no guarantees that these cached copies will be the most recent and updated. The system tracks catalog entries for sites where the object was created and for sites that contain copies of this object. Any changes to copies are propagated immediately to the original (birth) site. Retrieving updated copies to replace stale data may be delayed until an access to this data occurs. In general, fragments of relations across sites should be uniquely accessible. Also, to ensure data distribution transparency, users should be allowed to create synonyms for remote objects and use these synonyms for subsequent referrals.

25.9 Current Trends in Distributed Databases

Current trends in distributed data management are centered on the Internet, in which petabytes of data can be managed in a scalable, dynamic, and reliable fashion. Two important areas in this direction are cloud computing and peer-to-peer databases.

25.9.1 Cloud Computing

Cloud computing is the paradigm of offering computer infrastructure, platforms, and software as services over the Internet. It offers significant economic advantages by limiting both up-front capital investments toward computer infrastructure as well as total cost of ownership. It has introduced a new challenge of managing petabytes of data in a scalable fashion. Traditional database systems for managing enterprise data proved to be inadequate in handling this challenge, which has resulted in a major architectural revision. The Claremont report[10] by a group of senior database researchers envisions that future research in cloud computing will result in the emergence of new data management architectures and the interplay of structured and unstructured data as well as other developments.

Performance costs associated with partial failures and global synchronization were key performance bottlenecks of traditional database solutions. The key insight is that the hash-value nature of the underlying datasets used by these organizations lends itself naturally to partitioning. For instance, search queries essentially involve a recursive process of mapping keywords to a set of related documents, which can benefit from such a partitioning. Also, the partitions can be treated independently, thereby eliminating the need for a coordinated commit. Another problem with traditional DDBMSs is the lack of support for efficient dynamic partitioning of data, which limited scalability and resource utilization. Traditional systems treated system metadata and application data alike, with the system data requiring strict consistency and availability guarantees. But application data has variable requirements on these characteristics, depending on its nature. For example, while a search engine can afford weaker consistency guarantees, an online text editor like Google Docs, which allows concurrent users, has strict consistency requirements.

The metadata of a distributed database system should be decoupled from its actual data in order to ensure scalability. This decoupling can be used to develop innovative solutions to manage the actual data by exploiting their inherent suitability to partitioning and using traditional database solutions to manage critical system metadata. Since metadata is only a fraction of the total data set, it does not prove to be a performance bottleneck. Single object semantics of these implementations enables higher tolerance to nonavailability of certain sections of data. Access to data is typically by a single object in an atomic fashion. Hence, transaction support to such data is not as stringent as for traditional databases.[11] There is a varied set of

[10]"The Claremont Report on Database Research" is available at http://db.cs.berkeley.edu/claremont/claremontreport08.pdf.

[11]Readers may refer to the work done by Das et al. (2008) for further details.

cloud services available today, including application services (salesforce.com), storage services (Amazon Simple Storage Service, or Amazon S3), compute services (Google App Engine, Amazon Elastic Compute Cloud—Amazon EC2), and data services (Amazon SimpleDB, Microsoft SQL Server Data Services, Google's Datastore). More and more data-centric applications are expected to leverage data services in the cloud. While most current cloud services are data-analysis intensive, it is expected that business logic will eventually be migrated to the cloud. The key challenge in this migration would be to ensure the scalability advantages for multiple object semantics inherent to business logic. For a detailed treatment of cloud computing, refer to the relevant bibliographic references in this chapter's Selected Bibliography.

25.9.2 Peer-to-Peer Database Systems

A peer-to-peer database system (PDBS) aims to integrate advantages of P2P (peer-to-peer) computing, such as scalability, attack resilience, and self-organization, with the features of decentralized data management. Nodes are autonomous and are linked only to a small number of peers individually. It is permissible for a node to behave purely as a collection of files without offering a complete set of traditional DBMS functionality. While FDBS and MDBS mandate the existence of mappings between local and global federated schemas, PDBSs attempt to avoid a global schema by providing mappings between pairs of information sources. In PDBS, each peer potentially models semantically related data in a manner different from other peers, and hence the task of constructing a central mediated schema can be very challenging. PDBSs aim to decentralize data sharing. Each peer has a schema associated with its domain-specific stored data. The PDBS constructs a semantic path[12] of mappings between peer schemas. Using this path, a peer to which a query has been submitted can obtain information from any relevant peer connected through this path. In multidatabase systems, a separate global query processor is used, whereas in a P2P system a query is shipped from one peer to another until it is processed completely. A query submitted to a node may be forwarded to others based on the mapping graph of semantic paths. Edutella and Piazza are examples of PDBSs. Details of these systems can be found from the sources mentioned in this chapter's Selected Bibliography.

25.10 Distributed Databases in Oracle[13]

Oracle provides support for homogeneous, heterogeneous, and client server architectures of distributed databases. In a homogeneous architecture, a minimum of two Oracle databases reside on at least one machine. Although the location and platform of the databases are transparent to client applications, they would need to

[12]A **semantic path** describes the higher-level relationship between two domains that are dissimilar but not unrelated.

[13]The discussion is based on available documentation at http://docs.oracle.com.

distinguish between local and remote objects semantically. Using synonyms, this need can be overcome wherein users can access the remote objects with the same syntax as local objects. Different versions of DBMSs can be used, although it must be noted that Oracle offers backward compatibility but not forward compatibility between its versions. For example, it is possible that some of the SQL extensions that were incorporated into Oracle 11i may not be understood by Oracle 9.

In a heterogeneous architecture, at least one of the databases in the network is a non-Oracle system. The Oracle database local to the application hides the underlying heterogeneity and offers the view of a single local, underlying Oracle database. Connectivity is handled by use of an ODBC- or OLE-DB-compliant protocol or by Oracle's Heterogeneous Services and Transparent Gateway agent components. A discussion of the Heterogeneous Services and Transparent Gateway agents is beyond the scope of this book, and the reader is advised to consult the online Oracle documentation.

In the client-server architecture, the Oracle database system is divided into two parts: a front end as the client portion, and a back end as the server portion. The client portion is the front-end database application that interacts with the user. The client has no data access responsibility and merely handles the requesting, processing, and presentation of data managed by the server. The server portion runs Oracle and handles the functions related to concurrent shared access. It accepts SQL and PL/SQL statements originating from client applications, processes them, and sends the results back to the client. Oracle client-server applications provide location transparency by making the location of data transparent to users; several features like views, synonyms, and procedures contribute to this. Global naming is achieved by using <TABLE_NAME@DATABASE_NAME> to refer to tables uniquely.

Oracle uses a two-phase commit protocol to deal with concurrent distributed transactions. The COMMIT statement triggers the two-phase commit mechanism. The RECO (recoverer) background process automatically resolves the outcome of those distributed transactions in which the commit was interrupted. The RECO of each local Oracle server automatically commits or rolls back any *in-doubt* distributed transactions consistently on all involved nodes. For long-term failures, Oracle allows each local DBA to manually commit or roll back any in-doubt transactions and free up resources. Global consistency can be maintained by restoring the database at each site to a predetermined fixed point in the past.

Oracle's distributed database architecture is shown in Figure 25.12. A node in a distributed database system can act as a client, as a server, or both, depending on the situation. The figure shows two sites where databases called HQ (headquarters) and Sales are kept. For example, in the application shown running at the headquarters, for an SQL statement issued against local data (for example, DELETE FROM DEPT ...), the HQ computer acts as a server, whereas for a statement against remote data (for example, INSERT INTO EMP@SALES), the HQ computer acts as a client.

Communication in such a distributed heterogeneous environment is facilitated through Oracle Net Services, which supports standard network protocols and APIs. Under Oracle's client-server implementation of distributed databases, Net Services

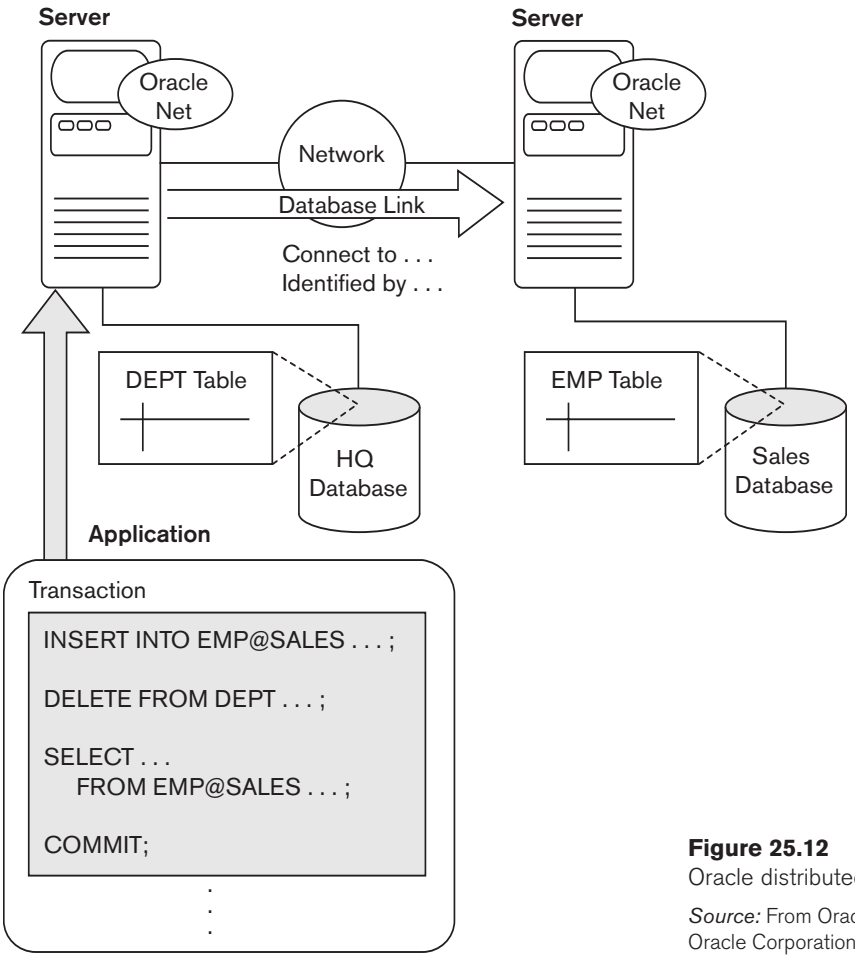


Figure 25.12
Oracle distributed database system.

Source: From Oracle (2008). Copyright © Oracle Corporation 2008. All rights reserved.

is responsible for establishing and managing connections between a client application and database server. It is present in each node on the network running an Oracle client application, database server, or both. It packages SQL statements into one of the many communication protocols to facilitate client-to-server communication and then packages the results back similarly to the client. The support offered by Net Services to heterogeneity refers to platform specifications only and not the database software. Support for DBMSs other than Oracle is through Oracle's Heterogeneous Services and Transparent Gateway. Each database has a unique global name provided by a hierarchical arrangement of network domain names that is prefixed to the database name to make it unique.

Oracle supports database links that define a one-way communication path from one Oracle database to another. For example,

CREATE DATABASE LINK sales.us.americas;

establishes a connection to the sales database in Figure 25.12 under the network domain us that comes under domain americas. Using links, a user can access a remote object on another database subject to ownership rights without the need for being a user on the remote database.

Data in an Oracle DDBS can be replicated using snapshots or replicated master tables. Replication is provided at the following levels:

- **Basic replication.** Replicas of tables are managed for read-only access. For updates, data must be accessed at a single primary site.
- **Advanced (symmetric) replication.** This extends beyond basic replication by allowing applications to update table replicas throughout a replicated DDBS. Data can be read and updated at any site. This requires additional software called *Oracle's advanced replication option*. A **snapshot** generates a copy of a part of the table by means of a query called the *snapshot defining query*. A simple snapshot definition looks like this:

```
CREATE SNAPSHOT SALES_ORDERS AS
SELECT * FROM SALES_ORDERS@hq.us.americas;
```

Oracle groups snapshots into refresh groups. By specifying a refresh interval, the snapshot is automatically refreshed periodically at that interval by up to ten **Snapshot Refresh Processes** (**SNPs**). If the defining query of a snapshot contains a distinct or aggregate function, a GROUP BY or CONNECT BY clause, or join or set operations, the snapshot is termed a **complex snapshot** and requires additional processing. Oracle (up to version 7.3) also supports ROWID snapshots that are based on physical row identifiers of rows in the master table.

Heterogeneous Databases in Oracle. In a heterogeneous DDBS, at least one database is a non-Oracle system. **Oracle Open Gateways** provides access to a non-Oracle database from an Oracle server, which uses a database link to access data or to execute remote procedures in the non-Oracle system. The Open Gateways feature includes the following:

- **Distributed transactions.** Under the two-phase commit mechanism, transactions may span Oracle and non-Oracle systems.
- **Transparent SQL access.** SQL statements issued by an application are transparently transformed into SQL statements understood by the non-Oracle system.
- **Pass-through SQL and stored procedures.** An application can directly access a non-Oracle system using that system's version of SQL. Stored procedures in a non-Oracle SQL-based system are treated as if they were PL/SQL remote procedures.
- **Global query optimization.** Cardinality information, indexes, and so on at the non-Oracle system are accounted for by the Oracle server query optimizer to perform global query optimization.
- **Procedural access.** Procedural systems like messaging or queuing systems are accessed by the Oracle server using PL/SQL remote procedure calls.

In addition to the above, data dictionary references are translated to make the non-Oracle data dictionary appear as a part of the Oracle server's dictionary. Character set translations are done between national language character sets to connect multilingual databases.

From a security perspective, Oracle recommends that if a query originates at site A and accesses sites B, C, and D, then the auditing of links should be done in the database at site A only. This is because the remote databases cannot distinguish whether a successful connection request and following SQL statements are coming from another server or a locally connected client.

25.10.1 Directory Services

A concept closely related with distributed enterprise systems is **online directories**. Online directories are essentially a structured organization of metadata needed for management functions. They can represent information about a variety of sources ranging from security credentials, shared network resources, and database catalog. **Lightweight Directory Access Protocol** (LDAP) is an industry standard protocol for directory services. LDAP enables the use of a partitioned **Directory Information Tree** (DIT) across multiple LDAP servers, which in turn can return references to other servers as a result of a directory query. Online directories and LDAP are particularly important in distributed databases, wherein access of metadata related to transparencies discussed in Section 25.1 must be scalable, secure, and highly available.

Oracle supports LDAP Version 3 and online directories through Oracle Internet Directory, a general-purpose directory service for fast access and centralized management of metadata pertaining to distributed network resources and users. It runs as an application on an Oracle database and communicates with the database through Oracle Net Services. It also provides password-based, anonymous, and certificate-based user authentication using SSL Version 3.

Figure 25.13 illustrates the architecture of the Oracle Internet Directory. The main components are:

- **Oracle directory server**. Handles client requests and updates for information pertaining to people and resources.
- **Oracle directory replication server.** Stores a copy of the LDAP data from Oracle directory servers as a backup.
- **Directory administrator:** Supports both GUI-based and command line-based interfaces for directory administration.

25.11 Summary

In this chapter we provided an introduction to distributed databases. This is a very broad topic, and we discussed only some of the basic techniques used with distributed databases. First we discussed the reasons for distribution and the potential advantages of distributed databases over centralized systems. Then the concept of

CHAPTER

1

PL/SQL Overview

This chapter describes the basic definition and concepts PL/SQL:

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

Features of PL/SQL

PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.

- It supports developing web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. Dynamic SQL is SQL allows embedding DDL statements in PL/SQL blocks.

- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.

- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.

- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

- Applications written in PL/SQL are fully portable.

- PL/SQL provides high security level.

- PL/SQL provides access to predefined SQL packages.

- PL/SQL provides support for Object-Oriented Programming.

- PL/SQL provides support for Developing Web Applications and Server Pages.

CHAPTER

3

Basic Syntax

This chapter describes the basic syntax followed:

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

S.N.	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

The 'Hello World' Example:

```
DECLARE
   message  varchar2(20):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
```

```
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type **/** at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
Hello World

PL/SQL procedure successfully completed.
```

The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL:

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
******	Exponentiation operator
<<, >>	Label delimiter (begin and end)

/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

Program comments are explanatory statements that you can include in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow for some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by PL/SQL compiler. The PL/SQL single-line comments start with the delimiter **--**(double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE
   -- variable declaration
   message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
    *  PL/SQL executable statement(s)
    */
   dbms_output.put_line(message);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Hello World


PL/SQL procedure successfully completed.
```

PL/SQL Program Units

A PL/SQL unit is any one of the following:

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger

- Type
- Type body

Each of these units will be discussed in the forthcoming chapters.

CHAPTER

4

Data Types

This chapter describes the Data Types used under PL/SQL:

PL/SQL variables, constants and parameters must have a valid data type which specifies a storage format, constraints, and valid range of values. This tutorial will take you through **SCALAR** and **LOB** data types available in PL/SQL and other two data types will be covered in other chapters.

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
Reference	Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories:

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
Datetime	Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following is the detail of PL/SQL pre-defined numeric data types and their sub-types:

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, secale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration:

```
DECLARE
   num1 INTEGER;
   num2 REAL;
   num3 DOUBLE PRECISION;
BEGIN
   null;
END;
/
```

When the above code is compiled and executed, it produces the following result:

```
PL/SQL procedure successfully completed
```

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types:

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in:

- SQL statements
- Built-in SQL functions (such as TO_CHAR)
- PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The **DATE** datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year, for example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field:

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types:

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype:

```
DECLARE
   SUBTYPE name IS char(20);
   SUBTYPE message IS varchar2(100);
   salutation name;
   greetings message;
BEGIN
   salutation := 'Reader ';
   greetings := 'Welcome to the World of PL/SQL';
   dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Hello Reader Welcome to the World of PL/SQL


PL/SQL procedure successfully completed.
```

NULLs in PL/SQL

PL/SQL NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type. Note that NULL is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

CHAPTER

5

Variables

This chapter describes the variables used:

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows to define various types of variables,s which we will cover in subsequent chapters like date time data types, records, collections, etc. For this chapter, let us study only basic variable types.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in last chapter. Some valid variable declarations along with their definition are shown below:

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example:

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following:

- The **DEFAULT** keyword
- The **assignment** operator

For example:

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes program would produce unexpected result. Try the following example which makes use of various types of variables:

```
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result:

```
Value of c: 30

Value of f: 23.333333333333333


PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of Blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a

variable is declared and accessible to an outer Block, it is also accessible to all nested inner Blocks. There are two types of variable scope:

- **Local variables** - variables declared in an inner block and not accessible to outer blocks.
- **Global variables** - variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form:

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
/
```

When the above code is executed, it produces the following result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the SELECT INTO statement of SQL to assign values to PL/SQL variables. For each item in the SELECT list, there must be a corresponding, type-compatible variable in the INTO list. The following example illustrates the concept: Let us create a table named CUSTOMERS:

(**For SQL statements please look at the SQL tutorial**)

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);

Table Created
```

Next, let us insert some values in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the SELECT INTO clause of SQL:

```
DECLARE
   c_id customers.id%type := 1;
   c_name  customers.name%type;
   c_addr customers.address%type;
   c_sal  customers.salary%type;
BEGIN
   SELECT name, address, salary INTO c_name, c_addr, c_sal
   FROM customers
   WHERE id = c_id;

   dbms_output.put_line
   ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

When the above code is executed, it produces the following result:

```
Customer Ramesh from Ahmedabad earns 2000


PL/SQL procedure completed successfully
```

CHAPTER

6

Constants

This chapter shows the usage of constants:

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Declaring a Constant

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed. For example:

```
PI CONSTANT NUMBER := 3.141592654;
DECLARE
   -- constant declaration
   pi constant number := 3.141592654;
   -- other declarations
   radius number(5,2);
   dia number(5,2);
   circumference number(7, 2);
   area number (10, 2);
BEGIN
   -- processing
   radius := 9.5;
   dia := radius * 2;
   circumference := 2.0 * pi * radius;
   area := pi * radius * radius;
   -- output
   dbms_output.put_line('Radius: ' || radius);
   dbms_output.put_line('Diameter: ' || dia);
   dbms_output.put_line('Circumference: ' || circumference);
   dbms_output.put_line('Area: ' || area);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Radius: 9.5
Diameter: 19
```

```
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals:

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

Literal Type	Example:
Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL.
Date and Time Literals	DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown below:

```
DECLARE
   message  varchar2(20):= ''That''s tutorialspoint.com!'';
BEGIN
   dbms_output.put_line(message);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
That's tutorialspoint.com!

PL/SQL procedure successfully completed.
```

CHAPTER

7

Operators

This chapter describes the different operators used under PL/SQL:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators:

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

This tutorial will explain the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed under the chapter: **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5, then:

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Example:

```
BEGIN
   dbms_output.put_line( 10 + 5);
   dbms_output.put_line( 10 - 5);
   dbms_output.put_line( 10 * 5);
   dbms_output.put_line( 10 / 5);
   dbms_output.put_line( 10 ** 5);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
15
5
50
2
100000


PL/SQL procedure successfully completed.
```

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example:

```
DECLARE
   a number (2) := 21;
   b number (2) := 10;
BEGIN
   IF (a = b) then
      dbms_output.put_line('Line 1 - a is equal to b');
   ELSE
      dbms_output.put_line('Line 1 - a is not equal to b');
   END IF;
```

```
   IF (a < b) then
      dbms_output.put_line('Line 2 - a is less than b');
   ELSE
      dbms_output.put_line('Line 2 - a is not less than b');
   END IF;

   IF ( a > b ) THEN
      dbms_output.put_line('Line 3 - a is greater than b');
   ELSE
      dbms_output.put_line('Line 3 - a is not greater than b');
   END IF;

   -- Lets change value of a and b
   a := 5;
   b := 20;
   IF ( a <= b ) THEN
      dbms_output.put_line('Line 4 - a is either equal or less than b');
   END IF;

   IF ( b >= a ) THEN
      dbms_output.put_line('Line 5 - b is either equal or greater than a');
   END IF;

   IF ( a <> b ) THEN
      dbms_output.put_line('Line 6 - a is not equal to b');
   ELSE
      dbms_output.put_line('Line 6 - a is equal to b');
   END IF;

END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either equal or less than b
Line 5 - b is either equal or greater than a
Line 6 - a is not equal to b

PL/SQL procedure successfully completed
```

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a	If x = 10 then, x between 5 and 20

	and x <= b.	returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

LIKE Operator:

This program tests the LIKE operator, though you will learn how to write procedure in PL/SQL, but I'm going to use a small *procedure()* to show the functionality of LIKE operator:

```
DECLARE
PROCEDURE compare (value  varchar2,  pattern varchar2 ) is
BEGIN
   IF value LIKE pattern THEN
      dbms_output.put_line ('True');
   ELSE
      dbms_output.put_line ('False');
   END IF;
END;

BEGIN
   compare('Zara Ali', 'Z%A_i');
   compare('Nuha Ali', 'Z%A_i');
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
True
False

PL/SQL procedure successfully completed.
```

BETWEEN Operator:

The following program shows the usage of the BETWEEN operator:

```
DECLARE
   x number(2) := 10;
BEGIN
   IF (x between 5 and 20) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False');
   END IF;

   IF (x BETWEEN 5 AND 10) THEN
      dbms_output.put_line('True');
```

```
   ELSE
      dbms_output.put_line('False');
   END IF;

   IF (x BETWEEN 11 AND 20) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False');
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
True
True
False

PL/SQL procedure successfully completed.
```

IN and IS NULL Operators:

The following program shows the usage of IN and IS NULL operators:

```
DECLARE
   letter varchar2(1) := 'm';
BEGIN
   IF (letter in ('a', 'b', 'c')) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False');
   END IF;

   IF (letter in ('m', 'n', 'o')) THEN
       dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False');
   END IF;

   IF (letter is null) THEN
    dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False');
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
False
True
False

PL/SQL procedure successfully completed.
```

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

Operator	Description	Example
and	Called logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

Example:

```
DECLARE
   a boolean := true;
   b boolean := false;
BEGIN
   IF (a AND b) THEN
      dbms_output.put_line('Line 1 - Condition is true');
   END IF;
   IF (a OR b) THEN
      dbms_output.put_line('Line 2 - Condition is true');
   END IF;
   IF (NOT a) THEN
      dbms_output.put_line('Line 3 - a is not true');
   ELSE
      dbms_output.put_line('Line 3 - a is true');
   END IF;
   IF (NOT b) THEN
      dbms_output.put_line('Line 4 - b is not true');
   ELSE
      dbms_output.put_line('Line 4 - b is true');
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Line 2 - Condition is true
Line 3 - a is true
Line 4 - b is not true


PL/SQL procedure successfully completed.
```

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator	Operation
**	Exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	logical negation
AND	Conjunction
OR	Inclusion

Example:

Try the following example to understand the operator precedence available in PL/SQL:

```
DECLARE
   a number(2) := 20;
   b number(2) := 10;
   c number(2) := 15;
   d number(2) := 5;
   e number(2) ;
BEGIN
   e := (a + b) * c / d;       -- ( 30 * 15 ) / 5
   dbms_output.put_line('Value of (a + b) * c / d is : '|| e );

   e := ((a + b) * c) / d;    -- (30 * 15 ) / 5
   dbms_output.put_line('Value of ((a + b) * c) / d is  : ' ||  e );

   e := (a + b) * (c / d);    -- (30) * (15/5)
   dbms_output.put_line('Value of (a + b) * (c / d) is  : '||  e );

   e := a + (b * c) / d;      --  20 + (150/5)
   dbms_output.put_line('Value of a + (b * c) / d is  : ' ||  e );
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50

PL/SQL procedure successfully completed.
```

CHAPTER

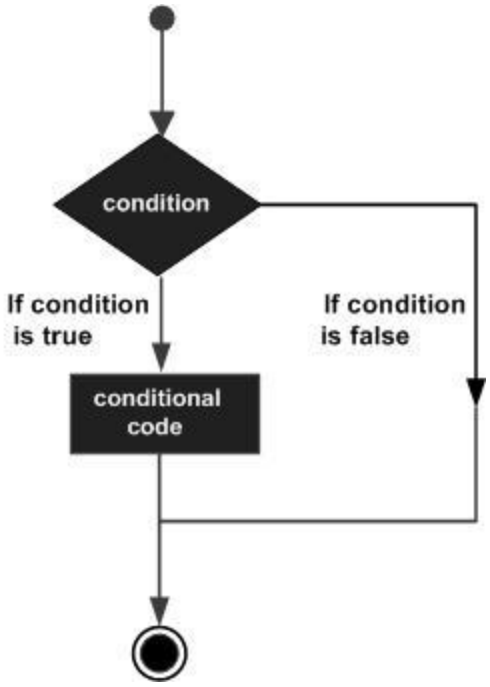
8

Conditions

This chapter describes the Decision Making Structure:

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical conditional (i.e., decision making) structure found in most of the programming languages:



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

Statement	Description
IF - THEN statement	The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
IF-THEN-ELSE statement	**IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL , then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
IF-THEN-ELSIF statement	It allows you to choose between several alternatives.
Case statement	Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
Searched CASE statement	The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
nested IF-THEN-ELSE	You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

IF - THEN statement

It is the simplest form of **IF** control statement, frequently used in decision making and changing the control flow of the program execution.
The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL,** then the **IF** statement does nothing.

Syntax:

Syntax for IF-THEN statement is:

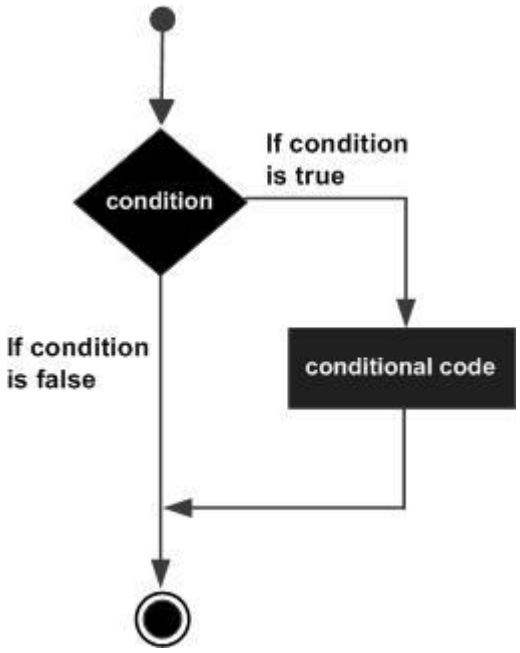
```
IF condition THEN
   S;
END IF;
```

Where *condition* is a Boolean or relational condition and *S* is a simple or compound statement. Example of an IF-THEN statement is:

```
IF (a <= 20) THEN
   c:= c+1;
END IF;
```

If the Boolean expression *condition* evaluates to true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

Flow Diagram:



Example 1:

Let us try a complete example that would illustrate the concept:

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
a is less than 20
value of a is : 10

PL/SQL procedure successfully completed.
```

Example 2:

Consider we have a table and few records in the table as we had created in PL/SQL Variable Types

```
DECLARE
   c_id customers.id%type := 1;
   c_sal  customers.salary%type;
BEGIN
```

```
   SELECT  salary
   INTO  c_sal
   FROM customers
   WHERE id = c_id;
   IF (c_sal <= 2000) THEN
      UPDATE customers
      SET salary =  salary + 1000
         WHERE id = c_id;
      dbms_output.put_line ('Salary updated');
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Salary updated


PL/SQL procedure successfully completed.
```

IF-THEN-ELSE statement

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

Syntax:

Syntax for the IF-THEN-ELSE statement is:

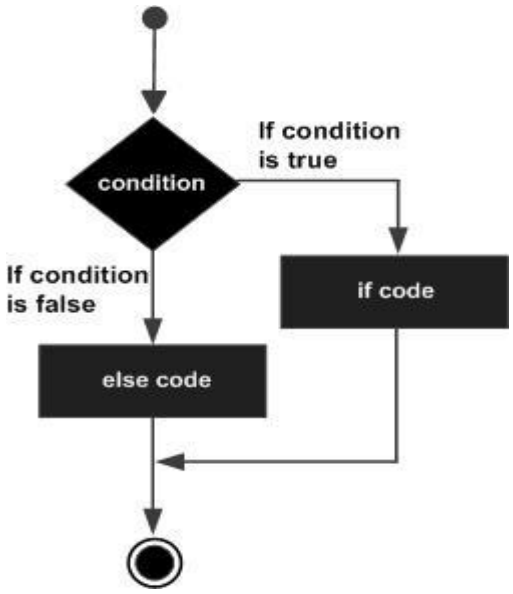
```
IF condition THEN
   S1;
ELSE
   S2;
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed. For example:

```
IF color = red THEN
  dbms_output.put_line('You have chosen a red car')
ELSE
  dbms_output.put_line('Please choose a color for your car');
END IF;
```

If the Boolean expression *condition* evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Flow Diagram:



Example:

Let us try a complete example that would illustrate the concept:

```
DECLARE
   a number(3) := 100;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   ELSE
      dbms_output.put_line('a is not less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
a is not less than 20
value of a is : 100


PL/SQL procedure successfully completed.
```

IF-THEN-ELSIF statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.
When using **IF-THEN-ELSIF** statements, there are few points to keep in mind.

- It's ELSIF, not ELSEIF

- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax:

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

```
IF(boolean_expression 1)THEN
   S1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
   S2;  -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
   S3; -- Executes when the boolean expression 3 is true
ELSE
   S4; -- executes when the none of the above condition is true
END IF;
```

Example:

```
DECLARE
   a number(3) := 100;
BEGIN
   IF ( a = 10 ) THEN
      dbms_output.put_line('Value of a is 10' );
   ELSIF ( a = 20 ) THEN
      dbms_output.put_line('Value of a is 20' );
   ELSIF ( a = 30 ) THEN
      dbms_output.put_line('Value of a is 30' );
   ELSE
       dbms_output.put_line('None of the values is matching');
   END IF;
   dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
None of the values is matching
Exact value of a is: 100

PL/SQL procedure successfully completed.
```

Case statement

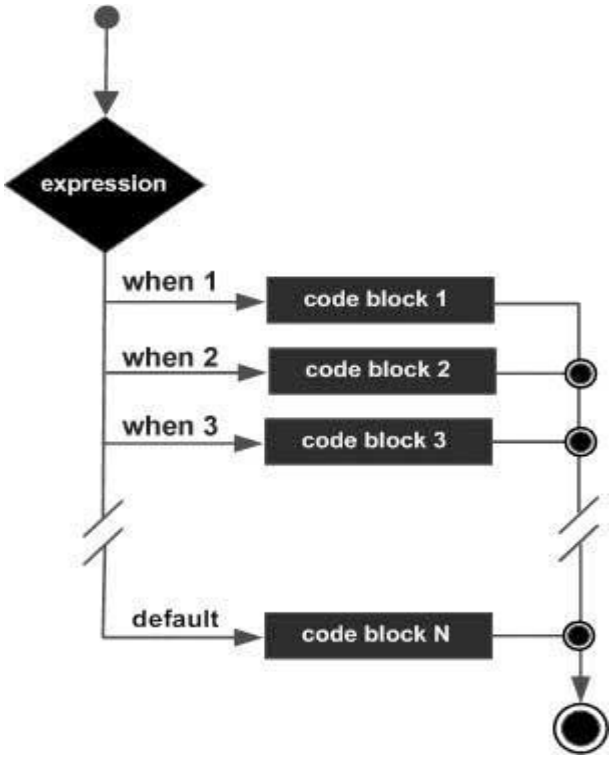
Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, whose value is used to select one of several alternatives.

Syntax:

The syntax for case statement in PL/SQL is:

```
CASE selector
    WHEN 'value1' THEN S1;
    WHEN 'value2' THEN S2;
    WHEN 'value3' THEN S3;
    ...
    ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Example:

```
DECLARE
   grade char(1) := 'A';
BEGIN
   CASE grade
      when 'A' then dbms_output.put_line('Excellent');
      when 'B' then dbms_output.put_line('Very good');
      when 'C' then dbms_output.put_line('Well done');
      when 'D' then dbms_output.put_line('You passed');
      when 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
   END CASE;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Excellent

PL/SQL procedure successfully completed.
```

Searched CASE statement

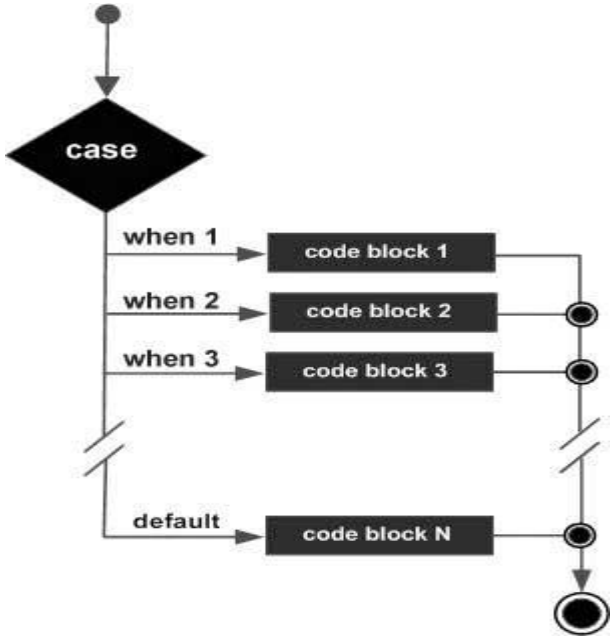
The searched **CASE** statement has no selector and its **WHEN** clauses contain search conditions that give Boolean values.

Syntax:

The syntax for searched case statement in PL/SQL is:

```
CASE
    WHEN selector = 'value1' THEN S1;
    WHEN selector = 'value2' THEN S2;
    WHEN selector = 'value3' THEN S3;
    ...
    ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Example:

```
DECLARE
   grade char(1) := 'B';
BEGIN
   case
      when grade = 'A' then dbms_output.put_line('Excellent');
      when grade = 'B' then dbms_output.put_line('Very good');
      when grade = 'C' then dbms_output.put_line('Well done');
      when grade = 'D' then dbms_output.put_line('You passed');
      when grade = 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
   end case;
```

```
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Very good

PL/SQL procedure successfully completed.
```

Nested IF-THEN-ELSE

It is always legal in PL/SQL programming to nest **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax:

```
IF( boolean_expression 1)THEN
    -- executes when the boolean expression 1 is true
    IF(boolean_expression 2) THEN
      -- executes when the boolean expression 2 is true
      sequence-of-statements;
   END IF;
ELSE
   -- executes when the boolean expression 1 is not true
  else-statements;
END IF;
```

Example:

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;
BEGIN
   -- check the boolean condition
   IF( a = 100 ) THEN
   -- if condition is true then check the following
      IF( b = 200 ) THEN
      -- if condition is true then print the following
         dbms_output.put_line('Value of a is 100 and b is 200' );
      END IF;
   END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

PL/SQL procedure successfully completed.
```

CHAPTER

9

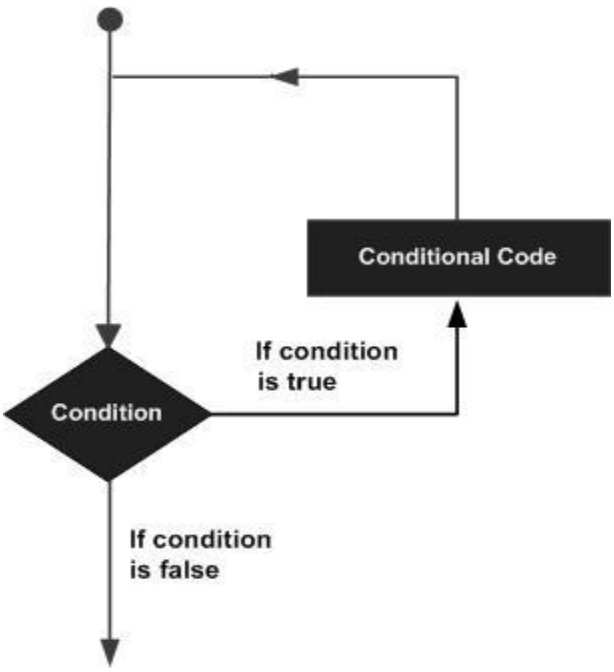
Loops

This chapter describes the various loops used under PL/SQL:

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

Loop Type	Description
PL/SQL Basic LOOP	In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
PL/SQL WHILE LOOP	Repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body.
PL/SQL FOR LOOP	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
Nested loops in PL/SQL	You can use one or more loop inside any another basic loop, while or for loop.

PL/SQL Basic LOOP

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax:

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP
  Sequence of statements;
END LOOP;
```

Here, sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

Example:

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
10
20
30
40
50
After Exit x is: 60
```

```
PL/SQL procedure successfully completed.
```

You can use the **EXIT WHEN** statement instead of the **EXIT** statement:

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
10

20

30

40

50

After Exit x is: 60


PL/SQL procedure successfully completed.
```

PL/SQL WHILE LOOP

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
```

```
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

PL/SQL FOR LOOP

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

Here is the flow of control in a for loop:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition ,i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the value of the *counter* variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop:

- The *initial_value* and *final_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The *initial_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows determine the loop range dynamically at run time.

Example:

```
DECLARE
   a number(2);
BEGIN
```

```
   FOR a in 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this:

```
DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

Nested loops in PL/SQL

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows:

```
LOOP
   Sequence of statements1
   LOOP
      Sequence of statements2
   END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
   sequence_of_statements1
   FOR counter2 IN initial_value2 .. final_value2 LOOP
      sequence_of_statements2
   END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
   sequence_of_statements1
   WHILE condition2 LOOP
      sequence_of_statements2
   END LOOP;
END LOOP;
```

Example:

The following program uses a nested basic loop to find the prime numbers from 2 to 100:

```
DECLARE
   i number(3);
   j number(3);
BEGIN
   i := 2;
   LOOP
      j:= 2;
      LOOP
         exit WHEN ((mod(i, j) = 0) or (j = i));
         j := j +1;
      END LOOP;
   IF (j = i ) THEN
      dbms_output.put_line(i || ' is prime');
   END IF;
   i := i + 1;
   exit WHEN i = 50;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

PL/SQL procedure successfully completed.
```

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept:

```
DECLARE
   i number(1);
   j number(1);
BEGIN
   << outer_loop >>
   FOR i IN 1..3 LOOP
      << inner_loop >>
      FOR j IN 1..3 LOOP
         dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
      END loop inner_loop;
   END loop outer_loop;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3

PL/SQL procedure successfully completed.
```

The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also helps in taking the control outside a loop. Click the following links to check their detail.

Control Statement	Description
EXIT statement	The Exit statement completes the loop and control passes to the statement immediately after END LOOP
CONTINUE statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
GOTO statement	Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program.

EXIT statement

The **EXIT** statement in PL/SQL programming language has following two usages:

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

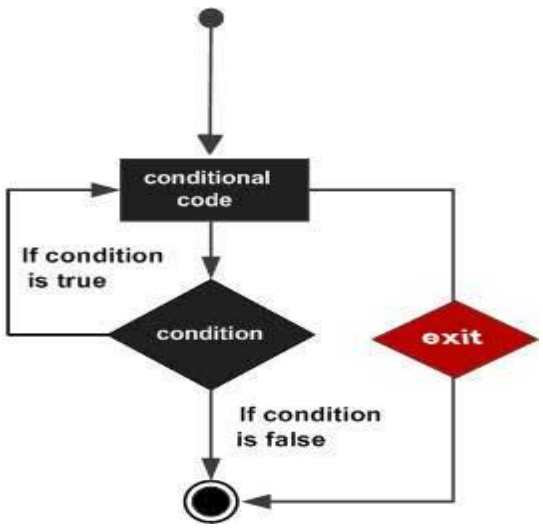
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for an EXIT statement in PL/SQL is as follows:

```
EXIT;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a > 15 THEN
         -- terminate the loop using the exit statement
         EXIT;
      END IF;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```

The EXIT WHEN Statement

The **EXIT-WHEN** statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after END LOOP.

Following are two important aspects for the EXIT WHEN statement:

- Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.

- A statement inside the loop must change the value of the condition.

Syntax:

The syntax for an EXIT WHEN statement in PL/SQL is as follows:

```
EXIT WHEN condition;
```

The EXIT WHEN statement **replaces a conditional statement like if-then** used with the EXIT statement.

Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
```

```
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      -- terminate the loop using the exit when statement
   EXIT WHEN a > 15;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15


PL/SQL procedure successfully completed.
```

CONTINUE statement

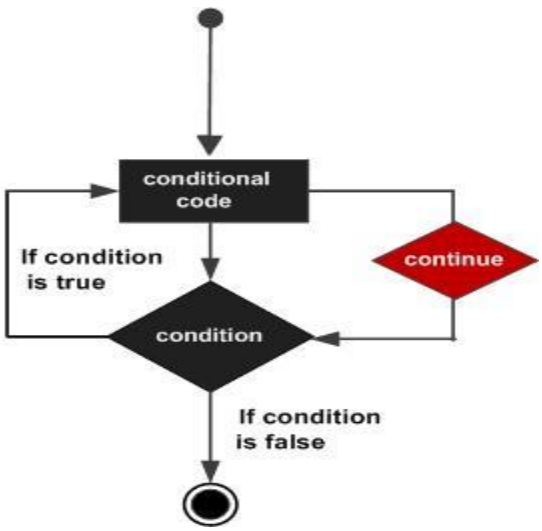
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax:

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a = 15 THEN
         -- skip the loop using the CONTINUE statement
         a := a + 1;
         CONTINUE;
      END IF;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

GOTO statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

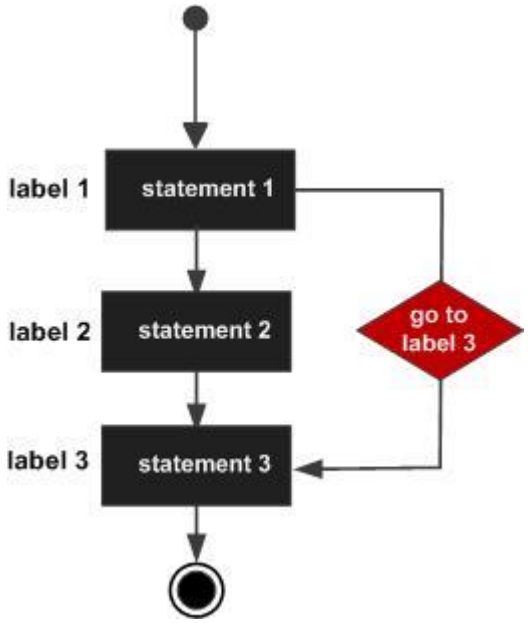
NOTE: Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;
..
..
<< label >>
statement;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   <<loopstart>>
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a = 15 THEN
         a := a + 1;
         GOTO loopstart;
      END IF;
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions:

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

CHAPTER

10

Strings

This chapter describes the concepts under strings:

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings:

- **Fixed-length strings**: In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- **Variable-length strings**: In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs)**: These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

```
'This is a string literal.' Or 'hello world'
```

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

```
'this isn''t what it looks like'
```

Declaring String Variables

Oracle database provides numerous string datatypes , like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```
DECLARE
   name varchar2(20);
   company varchar2(30);
```

```
   introduction clob;
   choice char(1);
BEGIN
   name := 'John Smith';
   company := 'Infotech';
   introduction := ' Hello! I''m John Smith from Infotech.';
   choice := 'y';
   IF choice = 'y' THEN
      dbms_output.put_line(name);
      dbms_output.put_line(company);
      dbms_output.put_line(introduction);
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed
```

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

```
 red_flag CHAR(1) := 'Y';
 red_flag CHAR    := 'Y';
```

PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator (||) for joining two strings. The following table provides the string functions provided by PL/SQL:

S.N.	Function & Purpose
1	**ASCII(x);** Returns the ASCII value of the character x.
2	**CHR(x);** Returns the character with the ASCII value of x.
3	**CONCAT(x, y);** Concatenates the strings x and y and return the appended string.
4	**INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.
5	**INSTR(x, find_string [, start] [, occurrence]);** Searches for find_string in x and returns the position at which it occurs.
6	**INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
7	**LENGTH(x);** Returns the number of characters in x.

8	**LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.
9	**LOWER(x);** Converts the letters in x to lowercase and returns that string.
10	**LPAD(x, width [, pad_string]) ;** Pads x with spaces to left, to bring the total length of the string up to width characters.
11	**LTRIM(x [, trim_string]);** Trims characters from the left of x.
12	**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise x is returned.
13	**NLS_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.
14	**NLS_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	**NLS_UPPER(x);** Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16	**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	**NVL(x, value);** Returns value if x is null; otherwise, x is returned.
18	**NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.
19	**REPLACE(x, search_string, replace_string);** Searches x for search_string and replaces it with replace_string.
20	**RPAD(x, width [, pad_string]);** Pads x to the right.
21	**RTRIM(x [, trim_string]);** Trims x from the right.
22	**SOUNDEX(x) ;** Returns a string containing the phonetic representation of x.
23	**SUBSTR(x, start [, length]);** Returns a substring of x that begins at the position specified by start. An optional length for the substring may be supplied.
24	**SUBSTRB(x);** Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	**TRIM([trim_char FROM) x);** Trims characters from the left and right of x.
26	**UPPER(x);** Converts the letters in x to uppercase and returns that string.

The following examples illustrate some of the above-mentioned functions and their use:

Example 1

```
DECLARE
   greetings varchar2(11) := 'hello world';
BEGIN
   dbms_output.put_line(UPPER(greetings));

   dbms_output.put_line(LOWER(greetings));

   dbms_output.put_line(INITCAP(greetings));

   /* retrieve the first character in the string */
   dbms_output.put_line ( SUBSTR (greetings, 1, 1));

   /* retrieve the last character in the string */
   dbms_output.put_line ( SUBSTR (greetings, -1, 1));

   /* retrieve five characters,
      starting from the seventh position. */
   dbms_output.put_line ( SUBSTR (greetings, 7, 5));

   /* retrieve the remainder of the string,
      starting from the second position. */
   dbms_output.put_line ( SUBSTR (greetings, 2));

   /* find the location of the first "e" */
   dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2

PL/SQL procedure successfully completed.
```

Example 2

```
DECLARE
   greetings varchar2(30) := '......Hello World.....';
BEGIN
   dbms_output.put_line(RTRIM(greetings,'.'));
   dbms_output.put_line(LTRIM(greetings, '.'));
   dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
......Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

CHAPTER

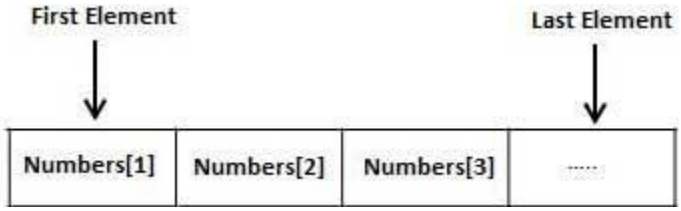
11

Arrays

This chapter describes concepts under Arrays:

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VRRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- *varray_type_name* is a valid attribute name,

- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/

Type created.
```

The basic syntax for creating a VRRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

Example 1

The following program illustrates using varrays:

```
DECLARE
   type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER;
   names namesarray;
   marks grades;
   total integer;
BEGIN
   names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i in 1 .. total LOOP
      dbms_output.put_line('Student: ' || names(i) || '
      Marks: ' || marks(i));
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Student: Kavita  Marks: 98
Student: Pritam  Marks: 97
Student: Ayan  Marks: 78
Student: Rishav  Marks: 87
Student: Aziz  Marks: 92

PL/SQL procedure successfully completed.
```

Please note:

- In oracle environment, the starting index for varrays is always 1.

- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.

- Varrays are one-dimensional arrays.

- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

.Example 2

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept:

We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

Following example makes use of **cursor,** which you will study in detail in a separate chapter.

```
DECLARE
   CURSOR c_customers is
   SELECT  name FROM customers;
   type c_list is varray (6) of customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter + 1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter
||'):'||name_list(counter));
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```

CHAPTER

12

Procedures

This chapter describes the procedures under PL/SQL:

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

- **Functions**: these subprograms return a single value, mainly used to compute and return a value.
- **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.
  This chapter is going to cover important aspects of a **PL/SQL procedure** and we will cover **PL/SQL function** in next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

S.N.	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When above code is executed using SQL prompt, it will produce the following result:

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as:

```
EXECUTE greetings;
```

The above call would display:

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block:

```
BEGIN
   greetings;
END;
/
```

The above call would display:

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

```
DROP PROCEDURE procedure-name;
```

So you can drop *greetings* procedure by using the following statement:

```
BEGIN
   DROP PROCEDURE greetings;
END;
/
```

Parameter Modes in PL/SQL Subprograms

S.N.	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also

	initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference.**
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
2	**IN OUT** An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
```

```
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters could be passed in three ways:

- Positional notation
- Named notation
- Mixed notation

POSITIONAL NOTATION

In positional notation, you can call the procedure as:

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

NAMED NOTATION

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (=>). So the procedure call would look like:

```
findMin(x=>a, y=>b, z=>c, m=>d);
```

MIXED NOTATION

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal:

```
findMin(a, b, c, m=>d);
```

But this is not legal:

```
findMin(x=>a, b, c, d);
```

CHAPTER

13

Functions

This chapter describes the functions:

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- *RETURN* clause specifies that data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example:

The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table. We will use the CUSTOMERS table, which we had created in PL/SQL Variables chapter:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

When above code is executed using SQL prompt, it will produce the following result:

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Total no. of Customers: 6


PL/SQL procedure successfully completed.
```

Example:

The following is one more example which demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;

   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;

   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Maximum of (23,45): 78


PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively:

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

CHAPTER

14

Cursors

This chapter explains cursors under PL/SQL:

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected

	one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example:

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program would update the table and increase salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected:

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
6 customers selected


PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor involves four steps:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_customers;
```

Example:

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
      EXIT WHEN c_customers%notfound;
   END LOOP;
   CLOSE c_customers;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

CHAPTER

15

Records

This chapter describes Records in PL/SQL which is a data structure:

A PL/SQL **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book like, Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records:

- Table-based
- Cursor-based records
- User-defined records

Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursor-based** records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec
   FROM customers
   WHERE id = 5;

   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' || customer_rec.address);
   dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
```

```
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure successfully completed.
```

Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   CURSOR customer_cur is
      SELECT id, name, address
      FROM customers;
   customer_rec customer_cur%rowtype;
BEGIN
   OPEN customer_cur;
   LOOP
      FETCH customer_cur into customer_rec;
      EXIT WHEN customer_cur%notfound;
      DBMS_OUTPUT.put_line(customer_rec.id || ' ' ||
customer_rec.name);
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

PL/SQL procedure successfully completed.
```

User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject

- Book ID

Defining a Record

The record type is defined as:

```
TYPE
type_name IS RECORD
  ( field_name1  datatype1  [NOT NULL]  [:= DEFAULT EXPRESSION],
   field_name2   datatype2   [NOT NULL]  [:= DEFAULT EXPRESSION],
   ...
   field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT EXPRESSION);
record-name  type_name;
```

Here is the way you would declare the Book record:

```
DECLARE
TYPE books IS RECORD
(title  varchar(50),
    author  varchar(50),
    subject varchar(100),
    book_id   number);
book1 books;
book2 books;
```

Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of record:

```
DECLARE
   type books is record
      (title varchar(50),
       author varchar(50),
       subject varchar(100),
       book_id number);
   book1 books;
   book2 books;
BEGIN
   -- Book 1 specification
   book1.title  := 'C Programming';
   book1.author := 'Nuha Ali ';
   book1.subject := 'C Programming Tutorial';
   book1.book_id := 6495407;

   -- Book 2 specification
   book2.title := 'Telecom Billing';
   book2.author := 'Zara Ali';
   book2.subject := 'Telecom Billing Tutorial';
   book2.book_id := 6495700;

   -- Print book 1 record
   dbms_output.put_line('Book 1 title : '|| book1.title);
   dbms_output.put_line('Book 1 author : '|| book1.author);
   dbms_output.put_line('Book 1 subject : '|| book1.subject);
   dbms_output.put_line('Book 1 book_id : ' || book1.book_id);
```

```
   -- Print book 2 record
   dbms_output.put_line('Book 2 title : '|| book2.title);
   dbms_output.put_line('Book 2 author : '|| book2.author);
   dbms_output.put_line('Book 2 subject : '|| book2.subject);
   dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

PL/SQL procedure successfully completed.
```

Records as Subprogram Parameters

You can pass a record as a subprogram parameter in very similar way as you pass any other variable. You would access the record fields in the similar way as you have accessed in the above example:

```
DECLARE
   type books is record
      (title  varchar(50),
      author  varchar(50),
      subject varchar(100),
      book_id   number);
   book1 books;
   book2 books;

PROCEDURE printbook (book books) IS
BEGIN
   dbms_output.put_line ('Book  title :  ' || book.title);
   dbms_output.put_line('Book  author : ' || book.author);
   dbms_output.put_line( 'Book  subject : ' || book.subject);
   dbms_output.put_line( 'Book book_id : ' || book.book_id);
END;

BEGIN
   -- Book 1 specification
   book1.title  := 'C Programming';
   book1.author := 'Nuha Ali ';
   book1.subject := 'C Programming Tutorial';
   book1.book_id := 6495407;

   -- Book 2 specification
   book2.title := 'Telecom Billing';
   book2.author := 'Zara Ali';
   book2.subject := 'Telecom Billing Tutorial';
   book2.book_id := 6495700;

   -- Use procedure to print book info
   printbook(book1);
```

```
   printbook(book2);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Book  title : C Programming
Book  author : Nuha Ali
Book subject : C Programming Tutorial
Book  book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700

PL/SQL procedure successfully completed.
```

CHAPTER

16

Exceptions

This chapter describes error conditions under PL/SQL:

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here, you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
     ........
   WHEN others THEN
      exception3-handling-statements
END;
```

Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   c_id customers.id%type := 8;
   c_name  customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;

   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND,** which is captured in **EXCEPTION** block.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
    WHEN exception_name THEN
   statement;
END;
```

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is:

```
DECLARE
   my-exception EXCEPTION;
```

Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name  customers.name%type;
   c_addr customers.address%type;

   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;

      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;
EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

Exception	Oracle Error	SQLCODE	Description

ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when s program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.

STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when s SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

CHAPTER

17

Triggers

This chapter describes Triggers under PL/SQL:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Example:

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
```

```
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Trigger created.
```

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in CUSTOMERS table, above create trigger display_salary_changes will be fired and it will display the following result:

```
Old salary:
New salary: 7500
```

```
Salary difference:
```

Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in CUSTOMERS table, above create trigger display_salary_changes will be fired and it will display the following result:

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

CHAPTER

18

Packages

This chapter describes Packages under PL/SQL:

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called public objects. Any subprogram not in the package specification but coded in the package body is called a private object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal* package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in PL/SQL - Variableschapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the find_sal method of the cust_sal package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 |
|  2 | Khilan   |  25 | Delhi     |  3000.00 |
|  3 | kaushik  |  23 | Kota      |  3000.00 |
|  4 | Chaitali |  25 | Mumbai    |  7500.00 |
|  5 | Hardik   |  27 | Bhopal    |  9500.00 |
|  6 | Komal    |  22 | MP        |  5500.00 |
+----+----------+-----+-----------+----------+
```

THE PACKAGE SPECIFICATION:

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customers.name%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at SQL prompt, it creates the above package and displays the following result:

```
Package created.
```

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id  customers.id%type,
      c_name customers.name%type,
      c_age  customers.age%type,
      c_addr  customers.address%type,
      c_sal   customers.salary%type)
   IS
   BEGIN
      INSERT INTO customers (id,name,age,address,salary)
         VALUES(c_id, c_name, c_age, c_addr, c_sal);
   END addCustomer;

   PROCEDURE delCustomer(c_id   customers.id%type) IS
   BEGIN
       DELETE FROM customers
         WHERE id = c_id;
   END delCustomer;
```

```
   PROCEDURE listCustomer IS
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list is TABLE OF customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
      FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer(' ||counter||
')'||name_list(counter));
      END LOOP;
   END listCustomer;
END c_package;
/
```

Above example makes use of **nested table** which we will discuss in the next chapter. When the above code is executed at SQL prompt, it produces the following result:

```
Package body created.
```

USING THE PACKAGE:

The following program uses the methods declared and defined in the package c_package.

```
DECLARE
   code customers.id%type:= 8;
BEGIN
      c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
         c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
         c_package.listcustomer;
      c_package.delcustomer(code);
      c_package.listcustomer;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed
```

CHAPTER

19

Collections

This chapter describes Collection under PL/SQL:

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections:

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

We have already discussed varray in the chapter 'PL/SQL arrays'. In this chapter, we will discuss PL/SQL tables.

Both types of PL/SQL tables, i.e., index-by tables and nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An index-by table (also called an associative array) is a set of key-value pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an index-by table namedtable_name whose keys will be of subscript_type and associated values will be of element_type.

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY
subscript_type;

table_name type_name;
```

Example:

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
   salary_list salary;
   name   VARCHAR2(20);
BEGIN
   -- adding elements to the table
   salary_list('Rajnish')  := 62000;
   salary_list('Minakshi')  := 75000;
   salary_list('Martin') := 100000;
   salary_list('James') := 78000;

   -- printing the table
   name := salary_list.FIRST;
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Salary of Rajnish is 62000
Salary of Minakshi is 75000
Salary of Martin is 100000
Salary of James is 78000

PL/SQL procedure successfully completed.
```

Example:

Elements of an index-by table could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+


DECLARE
   CURSOR c_customers is
      select  name from customers;

   TYPE c_list IS TABLE of customers.name%type INDEX BY binary_integer;
   name_list c_list;
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter||
'):'||name_list(counter));
  END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed
```

Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.

- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A **nested table** is created using the following syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;
```

This declaration is similar to declaration of an index-by table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example:

The following examples illustrate the use of nested table:

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;

   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
      dbms_output.put_line('Student:'||names(i)||', Marks:' ||
marks(i));
   end loop;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

PL/SQL procedure successfully completed.
```

Example:

Elements of a **nested table** could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;

+----+----------+-----+-----------+----------+
```

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+


DECLARE
   CURSOR c_customers is
      SELECT  name FROM customers;

   TYPE c_list IS TABLE of customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;

dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```

Collection Methods

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose:

S.N.	Method Name & Purpose
1	**EXISTS(n)** Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2	**COUNT** Returns the number of elements that a collection currently contains.
3	**LIMIT**

	Checks the Maximum Size of a Collection.
4	**FIRST** Returns the first (smallest) index numbers in a collection that uses integer subscripts.
5	**LAST** Returns the last (largest) index numbers in a collection that uses integer subscripts.
6	**PRIOR(n)** Returns the index number that precedes index n in a collection.
7	**NEXT(n)** Returns the index number that succeeds index n.
8	**EXTEND** Appends one null element to a collection.
9	**EXTEND(n)** Appends n null elements to a collection.
10	**EXTEND(n,i)** Appends n copies of the ith element to a collection.
11	**TRIM** Removes one element from the end of a collection.
12	**TRIM(n)** Removes n elements from the end of a collection.
13	**DELETE** Removes all elements from a collection, setting COUNT to 0.
14	**DELETE(n)** Removes the nth element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing.
15	**DELETE(m,n)** Removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

Collection Exceptions

The following table provides the collection exceptions and when they are raised:

Collection Exception	Raised in Situations
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the allowed range.

VALUE_ERROR	A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS_INTEGER range, and the subscript is outside this range.

CHAPTER

20

Transactions

This chapter describes the Transactions under PL/SQL:

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting an Ending a Transaction

A transaction has a beginning and an end. A transaction starts when one of the following events take place:

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place:

- A COMMIT or a ROLLBACK statement is issued.
- A DDL statement, like CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.

- SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
- A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is:

```
COMMIT;
```

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using savepoint, then simply use the following statement to rollback all the changes:

```
ROLLBACK;
```

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is:

```
SAVEPOINT < savepoint_name >;
```

For example:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;
COMMIT;
```

Here, **ROLLBACK TO sav1;** statement rolls sback the changes up to the point, where you had marked savepoint **sav1** and after that new changes will start.

Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as:

```
SET AUTOCOMMIT ON;
```

You can turn-off auto commit mode using the following command:

```
SET AUTOCOMMIT OFF;
```

CHAPTER

21

Date & Time

This chapter describes the Date & Time classes:

PL/SQL provides two classes of date and time related data types:

- Datetime data types
- Interval data types

The Datetime data types are:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

The Interval data types are:

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Field Values for Datetime and Interval Data Types

Both **datetime** and **interval** data types consist of **fields**. The values of these fields determine the value of the datatype. The following table lists the fields and their possible values for datetimes and intervals.

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer

MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds The 9(n) portion is not applicable for DATE.	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes) Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_MINUTE	00 to 59 Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_REGION	Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_ABBR	Not applicable for DATE or TIMESTAMP.	Not applicable

The Datetime Data Types and Functions

Following are the Datetime data types:

- **DATE** - it stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as:
- **TIMESTAMP** - it is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.
- **TIMESTAMP WITH TIME ZONE** - it is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This datatype is useful for collecting and evaluating date information across geographic regions.

- **TIMESTAMP WITH LOCAL TIME ZONE** - it is another variant of TIMESTAMP that includes a time zone offset in its value.

Following table provides the Datetime functions (where, x has datetime value):

S.N	Function Name & Description
1	**ADD_MONTHS(x, y);** Adds y months to x.
2	**LAST_DAY(x);** Returns the last day of the month.
3	**MONTHS_BETWEEN(x, y);** Returns the number of months between x and y.
4	**NEXT_DAY(x, day);** Returns the datetime of the next *day* after x.
5	**NEW_TIME;** Returns the time/day value from a time zone specified by the user.
6	**ROUND(x [, unit]);** Rounds x;
7	**SYSDATE();** Returns the current datetime.
8	**TRUNC(x [, unit]);** Truncates x.

Timestamp functions (where, x has a timestamp value):

S.N	Function Name & Description
1	**CURRENT_TIMESTAMP();** Returns a TIMESTAMP WITH TIME ZONE containing the current session time along with the session time zone.
2	**EXTRACT({ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND } | { TIMEZONE_HOUR | TIMEZONE_MINUTE } | { TIMEZONE_REGION | } TIMEZONE_ABBR) FROM x)** Extracts and returns a year, month, day, hour, minute, second, or time zone from x;
3	**FROM_TZ(x, time_zone);** Converts the TIMESTAMP x and time zone specified by time_zone to a TIMESTAMP WITH TIMEZONE.

4	**LOCALTIMESTAMP();** Returns a TIMESTAMP containing the local time in the session time zone.
5	**SYSTIMESTAMP();** Returns a TIMESTAMP WITH TIME ZONE containing the current database time along with the database time zone.
6	**SYS_EXTRACT_UTC(x);** Converts the TIMESTAMP WITH TIMEZONE x to a TIMESTAMP containing the date and time in UTC.
7	**TO_TIMESTAMP(x, [format]);** Converts the string x to a TIMESTAMP.
8	**TO_TIMESTAMP_TZ(x, [format]);** Converts the string x to a TIMESTAMP WITH TIMEZONE.

Examples:

The following code snippets illustrate the use of the above functions:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
08/31/2012 5:25:34 PM
```

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS') FROM DUAL;
```

Output:

```
31-08-2012 05:26:14
```

```
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output:

```
01/31/2013 5:26:31 PM
```

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output:

```
8/31/2012 5:26:55.347000 PM
```

The Interval Data Types and Functions

Following are the Interval data types:

- INTERVAL YEAR TO MONTH - it stores a period of time using the YEAR and MONTH datetime fields.
- INTERVAL DAY TO SECOND - it stores a period of time in terms of days, hours, minutes, and seconds.

Interval functions:

S.N	Function Name & Description
1	**NUMTODSINTERVAL(x, interval_unit);** Converts the number x to an INTERVAL DAY TO SECOND.
2	**NUMTOYMINTERVAL(x, interval_unit);** Converts the number x to an INTERVAL YEAR TO MONTH.
3	**TO_DSINTERVAL(x);** Converts the string x to an INTERVAL DAY TO SECOND.
4	**TO_YMINTERVAL(x);** Converts the string x to an INTERVAL YEAR TO MONTH.

CHAPTER

22

DBMS Output

This chapter describes the built-in-package DBMS_OUTPUT:

The **DBMS_OUTPUT** is a built-in package that enables you to display output, display debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers. We have already used this package all throughout our tutorial.

Let us look at a small code snippet that would display all the user tables in the database. Try it in your database to list down all the table names:

```
BEGIN
   dbms_output.put_line  (user || ' Tables in the database:');
   FOR t IN (SELECT table_name FROM user_tables)
   LOOP
      dbms_output.put_line(t.table_name);
   END LOOP;
END;
/
```

DBMS_OUTPUT Subprograms

The DBMS_OUTPUT package has the following subprograms:

S. N	Subprogram & Purpose
1	**DBMS_OUTPUT.DISABLE;** Disables message output
2	**DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 20000);** Enables message output. A NULL value of buffer_size represents unlimited buffer size.
3	**DBMS_OUTPUT.GET_LINE (line OUT VARCHAR2, status OUT INTEGER);** Retrieves a single line of buffered information.

4	**DBMS_OUTPUT.GET_LINES (lines OUT CHARARR, numlines IN OUT INTEGER);** Retrieves an array of lines from the buffer.
5	**DBMS_OUTPUT.NEW_LINE;** Puts an end-of-line marker
6	**DBMS_OUTPUT.PUT(item IN VARCHAR2);** Places a partial line in the buffer.
7	**DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2);** Places a line in the buffer.

Example:

```
DECLARE
   lines dbms_output.chararr;
   num_lines number;
BEGIN
   -- enable the buffer with default size 20000
   dbms_output.enable;

   dbms_output.put_line('Hello Reader!');
   dbms_output.put_line('Hope you have enjoyed the tutorials!');
   dbms_output.put_line('Have a great time exploring pl/sql!');

   num_lines := 3;

   dbms_output.get_lines(lines, num_lines);

   FOR i IN 1..num_lines LOOP
      dbms_output.put_line(lines(i));
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!

PL/SQL procedure successfully completed.
```

CHAPTER

23

Object Oriented

This chapter describes the Object oriented concept:

PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An object type allows you to crate composite types. Using objects allow you implementing real world objects with specific structure of data and methods for operating it. Objects have attributes and methods. Attributes are properties of an object and are used for storing an object's state; and methods are used for modeling its behaviors.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Below is an example to create a simple **address** object consisting of few attributes:

```
CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
 street varchar2(30),
 city varchar2(20),
 state varchar2(10),
 pincode varchar2(10)
);
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Let's create one more object **customer** where we will wrap **attributes** and **methods** together to have object oriented feeling:

```
CREATE OR REPLACE TYPE customer AS OBJECT
(code number(5),
 name varchar2(30),
 contact_no varchar2(12),
 addr address,
 member procedure display
);
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Instantiating an Object

Defining an object type provides a blueprint for the object. To use this object, you need to create instances of this object. You can access the attributes and methods of the object using the instance name and **the access operator (.)** as follows:

```
DECLARE
   residence address;
BEGIN
   residence := address('103A', 'M.G.Road', 'Jaipur',
'Rajasthan','201301');
   dbms_output.put_line('House No: '|| residence.house_no);
   dbms_output.put_line('Street: '|| residence.street);
   dbms_output.put_line('City: '|| residence.city);
   dbms_output.put_line('State: '|| residence.state);
   dbms_output.put_line('Pincode: '|| residence.pincode);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
House No: 103A
Street: M.G.Road
City: Jaipur
State: Rajasthan
Pincode: 201301

PL/SQL procedure successfully completed.
```

Member Methods

Member **methods** are used for manipulating the **attributes** of the object. You provide the declaration of a member method while declaring the object type. The object body defines the code for the member methods. The object body is created using the CREATE TYPE BODY statement.

Constructors are functions that return a new object as its value. Every object has a system defined constructor method. The name of the constructor is same as the object type. For example:

```
residence := address('103A', 'M.G.Road', 'Jaipur',
'Rajasthan','201301');
```

The **comparison methods** are used for comparing objects. There are two ways to compare objects:

- **Map method**: The **Map method** is a function implemented in such a way that its value depends upon the value of the attributes. For example, for a customer object, if the customer code is same for two customers, both customers could be the same and one. So the relationship between these two objects would depend upon the value of code.
- **Order method**: The **Order methods** implement some internal logic for comparing two objects. For example, for a rectangle object, a rectangle is bigger than another rectangle if both its sides are bigger.

Using Map method

Let us try to understand above concepts using the following rectangle object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member function enlarge( inc number) return rectangle,
 member procedure display,
 map member function measure return number
);
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Creating the type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER FUNCTION enlarge(inc number) return rectangle IS
   BEGIN
      return rectangle(self.length + inc, self.width + inc);
   END enlarge;

   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;

   MAP MEMBER FUNCTION measure return number IS
   BEGIN
      return (sqrt(length*length + width*width));
   END measure;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type body created.
```

Now using the rectangle object and its member functions:

```
DECLARE
   r1 rectangle;
   r2 rectangle;
   r3 rectangle;
   inc_factor number := 5;
BEGIN
   r1 := rectangle(3, 4);
   r2 := rectangle(5, 7);
   r3 := r1.enlarge(inc_factor);
   r3.display;
```

```
   IF (r1 > r2) THEN -- calling measure function
      r1.display;
   ELSE
      r2.display;
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Length: 8
Width: 9
Length: 5
Width: 7

PL/SQL procedure successfully completed.
```

Using Order method

Now, the **same effect could be achieved using an order method**. Let us recreate the rectangle object using an order method:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member procedure display,
 order member function measure(r rectangle) return number
);
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Creating the type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;

   ORDER MEMBER FUNCTION measure(r rectangle) return number IS
   BEGIN
      IF(sqrt(self.length*self.length + self.width*self.width)>
sqrt(r.length*r.length + r.width*r.width)) then
         return(1);
      ELSE
         return(-1);
      END IF;
   END measure;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type body created.
```

Using the rectangle object and its member functions:

```
DECLARE
   r1 rectangle;
   r2 rectangle;
BEGIN
   r1 := rectangle(23, 44);
   r2 := rectangle(15, 17);
   r1.display;
   r2.display;
   IF (r1 > r2) THEN -- calling measure function
      r1.display;
   ELSE
      r2.display;
   END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Length: 23
Width: 44
Length: 15
Width: 17
Length: 23
Width: 44

PL/SQL procedure successfully completed.
```

Inheritance for PL/SQL Objects:

PL/SQL allows creating object from existing base objects. To implement inheritance, the base objects should be declared as NOT FINAL. The default is FINAL.

The following programs illustrate inheritance in PL/SQL Objects. Let us create another object named TableTop, which is inheriting from the Rectangle object. Creating the base *rectangle* object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member function enlarge( inc number) return rectangle,
 NOT FINAL member procedure display) NOT FINAL
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Creating the base type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER FUNCTION enlarge(inc number) return rectangle IS
   BEGIN
      return rectangle(self.length + inc, self.width + inc);
   END enlarge;

   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type body created.
```

Creating the child object *tabletop*:

```
CREATE OR REPLACE TYPE tabletop UNDER rectangle
(
   material varchar2(20);
   OVERRIDING member procedure display
)
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```

Creating the type body for the child object tabletop:

```
CREATE OR REPLACE TYPE BODY tabletop AS
OVERRIDING MEMBER PROCEDURE display IS
BEGIN
   dbms_output.put_line('Length: '|| length);
   dbms_output.put_line('Width: '|| width);
   dbms_output.put_line('Material: '|| material);
END display;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type body created.
```

Using the tabletop object and its member functions:

```
DECLARE
   t1 tabletop;
   t2 tabletop;
BEGIN
   t1:= tabletop(20, 10, 'Wood');
   t2 := tabletop(50, 30, 'Steel');
   t1.display;
```

```
   t2.display;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Length: 20
Width: 10
Material: Wood
Length: 50
Width: 30
Material: Steel

PL/SQL procedure successfully completed.
```

Abstract Objects in PL/SQL

The NOT INSTANTIABLE clause allows you to declare an abstract object. You cannot use an abstract object as it is; you will have to create a subtype or child type of such objects to use its functionalities.

For example,

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 NOT INSTANTIABLE NOT FINAL MEMBER PROCEDURE display)
 NOT INSTANTIABLE NOT FINAL
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Type created.
```