Chapter 1: Databases and Database Users

Databases and database systems have become an essential component of everyday life in modern society. In the course of a day, most of us encounter several activities that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds; if we make a hotel or airline reservation; if we access a computerized library catalog to search for a bibliographic item; or if we order a magazine subscription from a publisher, chances are that our activities will involve someone accessing a database. Even purchasing items from a supermarket nowadays in many cases involves an automatic update of the database that keeps the inventory of supermarket items.

The above interactions are examples of what we may call **traditional database applications,** where most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have been leading to exciting new applications of database systems. **Multimedia databases** can now store pictures, video clips, and sound messages. **Geographic information systems (GIS)** can store and analyze maps, weather data, and satellite images. **Data warehouses** and **on-line analytical processing (OLAP)** systems are used in many companies to extract and analyze useful information from very large databases for decision making. **Real-time** and **active database technology** is used in controlling industrial and manufacturing processes. And database search techniques are being applied to the World Wide Web to improve the search for information that is needed by users browsing through the Internet.

1.1 Introduction

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name a few. The word *database* is in such common use that we must begin by defining a database. Our initial definition is quite general.

A **database** is a collection of related data (Note 1). By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL. This is a collection of related data with an implicit meaning and hence is a database.

A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **Universe of Discourse (UoD).** Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

A database can be of any size and of varying complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the card catalog of a large library may contain half a million cards stored under different categories.A database may be generated and maintained manually or it may be computerized.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, and *manipulating* databases for various applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database

includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

Example

Let us consider an example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 01.02 shows the database structure and a few sample data for such a database. The database is organized as five files, each of which stores data records of the same type (Note 2). The STUDENT file stores data on each student; the COURSE file stores data on each course; the SECTION file stores data on each section of a course; the GRADE_REPORT file stores the grades that students receive in the various sections they have completed; and the PREREQUISITE file stores the prerequisites of each course.

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

1.3 Characteristics of the Database Approach

1.3.1 Self-Describing Nature of a Database System

1.3.2 Insulation between Programs and Data, and Data Abstraction

1.3.3 Support of Multiple Views of the Data

1.3.4 Sharing of Data and Multiuser Transaction Processing

A number of characteristics distinguish the database approach from the traditional approach of programming with files. In traditional **file processing,** each user defines and implements the files needed for a specific application as part of programming the application. For example, one user, the *grade reporting office*, may keep a file on students and their grades. Programs to print a student's transcript and to enter new grades into the file are implemented. A second user, the accounting office, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common data up-to-date.

In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users.

The main characteristics of the database approach versus the file-processing approach are the following.

1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the system **catalog,** which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data,** and it describes the structure of the primary database.

Simplified database system environment

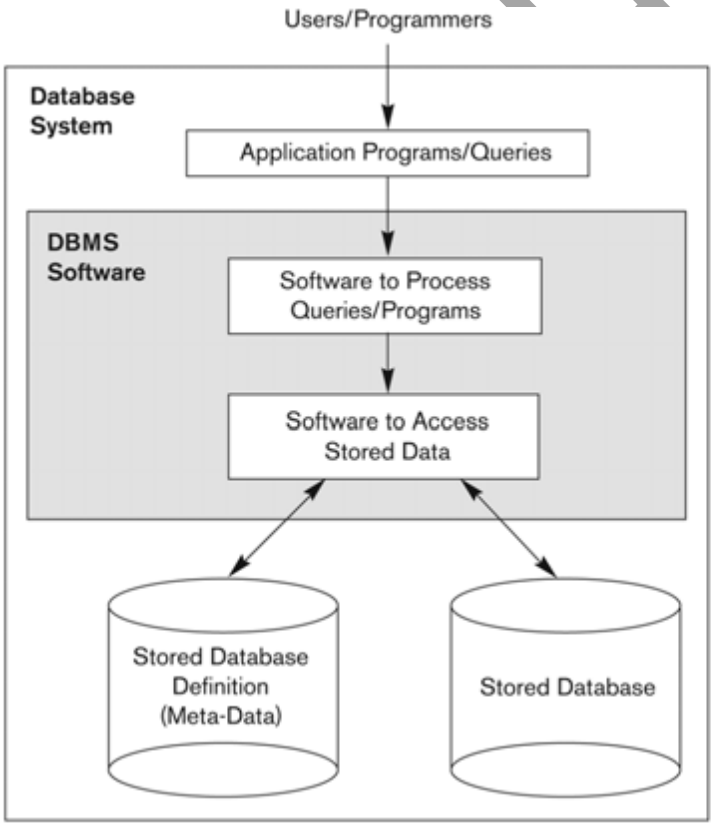


Figure 1.1
A simplified database system environment.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database*, whose structure is declared in the application programs. For example, a PASCAL program may have record structures declared in it; a C++ program may have "struct" or "class" declarations; and a COBOL program has Data Division statements to define its files. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and then using these definitions.

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require *changing all programs* that access this file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.**

In object-oriented and object-relational databases users can define operations on data as part of the database definitions. An **operation** (also called a *function*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence.**

The characteristic that allows program-data independence and program-operation independence is called **data abstraction. A data model** is a type of data abstraction that is used to provide this conceptual representation

1.3.3 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived.

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger. These types of applications are generally called **on-line transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly.

1.4 Actors on the Scene

1.4.1 Database Administrators

1.4.2 Database Designers

1.4.3 End Users

1.4.4 System Analysts and Application Programmers (Software Engineers)

1.4.1 Database Administrators

In any organization where many persons use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself and the

secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA).** The DBA is responsible for authorizing access to the database, for coordinating and monitoring its use, and for acquiring software and hardware resources as needed.

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. It is the responsibility of database designers to communicate with all prospective database users, in order to understand their requirements, and to come up with a design that meets these requirements.

1.4.3 End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.
- **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions—**that have been carefully programmed and tested.

  Bank tellers check account balances and post withdrawals and deposits.

- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.
- **Stand-alone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu- or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers (nowadays called **software engineers**) should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment.* These persons are typically not interested in the database itself. We call them the "workers behind the scene," and they include the following categories.

- **DBMS system designers and implementers** are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or **modules,** including modules for implementing the catalog, query language, interface processors, data access, concurrency control, recovery, and security.
- **Tool developers** include persons who design and implement **tools**—the software packages that facilitate database system design and use, and help improve performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation .
- **Operators and maintenance personnel** are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.6 Advantages of Using a DBMS

1.6.1 Controlling Redundancy

1.6.2 Restricting Unauthorized Access

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

1.6.4 Permitting Inferencing and Actions Using Rules

1.6.5 Providing Multiple User Interfaces

1.6.6 Representing Complex Relationships Among Data

1.6.7 Enforcing Integrity Constraints

1.6.8 Providing Backup and Recovery

1.6.9 Additional Implications of the Database Approach

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Much of the data is stored twice: once in the files of each user group. Additional user groups may further duplicate some or all of the same data in their own files.

This **redundancy** in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update—such as entering data on a new student—multiple times: once for each file where student data is recorded. This leads to *duplication of effort.* Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent.* This may happen because an update is applied to some of the files but not to others.

In the database approach, the views of different user groups are integrated during database design. For consistency, we should have a database design that stores each logical data item—such as a student's name or birth date—in *only one place* in the database. This does not permit inconsistency, and it saves storage space.

1.6.2 Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and to update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database.

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for the emergence of the **object-oriented database systems.** Programming languages typically have complex data structures, such as record types in PASCAL or class definitions in C++. The values of program variables are discarded once a program terminates.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem,** since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure **compatibility** with one or more object-oriented programming languages.

1.6.4 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems.** For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as **rules,** which when compiled and maintained by the DBMS can determine all students on probation.

1.6.5 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users; programming language interfaces for application programmers; forms and command codes for parametric users; and menu-driven interfaces and natural language interfaces for stand-alone users. Both forms-style interfaces and menu-driven interfaces are commonly known as **graphical user interfaces (GUIs).**

1.6.6 Representing Complex Relationships Among Data

A database may include numerous varieties of data that are interrelated in many ways. A DBMS must have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently.

1.6.7 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item.

For example we may specify that the value of the Class data item within each student record must be an integer between 1 and 5 and that the value of Name must be a string of no more than 30 alphabetic characters.

1.6.8 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update program, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the program started executing.

1.6.9 Additional Implications of the Database Approach

- Potential for Enforcing Standards
- Reduced Application Development Time
- Flexibility
- Availability of Up-to-Date Information
- Economies of Scale

Potential for Enforcing Standards

The Database approach permits the DBA to define and enforce standards among database users in a large organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on.

Reduced Application Development Time

A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application

Flexibility

It may be necessary to change the structure of a database as requirements change. Modern DBMSs allow certain types of changes to the structure of the database without affecting the stored data and the existing application programs.

Availability of Up-to-Date Information

A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases.

Economies of Scale

The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments.

1.8 When Not to Use a DBMS

In spite of the advantages of using a DBMS, there are a few situations in which such a system may involve unnecessary overhead costs as that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

- High initial investment in hardware, software, and training.
- Generality that a DBMS provides for defining and processing data.
- Overhead for providing security, concurrency control, recovery, and integrity functions.

A brief history of database applications

Early database systems: 1960's to 1970's and 1980's

The main types of early database systems were three types

1. Hierarchical
2. Network model
3. File systems

Large number of records of similar structure were stored and maintained in large organization.

❖ **Drawback**

1. There was intermixing a conceptual relationship with the physical storage and placement of records on disk. Although for original queries and transaction data access was efficient, it did not provide enough flexibility to access records efficiently when new queries and transactions were identified.
2. When changes were made to the requirements of the application, it was difficult to reorganize the database.
3. These systems only provide programming language interfaces.
4. Implementing new queries and transactions was time-consuming and expensive.

❖ **Relational databases:**

Relational databases could separate the physical storage of the data from its conceptual representation. It could provide a mathematical foundation for databases. It introduces high level query languages that provided an alternative to programming language interfaces.

It was developed in the late 1970's and the commercial RDBMS was introduced in the early 1980's.

→**Advantages**

1. Introduction of high level query language made it easier to write new queries and recognize database as required.
2. They did not use physical storage pointers and record placement to access related data records.
3. Performance greatly improved with the development of new storage and indexing.
4. Query processing became better.
5. They are widely used for traditional database systems.

❖ **Object-Oriented databases**

In the 1980's with the emergence of object oriented programming languages, it was necessary to store and share complex structured objects. This led to the development

of object oriented databases. They are used in specialized applications such as engineering design, multimedia publishing and manufacturing systems.

➔ **Advantages**

1. It provided more general data structures.
2. In incorporated many of the useful object oriented paradigms, such as ADT(Abstract Data Types), encapsulation, inheritance etc..

❖ **Web based database applications**

The World Wide Web is a large interconnection of a number of computer networks. User can create web documents (using HTML (Hyper Text Markup Language)) called web pages and store them on web servers from where other web clients can access. Documents can be linked together through hyperlinks, which are pointers to other documents.

❖ ***Advanced database applications***

Some of the advanced applications of database systems are

1. **Multimedia databases:** that can store pictures, video clips and sound messages.

2. **Geographic information systems (GIS):** that can store and analyze maps, weather data and satellite images.

3. **Data warehouse and online analytical processing (OLAP):** that are used in many companies extract and analyze useful information from very large databases for decision making.

4. **Real time and active database technology:** that is used in controlling industrial and manufacturing processes.

5. **Mobile databases:** they are available on the user portable computers, users interact with the mobile database application, which in turn accesses the data stored in the mobile databases through the DBMS.
6. **Web databases:** databases are integrated with web to support business operation like e-commerce supply chain management or web publishing.

7. **Spatial database:** provide concepts for database that keep track of objects in a multidimensional space. Ex: Data in CAD/CAM computers.

8. **Time series:** these applications store information such as economic data at regular points in time like daily sales or monthly sales.

Basic relational systems were not very suitable for many of these applications for the reasons:

1. More complex data structures were needed for modelling the application.
2. New data types were needed in addition to standard data types.
3. New operations and query language constructs were necessary manipulate with the new data types.
4. New storage and indexing structures were needed.

chapter 2: Database System Concepts and Architecture

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 .By *structure of a database* we mean the data types, relationships, and constraints that should hold on the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.
This allows the database designer to specify a set of valid **user-defined operations** that are allowed on the database objects .An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object.

2.1.1 Categories of Data Models

A)High-level or **conceptual data models** provide concepts that are close to the way many users perceive data.
B)low-level or **physical data models** provide concepts that describe the details of how data is stored in the computer.
C) **representational** (or **implementation**) **data models,** which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models represent data by using record structures and hence are sometimes called **record-based data models.**

An **entity** represents a real-world object or concept, such as an employee or a project, 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 interaction among the entities; for example, a works-on relationship between an employee and a project.

D)Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used **relational data model.**
E)legacy data models—the **network** and **hierarchical models.**
F) object data models are new family of higher-level implementation data models that are closer to conceptual data models.

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.
A displayed schema is called a **schema diagram.** 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.

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

Figure 2.1
Schema diagram for the database in Figure 1.2.

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 database.

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

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 the 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 an **extension** of the schema.

2.2 DBMS Architecture and Data Independence

In this section we specify an architecture for database systems, called the **three-schema architecture** which was proposed to help achieve and visualize the DBMS characteristics.

2.2.1 The Three-Schema Architecture

Proposed to support DBMS characteristics of:

. **Program-data independence.**

.Support of **multiple views** of the data.

.Not explicitly used in commercial DBMS products,but has been useful in explaining database system organization

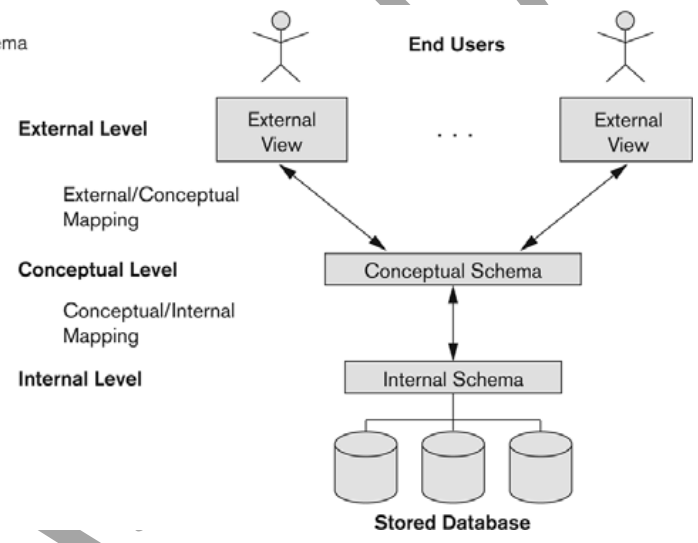


Figure 2.2
The three-schema architecture.

The goal of the three-schema architecture, illustrated in Figure.

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. A high-level data model or an implementation data model can be used at this level.
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. A high-level data model or an implementation data model can be used at this level.

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 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), or to reduce the database (by removing a record type or data item).

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be 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 .

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

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. the DDL is used to specify the conceptual schema only.

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.

View definition language (VDL) is used 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.

Data manipulation language (DML) 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 **data manipulation language (DML)** for these purposes. There are two main types of DMLs. A **high-level** or **nonprocedural DML** can be used on its own to specify complex database operations in a concise manner. 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. Low-level DMLs are also called **record-at-a-time DMLs.** High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement and are hence called **set-at-a-time** or **set-oriented DMLs.**

2.3.2 DBMS Interfaces

- Menu-Based Interfaces for Browsing
- Forms-Based Interfaces
- Graphical User Interfaces
- Natural Language Interfaces Interfaces for Parametric Users
- Interfaces for the DBA

Menu-Based Interfaces for 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. They are 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 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.

Graphical User Interfaces

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then 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 pick 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. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request.

Interfaces for Parametric Users

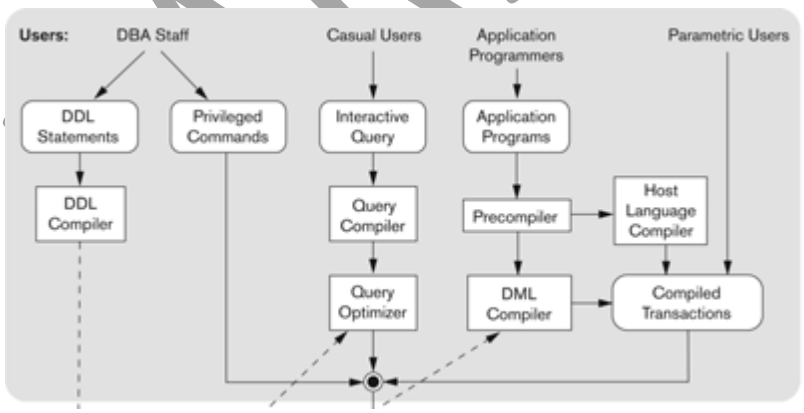
Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request

Interfaces for the DBA

Most database systems contain privileged commands that can be used only by the DBA's 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

2.4.1 DBMS Component Modules



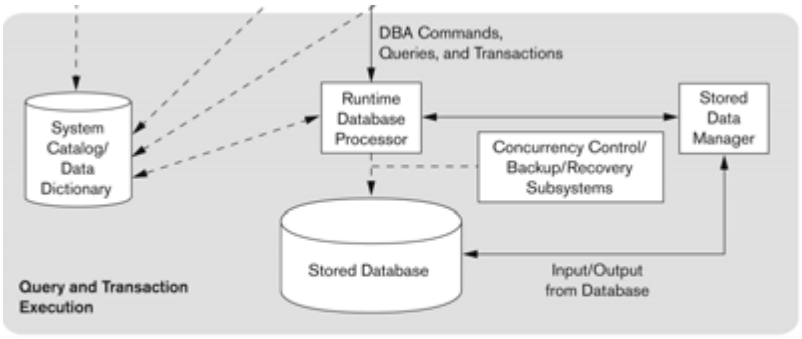


Figure 2.3
Component modules of a DBMS and their interactions.

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 input/output. 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. The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory.

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 of files, data items, storage details of each file and so on.

The **run-time database processor** handles database accesses at run time; it receives retrieval or update **query compiler** handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The **pre-compiler** 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.

2.4.2 Database System Utilities

Most DBMSs have **database utilities** that help the DBA in managing the database system.

Common utilities have the following types of functions:

1.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) format of the data file and the desired (target) database file structure are specified to the utility.

2. *Backup*: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but it saves space.

3. *File reorganization:* This utility can be used to reorganize a database file into a different file organization to improve performance.

4. *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 to improve performance .

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and DBAs. **CASE tools a**re 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.**

Application development environments, such as the PowerBuilder system, are becoming quite popular.

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 their local personal computers. These are connected to the database site through data communications hardware such as phone lines, long-haul networks, local-area networks, or satellite communication devices.

2.5 . DBMS architecture

a)Centralized architecture

.Combines everything into single system including-DBMS software, hardware, application programs, and user interface processing software.

.User can still connect through a remote terminal –however, all processing is done at centralized site.

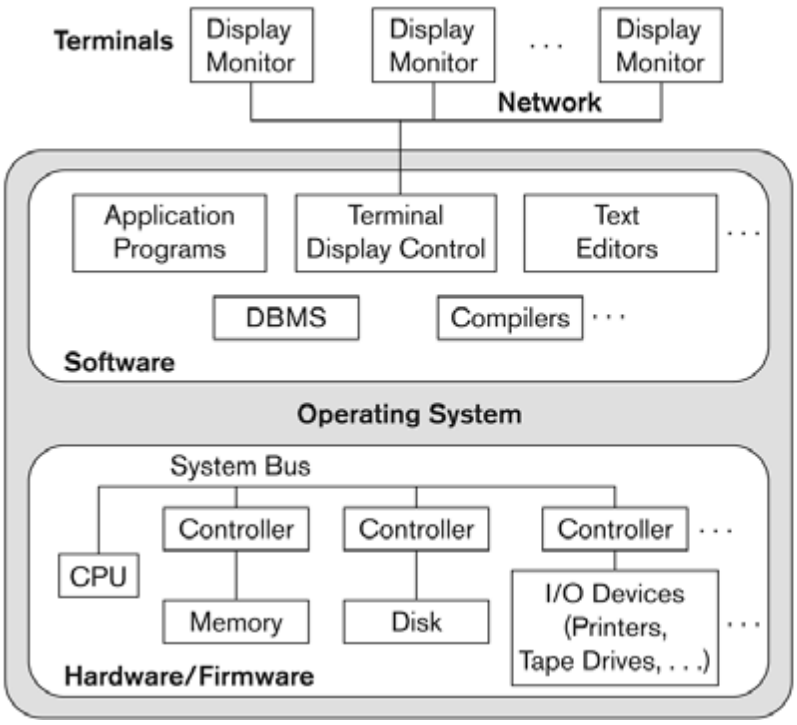


Figure 2.4
A physical centralized architecture.

b)Basic 2-tier Client-Server Architectures

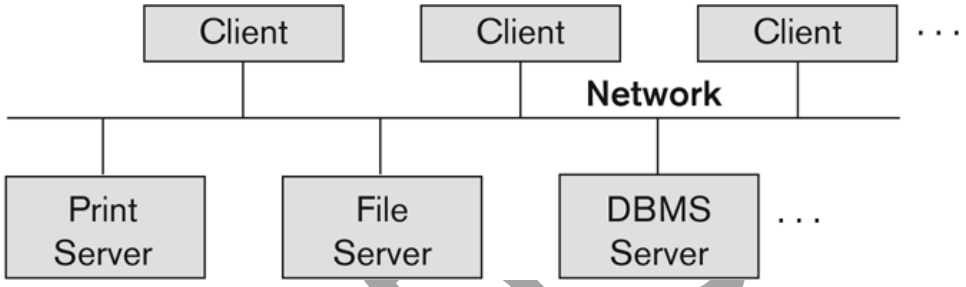
Specialized Servers with Specialized functions

Print server

File server
DBMS server
Web server
Email server
Clients can access the specialized servers as Needed

Logical two-tier client server architecture

Figure 2.5
Logical two-tier client/server architecture.



Clients

- Provide appropriate interfaces through a client software module to access and utilize the various servers resources.
  Clients may be diskless machines or PCs or Workstations with disks with only the client software installed.
- Connected to the servers via some form of a network.

  (LAN: local area network, wireless network, etc.)

DBMS SERVER

. Provides database query and transaction services to the clients..
.Relational DBMS servers are often called SQL servers, query servers, or transaction servers
Applications running on clients utilize an Application program Interface (**API**) to access server databases via standard interface such as:
 ODBC: Open Database Connectivity standard
 JDBC: for Java programming access
. Client and server must install appropriate client module and server module software for ODBC or JDBC

A client program may connect to several DBMSs, sometimes called the data sources.
 In general, data sources can be files or other non-DBMS software that manages data.
 Other variations of clients are possible: e.g., in some object DBMSs, more functionality is transferred To clients including data dictionary functions, optimization and recovery across multiple servers, etc.'

c)Three-tier client /server Architecture

Common for Web applications

1)Intermediate Layer called Application Server or WebServer:
.Stores the web connectivity software and the business logic part of the application used to access the corresponding data from the database server.
.Acts like a conduit for sending partially processed data between the database server and the client.

2).Three-tier Architecture Can Enhance Security:

.Database server only accessible via middle tier

. Clients cannot directly access database server

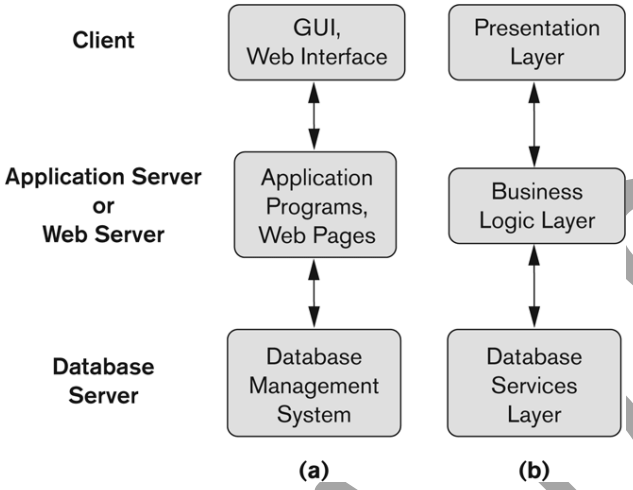


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

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 two types of data models used in many current commercial DBMSs are the **relational data model** and the **object data model.** Many legacy applications still run on database systems based on the **hierarchical** and **network data models.** 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 that are being called **object-relational DBMSs/.** We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

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 personal computers. **Multiuser systems,** which include the majority of DBMSs, support multiple users concurrently.

A 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 themselves 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 multiple sites.

A fourth criterion is the **cost** of the DBMS. The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. At the other end, a few elaborate packages cost more than $100,000.

A fifth criteria is 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 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.

Ex:Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **on-line transaction processing (OLTP) systems,** which must support a large number of concurrent transactions without imposing excessive delays.

Chapter 3: Data Modeling Using the Entity-Relationship Model

Conceptual modeling is an important phase in designing a successful database application. Generally, the term **database application** refers to a particular database—for example, a BANK database that keeps track of customer accounts—and the associated *programs* that implement the database queries and updates—for example, programs that implement database updates corresponding to customers making deposits and withdrawals. These programs often provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus. Hence, part of the database application will require the design, implementation, and testing of these **application programs.**

Database design methodologies attempt to include more of the concepts for specifying operations on database objects, and as software engineering methodologies specify in more detail the structure of the databases that software programs will use and access, it is certain that this commonality will increase .

Entity-Relationship (ER) model, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications.

1. Using High-Level Conceptual Data Models for Database Design

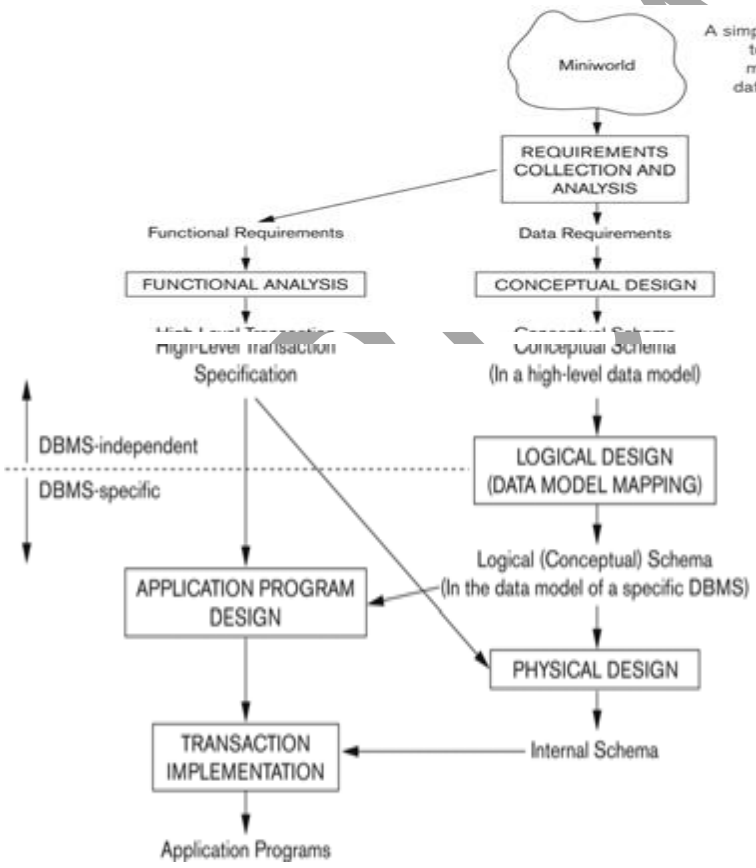


Figure 3.1
A simplified diagram to illustrate the main phases of database design.

Above Figure shows a simplified description of the database design process. The first step shown is **requirements collection and analysis.** During this step, the database designers interview prospective database users to understand and document their **data requirements.** The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with

specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, and they include both retrievals and updates.

Once all the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design.** The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users..

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified in the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping,** and its result is a database schema in the implementation data model of the DBMS.

Finally, the last step is the **physical design** phase, during which the internal storage structures, access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications

2 An Example Database Application

In this section we describe an example database application, called COMPANY, which serves to illustrate the ER model concepts and their use in schema design.

1. The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
2. A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
3. We store each employee's name, social security number (Note 1), address, salary, sex, and birth date. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.
4. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as entities, relationships, and attributes.

Entities

The basic object that the ER model represents is an **entity,** which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence—a particular person, car, house, or employee—or it may be an object with a conceptual existence—a company, a job, or a university course.

Attributes : Each entity has **attributes**—the particular properties that describe it. For example, an employee entity may be described by the employee's name, age, address, salary, and job.

A particular entity will have a **value** for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Ex: 1)The employee entity e_1 has four attributes: Name, Address, Age, and HomePhone; their values are "John Smith," "2311 Kirby, Houston, Texas 77001," "55," and "713-749-2630," respectively.

Several types of attributes occur in the ER model:

1) *simple* versus *composite*
2) *single-valued* versus *multi valued*
3) *stored* versus *derived.*

1)Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings.

Ex: Address attribute of the employee entity can be sub-divided into StreetAddress, City, State, and Zip with the values "2311 Kirby," "Houston," "Texas," and "77001."

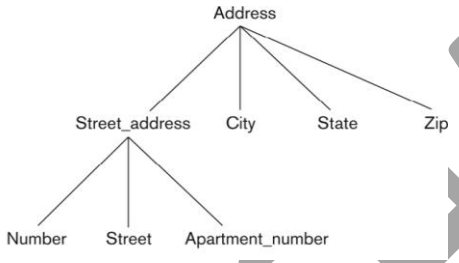


Figure 3.4
A hierarchy of composite attributes.

Composite attributes can form a hierarchy; for example, StreetAddress can be subdivided into three simple attributes, Number, Street, and ApartmentNumber, as shown in Figure .The value of a composite attribute is the concatenation of the values of its constituent simple attributes.

2)simple or **atomic attributes :**

Attributes that are not divisible are called **simple** or **atomic attributes.**

EX: Number, street are simple attributes

3)Single-valued

Most attributes have a single value for a particular entity; such attributes are called **single-valued.**

EX: Age is a single-valued attribute of person.

4) Multivalued Attributes

In some cases an attribute can have a set of values for the same entity—for example, a Colors attribute for a car, or a CollegeDegrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two values for Colors. A multivalued attribute may have lower and upper bounds on the number of values allowed for each individual entity.

Ex:The Colors attribute of a car may have between one and three values, if we assume that a car can have at most three colors.

5)Stored Versus Derived Attributes

In some cases two (or more) attribute values are related—for example, the Age and BirthDate attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's BirthDate. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the BirthDate attribute, which is called a **stored attribute.**

6)Null Values

In some cases a particular entity may not have an applicable value for an attribute. For example, the ApartmentNumber attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. For such situations, a special value called **null** is created. An address of a single-family home would have null for its ApartmentNumber attribute. Null can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone of "John Smith" .The meaning of the former type of null is *not applicable*, whereas the meaning of the latter is *unknown.*

The unknown category of null can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for example, if the Height attribute of a person is listed as null.

The second case arises when it is *not known* whether the attribute value exists—for example, if the HomePhone attribute of a person is null.

7)Complex Attributes

Notice that composite and multivalued attributes can be nested in an arbitrary way. We can represent arbitrary nesting by grouping components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces {}. Such attributes are called **complex attributes.**

1)For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by

{PreviousDegrees (College, Year, Degree, Field)}

2)If a person can have more than one residence and each residence can have multiple phones, an attribute AddressPhone for a PERSON entity type can be specified as complex attribute.

Entity Types

A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has *its own value(s)* for each attribute.

An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

Ex:Two entity types, named EMPLOYEE and COMPANY, and a list of attributes for each.

Entity Sets

The collection of all entities of a particular entity type in the database at any point in time is called an **entity set;** the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current *set of all employee entities* in the database.

An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type are grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type

An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the collection. Such an attribute is called a **key attribute,** and its values can be used to identify each entity uniquely.

Ex: 1)The Name attribute is a key of the COMPANY entity type , because no two companies are allowed to have the same name.

2)For the PERSON entity type, a typical key attribute is SocialSecurityNumber.

Sometimes, several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, we can define a *composite attribute* that becomes a key attribute of the entity type.

In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval.

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every extension* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time.

Some entity types have *more than one* key attribute.

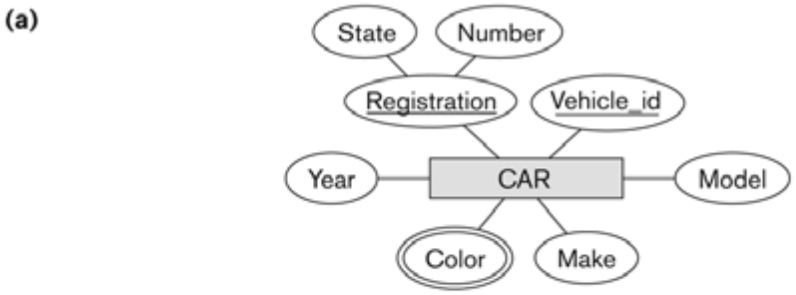


Figure 3.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

(b)

CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}

CAR_1
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004 {red, black})

CAR_2
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

CAR_3
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

.
.
.

For example, each of the VehicleID and Registration attributes of the entity type CAR is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, RegistrationNumber and State, neither of which is a key on its own. An entity type may also have *no key*, in which case it is called a *weak entity type* .

Value Sets (Domains) of Attributes

Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity.

Ex:The range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70.

Similarly, we can specify the value set for the Name attribute as being the set of strings of alphabetic characters separated by blank characters and so on. Value sets are not displayed in ER diagrams.

4 Relationships, Relationship Types, Roles, and Structural Constraints

There are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists.

For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department.In the ER model, these references should not be represented as attributes but as **relationships.**

4.1 Relationship Types, Sets and Instances

A **relationship type** *R* among *n* entity types , , . . ., defines a set of associations—or a **relationship set**—among entities from these types. As for entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name R.* Mathematically, the relationship set *R* is a set of **relationship instances .**

Informally, each relationship instance in *R* is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance represents the fact that the entities participating in are related in some way in the corresponding miniworld situation.

For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department the employee works for. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity.

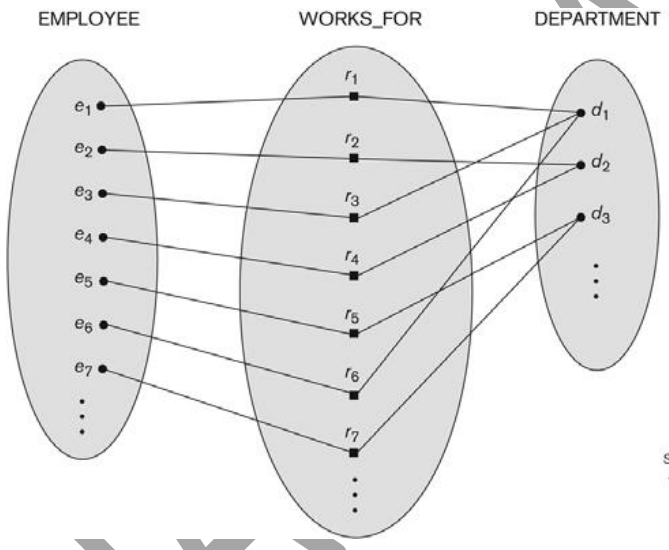


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

The above figure illustrates this example, where each relationship instance is shown connected to the employee and department entities that participate in . *E*mployees e_1, e_3, and e_6 work for department d_1; e_2 and e_4 work for d_2; and e_5 and e_7 work for d_3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box .

4.2 Degree of a Relationship Type

The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary,** and one of degree three is called **ternary.**

An example of a ternary relationship is SUPPLY, shown in above Figure . where each relationship instance associates three entities—a supplier *s*, a part *p*, and a project *j*—whenever *s* supplies part *p* to project *j*. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships.

Relationships as Attributes

It is sometimes convenient to think of a relationship type in terms of attributes. One can think of an attribute called Department of the EMPLOYEE entity type whose value for each employee entity is (a reference to) the *department entity* that the employee works for.

Role Names and Recursive Relationships

Each entity type that participates in a relationship type plays a particular **role** in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means.

For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of each participation. Such relationship types are called **recursive relationships.**

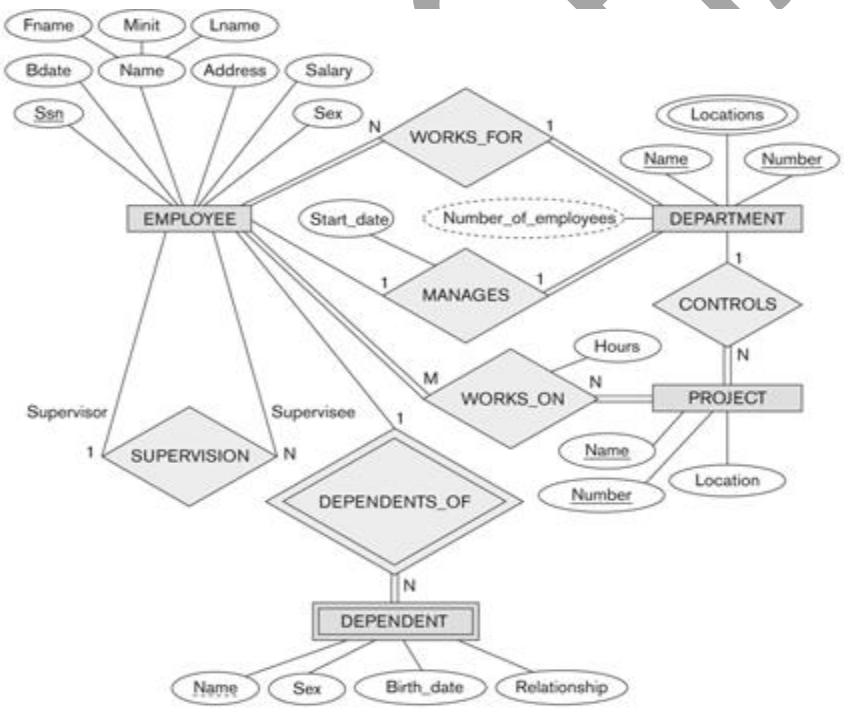


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation

The above figure shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or boss), and once in the role of *supervisee* (or subordinate).

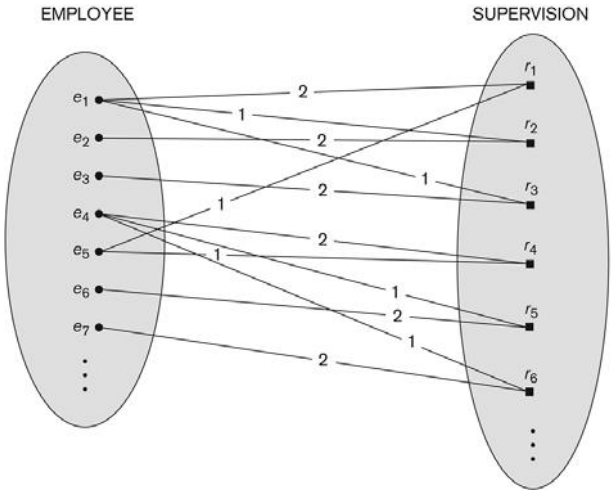


Figure 3.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

Each relationship instance in SUPERVISION associates two employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee.

In the above figure the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e_1 supervises e_2 and e_3; e_4 supervises e_6 and e_7; and e_5 supervises e_1 and e_4.

4.3 Constraints on Relationship Types

Cardinality Ratios for Binary Relationships Participation Constraints and Existence Dependencies.Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the mini world situation that the relationships represent.

For example, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of relationship constraints: *cardinality ratio* and *participation.*

Cardinality Ratios for Binary Relationships

cardinality ratio :

cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in.

For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to numerous employees but an employee can be related to only one department.

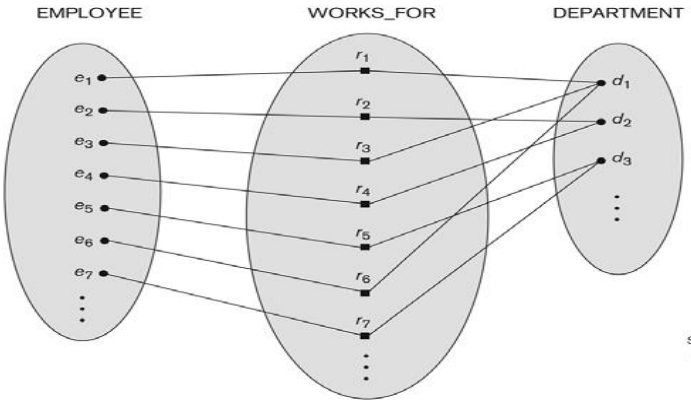


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES which relates a department entity to the employee who manages that department. This represents the miniworld constraints that an employee can manage only one department and that a department has only one manager.

The relationship type WORKS_ON is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.

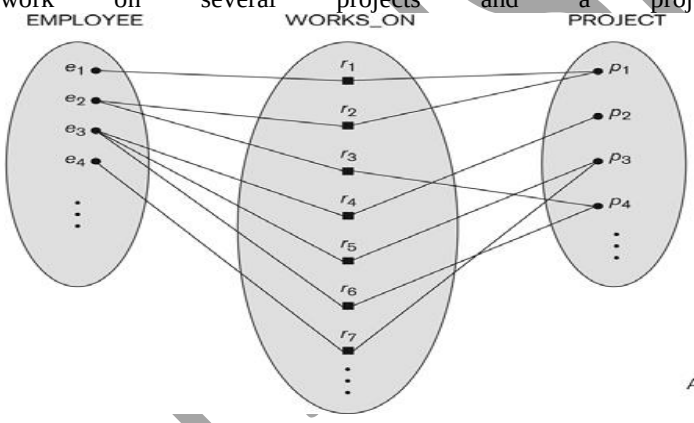


Figure 3.13
An M:N relationship, WORKS_ON.

Cardinality ratios for binary relationships are displayed on ER diagrams by displaying 1, M, and N on the diamonds .

Participation Constraints and Existence Dependencies

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. There are two types of participation constraints—**total** and **partial**—which we illustrate by example.

a)If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in a WORKS_FOR relationship instance .Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation,** meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency.**

b)we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial,** meaning that *some* or "part of the set of" employee entities are related to a department entity via MANAGES, but not necessarily all.

structural constraints

cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* .

Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute **Hours** for the WORKS_ON relationship type .

Another example is to include the date on which a manager started managing a department via an attribute **StartDate** for the MANAGES relationship type .

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the StartDate attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or DEPARTMENT—although conceptually it belongs to MANAGES. This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the StartDate attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type at the N-side of the relationship. For example if the WORKS_FOR relationship also has an attribute StartDate that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee entity participates in at most one relationship instance in WORKS_FOR. In both 1:1 and 1:N relationship types, the decision as to where a relationship attribute should be placed—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes *must be specified as relationship attributes.* An example is the Hours attribute of the M:N relationship WORKS_ON . the number of hours an employee works on a project is determined by an employee-project combination and not separately by either entity.

5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types.** In contrast, **regular entity types** that do have a key attribute are sometimes called **strong entity types.** Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attribute values. We call this other entity type the **identifying** or **owner entity type** and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type ..A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship .The attributes of DEPENDENT are Name (the first name of the dependent), BirthDate, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, BirthDate, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to **own** the dependent entities that are related to it.

A weak entity type normally has a **partial key,** which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity* .In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with *double lines*. The partial key attribute is underlined with a dashed or dotted line.

6 Refining the ER Design for the COMPANY Database

In our example, we specify the following relationship types:

1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation .The attribute StartDate is assigned to this relationship type.
2. WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users.
4. SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
6. DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

Summary of Notation for ER Diagrams

In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful because a database schema changes rarely, whereas the extension changes frequently. In addition, the schema is usually easier to display than the extension of a database, because it is much smaller.

The below Figure displays the COMPANY **ER database schema** as an ER diagram.

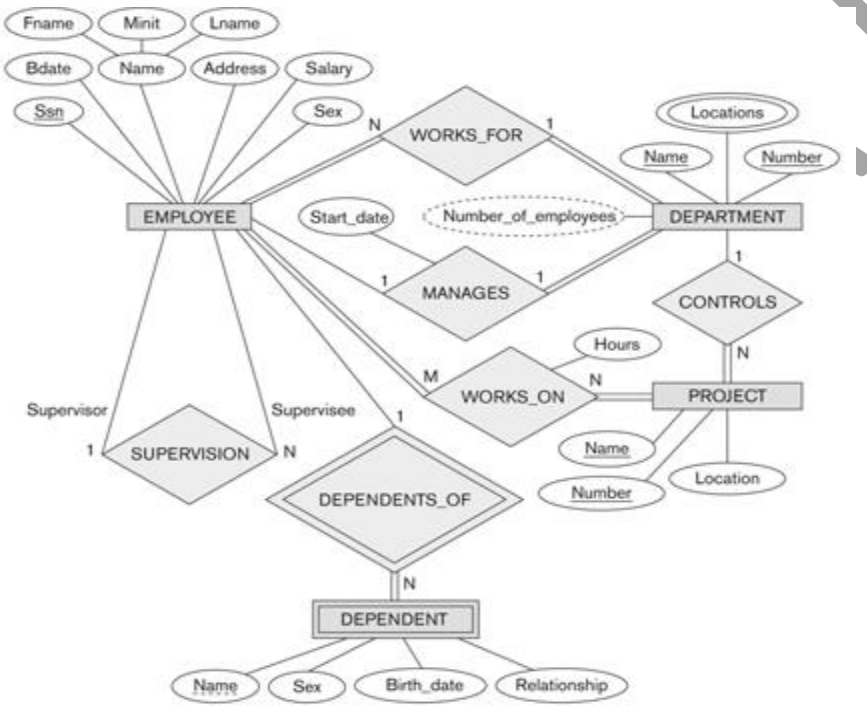


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation

Entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the NumberOfEmployees attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT: EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT:EMPLOYEE in WORKS_FOR, and it is M:N for WORKS_ON. The participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure we show the role names for the SUPERVISION relationship type because the EMPLOYEE entity type plays both roles in that relationship. Notice that the cardinality is 1:N from supervisor to supervisee because, on the one hand, each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.

Summary of notation for ER diagrams

Proper Naming of Schema Constructs

We use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type. In our ER diagrams, we will use the convention that entity type and relationship type names are in uppercase letters, attribute names are capitalized, and role names are in lowercase letters.

As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.

Another naming consideration involves choosing relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. One exception is the DEPENDENTS_OF relationship type, which reads from bottom to top. This is because we say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read: an EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type).

Design Choices for ER Conceptual Design

It is occasionally difficult to decide whether a particular concept in the miniworld should be modeled as an entity type, an attribute, or a relationship type. we give some brief guidelines as to which construct should be chosen in particular situations.

1. A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship.
2. Similarly, an attribute that exists in several entity types may be refined into its own independent entity type. For example, suppose that several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE each have an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute DeptName and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships.
3. An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute DeptName and related to only one other entity type STUDENT. In this case, DEPARTMENT may be refined into an attribute of STUDENT.

7.4 Alternative Notations for ER Diagrams

- This notation involves associating a pair of integer numbers (min,max) with each participation of an entity type E in a relationship type R,

  Where 0<=min<=max and max>=1
- Specified on each participation of an entity type E in a relationship type R ,Specifies that each entity e in E participates in at least min and at most max relationship instances in R at any point in time.
- In this method, min=0 implies partial participation, whereas min>0 implies total participation.
- **Examples:**

- A department has exactly one manager and an employee can manage at most one department.

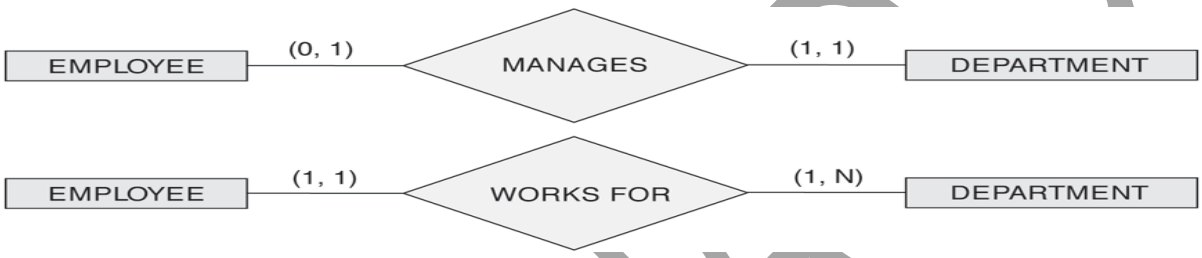
  Specify (0,1) for participation of EMPLOYEE in MANAGES

  Specify (1,1) for participation of DEPARTMENT in MANAGES

- An employee can work for exactly one department but a department can have any number of employees.

  Specify (1,1) for participation of EMPLOYEE in WORKS_FOR

  Specify (0,n) for participation of DEPARTMENT in WORKS_FOR



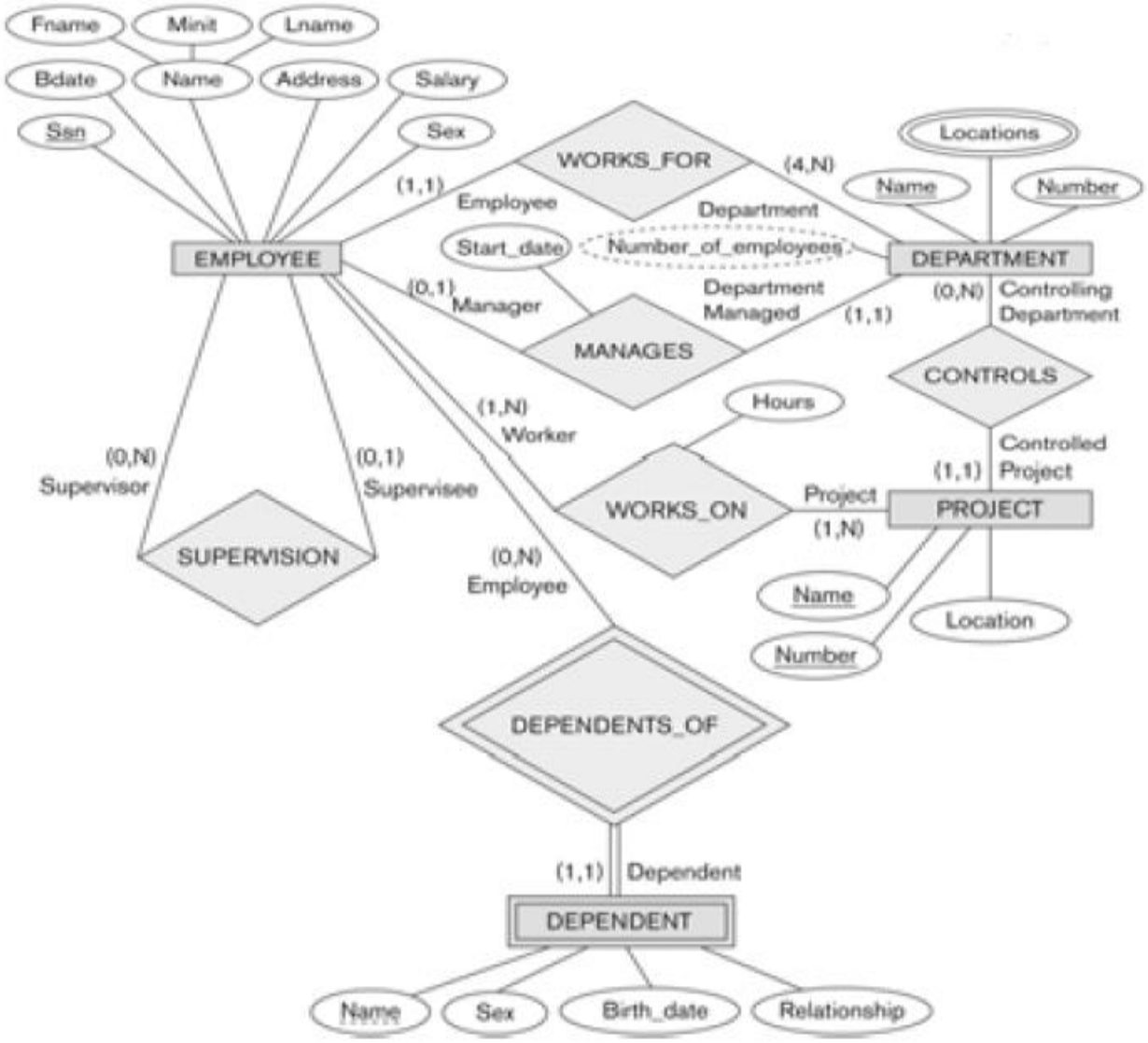


Figure: ER diagram for the company schema, with structural constraints specified using (min,max) notation and role names

8. Example of other notation: UML Class Diagrams

- **Universal Modeling Language** (UML) methodology:
  - Used extensively in software design
  - Many types of diagrams for various software design purposes
  - UML Class Diagrams correspond to E-R Diagram, but several differences.

UML class diagrams

- Entity types in ER corresponds to an class in UML
- Entity in ER corresponds to an object in UML

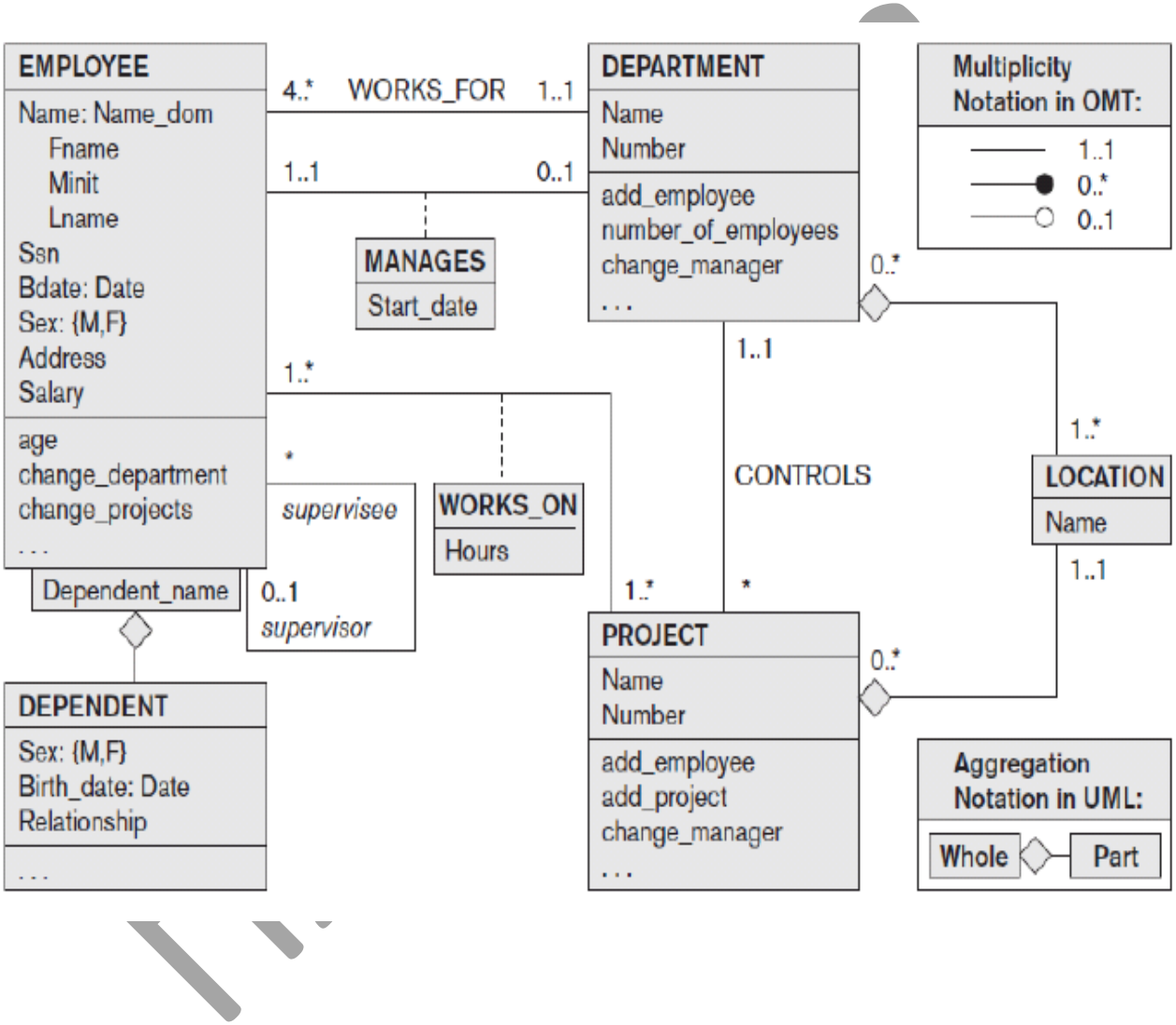


Figure: the COMPANY conceptual schema in UML class diagram notation

- A **Class** is displayed as a box that includes **three sections**:
- Top section gives the class name

- Middle section includes the attributes;
- Last section includes operations that can be applied to individual objects

- The designer can optionally specify the **domain** of an attribute if desired, by placing a colon (:) followed by the domain name or description.

Example: Name, Sex, and Bdate attributes of EMPLOYEE in above figure.

- A **composite attribute** is modeled as a structured domain, as illustrated by the name attribute of EMPLOYEE.
- A **multivalued attribute** be modeled as a separate class, as illustrated by the LOCATION class in above figure.

Types of relationships:

Associations

- Relationship instances called links.

Binary association

- Represented as a line connecting participating classes
- May optionally have a name
- A relationship attribute, called Link attribute is Placed in a box connected to the association's line by a dashed line.

Multiplicities: min..max

- Asterisk (*) indicates no maximum limit on participation
- The multiplicities are placed on the opposite ends of the relationship.
- A single asterisk(*) indicates a multiplicity of 0..*, and a single 1 indicates a multiplicity of 1..1
- A recursive relationship is called a reflexive association in UML, and the role names-like the multiplicities- are placed at the opposite ends of an association.

Aggregations

- Aggregation is meant to represent a relationship between a whole object and its component parts
- Locations of a department and the single location of a project as aggregations.

UML also distinguishes between unidirectional and bidirectional associations(or aggregations)

- In the unidirectional case, the line connecting the classes is displayed with an arrow to indicate that only one direction for accessing related objects is needed.
- If no arrow is displayed, the bidirectional case is assumed.

- Weak entities can be modeled using the construct called **qualified association**(**qualified aggregation**) in UML : this can represent both the identifying relationship and the partial key, which is placed in a box attached to the owner class.
- This is illustrated by the DEPENDENT class and it's qualified aggregation to EMPLOYEE in the above figure.
- The **partial key** Dependent_name is called discriminator in UML.

9. Relationship types of degree higher than two

Degree of a relationship type

- Number of participating entity types

Binary

- Relationship type of degree two

Ternary

- Relationship type of degree three

9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

- The ER diagram notation for a ternary relationship type is shown in below figure (a) , which displays the schema for the SUPPLY relationship type.

- The relationship set of SUPPLY is a set of relationship instances (s,j,p), where s is SUPPLIER who is currently supplying PART p to a PROJECT j.

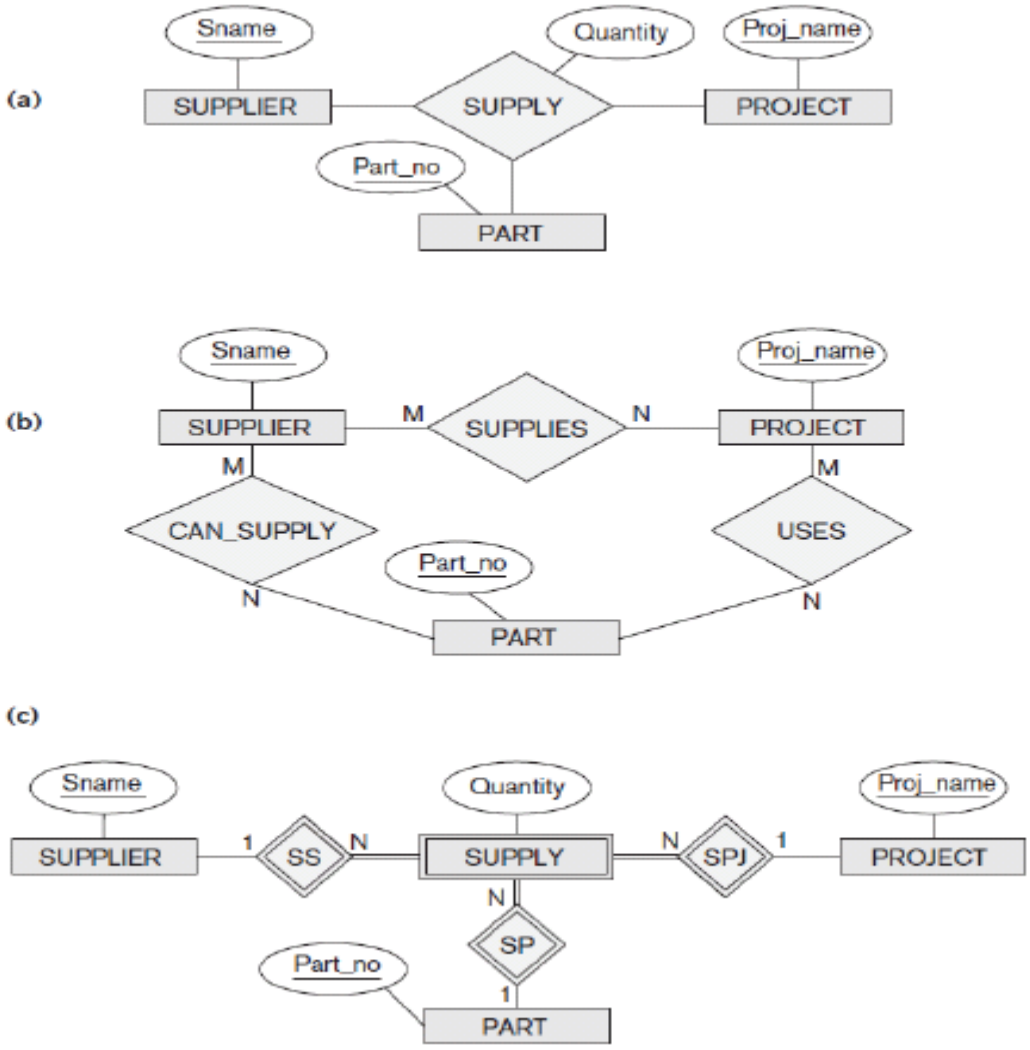


Figure: ternary relationship types(a), the SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

- In figure (b) shows an ER diagram for the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES.
- Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s,p) whenever supplier s can supply part p (to any project).
- USES, and PROJECT and PART, includes an instance (j,p) whenever project j uses part p;

- SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s,j) whenever supplier s supplies some part to project j.

Some database design tools permit only binary relationships

- A ternary relationship such as SUPPLY must be represented as weak entity type, with no partial key and with three identifying relationships.

- The three participating entity types SUPPLIER,PART, and PROJECT are together the owner entity types(see figure (c))

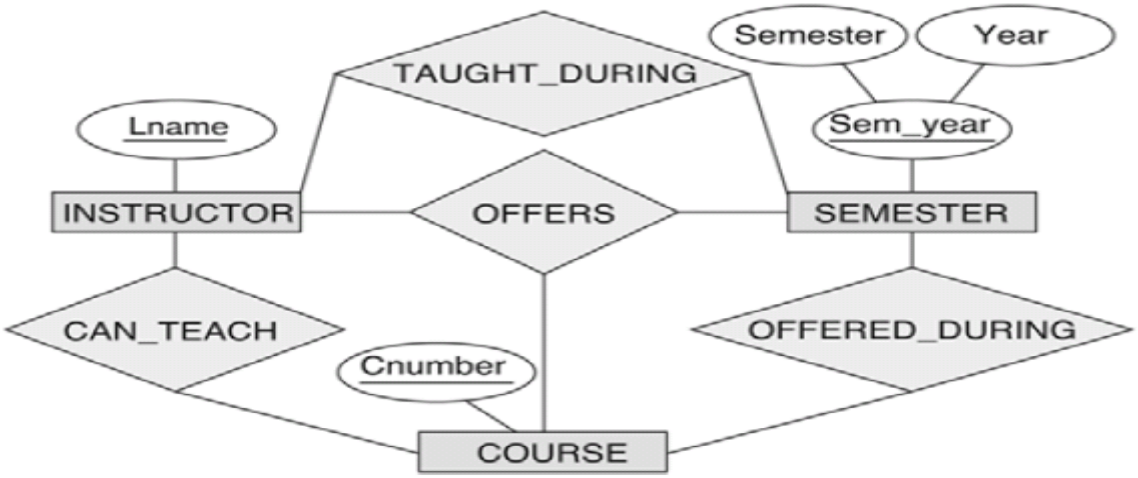


Figure: Another example of ternary versus binary relationship types

- The ternary relationship type OFFERS in the above figure represents information on instructors offering courses during particular semesters, hence it includes a relationship instance (i,s,c) whenever INSTRUCTOR i offers COURSE c during SEMESTER s.

The three binary relationship types shown in above figure have the following meanings:

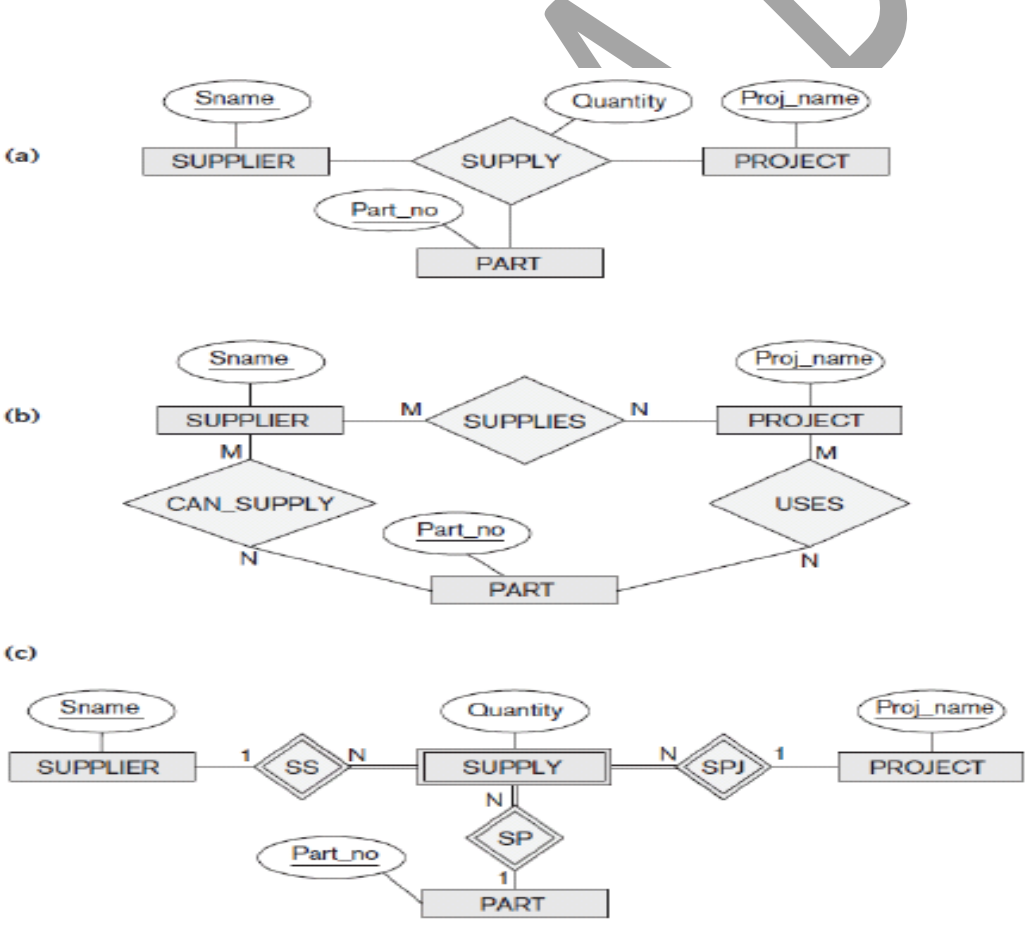
- CAN_TEACH relates a course to the instructors who can teach that course, TAUGHT_DURING relates a semester to the instructors who taught some courses offered during that semester, and OFFERED_DURING relates a semester to the courses offered during that semester by any instructor.

9.2 Constraints on Ternary (or Higher-Degree) Relationships

Notations for specifying structural constraints on *n*-ary relationships

Cardinality ratio notation for binary relationships

- Here ,a 1,M, or N is specified on each participation arc(both M and N symbols stand for many or any number).
- We place 1 on the SUPPLIER participation, and M,N on the PROJECT,PART participations in below figure.
- This specifies the constraint that a particular (j,p) combination can appear at most once in the relationship set because each such (PROJECT,PART) combination uniquely determines a single supplier.
- Hence, any relationship instance (s,j,p) is uniquely identified in the relationship set by its (j,p) combination, which makes (j,p) a key for the relationship set.



(min,max) notation for binary relationships

- A (min,max) on a participation here specifies that each entity is related to at least min and at most max relationship instances in the relationship set.
- These constraints have no bearing on determining the key of an n-ary relationship, where n>2, but specify a different type of constraint that places restrictions on how many relationship instances each entity can participate in.

2.7 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 8.1shows 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

attributes) of the subclass.

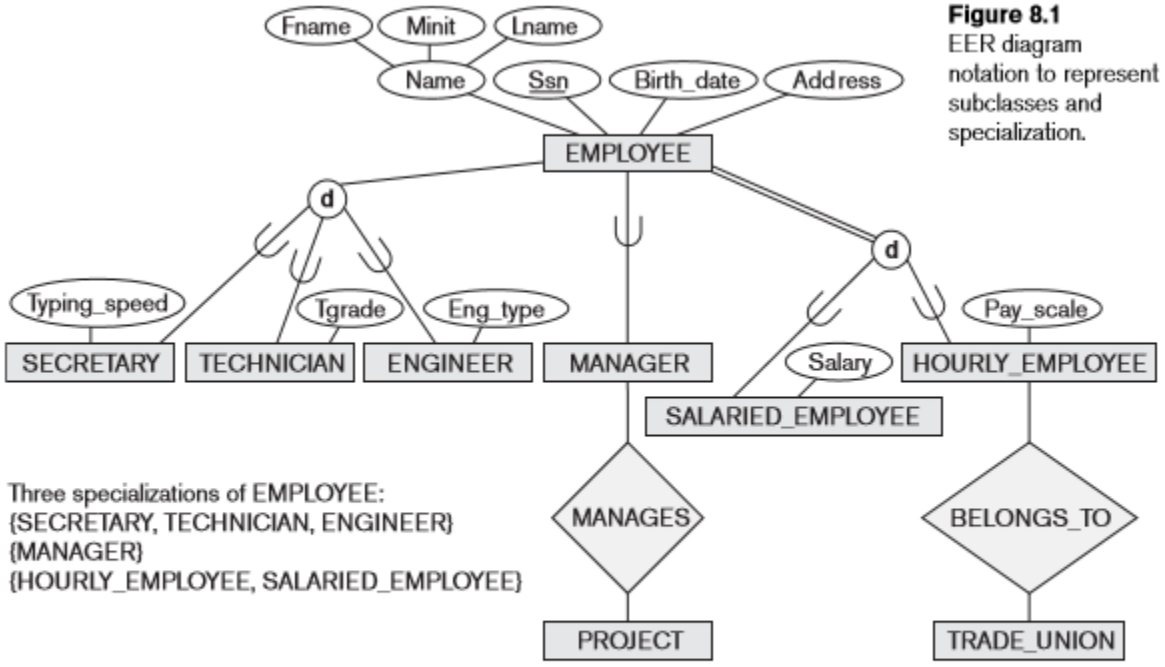


Figure 8.1
EER diagram notation to represent subclasses and specialization.

[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.

2.8 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 superclassof which the original entity types are special subclasses.

For example,consider the entity types CAR and TRUCKshown 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 andTRUCKare 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

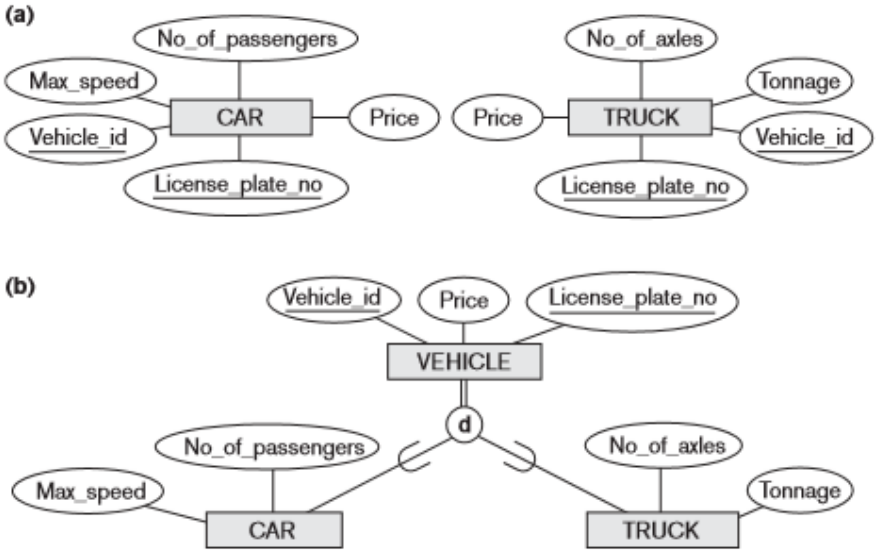


Figure 8.3
Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

Module 2

- The name SQL stands for ***Structured Query Language***.
- The SQL language may be considered as one of the major reasons for the success of relational databases in the commercial world.
- SQL is a comprehensive database language because

➢ It has statements for data definition ,database construction and database manipulation
➢ It does automatic query optimizations
➢ It has facilities for defining views on the database
➢ It has facilities for specifying security and authorization
➢ It has facilities for defining integrity constraints
➢ It has facilities for specifying transaction controls
➢ It also has rules for embedding SQL statements into a general-purpose programming language such as Java or COBOL or C/C

4.1 SQL DATA DEFINITION AND DATA TYPES:

- SQL uses the terms **table**, **row,** and **column** for the formal relational model terms **relation**, **tuple**, and **attribute**, respectively.
- The SQL command for **data definition** is the **CREATE** statement, which can be used to create **schemas**, **tables** (relations), and **domains** as well as other constructs such as views, assertions, and triggers.

4.1.1 Schema and Catalog Concepts in SQL:

- The database schema concept can be used to ***group together tables and other constructs that belong to the same database application.***
- An SQL schema is identified by a ***schema name***, ***and includes an authorization identifier to indicate the user or account who owns the schema,*** as well as descriptors for each element in the schema.
- Schema elements include ***tables, constraints, views, domains***, and other constructs (such as ***authorization grants***) that describe the schema.
- A schema is created via the **CREATE SCHEMA** statement, which can include all the schema elements' definitions.
- The schema can be assigned a **name** and **authorization identifier**, and the elements can be defined later.
- For example, the following statement creates a schema called **COMPANY**, owned by the user with authorization identifier **SMITH**:

  CREATE SCHEMA COMPANY **AUTHORIZATION** SMITH;

- In general, ***not all users are authorized to create schemas and schema elements***. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

- SQL2 uses the concept of a **catalog** - ***a named collection of schemas in an SQL environment.***

- A catalog always contains a special schema called **INFORMATION_SCHEMA**, which provides information on ***all the schemas*** in the catalog and all the element descriptors in these schemas.

- Integrity constraints such as **referential integrity** ***can be defined between relations only if they exist in schemas within the same catalog***.

4.1.2 The CREATE TABLE Command in SQL:

- The **CREATE TABLE** command is used to specify a ***new table*** by giving it a ***name*** and specifying its ***attributes*** and ***initial constraints***.

- The attributes are specified first, and each attribute is given a ***name***, ***a data type*** to specify its domain of values, and any ***attribute constraints,*** such as ***NOT NULL***.

- The **key**, **entity integrity**, and **referential integrity** constraints can be specified within the **CREATE TABLE** statement after the attributes are declared, or they can be added later using the **ALTER TABLE** command.

- We can ***explicitly attach the schema name to the relation name***, separated by a period.

  CREATE TABLE COMPANY. EMPLOYEE...
  rather than
  CREATE TABLE EMPLOYEE …

- The relations declared through CREATE TABLE statements are called **"base tables"** or **base relations**; this means that the relation and its rows are actually created and stored as a file by the DBMS.

- Base relations are distinguished from **"virtual relations"**, created through the **CREATE VIEW** statement, which may or may not correspond to an actual physical file.

- In SQL the attributes in a base table are considered to be ordered in the sequence in which they are specified in the **CREATE TABLE** statement. However, rows are not considered to be ordered within a table.
- Figure 8.1 shows sample data definition statements in SQL for the COMPANY database.

```
CREATE TABLE EMPLOYEE
      ( FNAME            VARCHAR(15)        NOT NULL,
        MINIT            CHAR,
        LNAME            VARCHAR(15)        NOT NULL,
        SSN              CHAR(9)            NOT NULL,
        BDATE            DATE,
        ADDRESS          VARCHAR(30),
        SEX              CHAR,
        SALARY           DECIMAL(10,2),
        SUPERSSN         CHAR(9),
        DNO              INT                NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE DEPARTMENT
      ( DNAME            VARCHAR(15)        NOT NULL,
        DNUMBER          INT                NOT NULL,
        MGRSSN           CHAR(9)            NOT NULL,
        MGRSTARTDATE     DATE,
      PRIMARY KEY (DNUMBER),
      UNIQUE (DNAME),
      FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) );
CREATE TABLE DEPT_LOCATIONS
      ( DNUMBER          INT                NOT NULL,
        DLOCATION        VARCHAR(15)        NOT NULL,
      PRIMARY KEY (DNUMBER, DLOCATION),
      FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE PROJECT
      ( PNAME            VARCHAR(15)        NOT NULL,
        PNUMBER          INT                NOT NULL,
        PLOCATION        VARCHAR(15),
        DNUM             INT                NOT NULL,
      PRIMARY KEY (PNUMBER),
      UNIQUE (PNAME),
      FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE WORKS_ON
      ( ESSN             CHAR(9)            NOT NULL,
        PNO              INT                NOT NULL,
        HOURS            DECIMAL(3,1)       NOT NULL,
      PRIMARY KEY (ESSN, PNO),
      FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN),
      FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) );
CREATE TABLE DEPENDENT
      ( ESSN             CHAR(9)            NOT NULL,
        DEPENDENT_NAME   VARCHAR(15)        NOT NULL,
        SEX              CHAR,
        BDATE            DATE,
        RELATIONSHIP     VARCHAR(8),
      PRIMARY KEY (ESSN, DEPENDENT_NAME),
      FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) );
```

FIGURE 8.1 SQL CREATE TABLE data definition statements for defining the COMPANY schema

4.1.3 Attribute Data Types and Domains in SQL:

The basic data types available for attributes include ***numeric, character string, bit string, boolean, date,*** and ***time***.

a) **Numeric** data types include:

- ***integer numbers*** of various sizes (INTEGER or INT, and SMALLINT).
- ***floating-point (real) numbers*** of various precision (FLOAT or REALand DOUBLE PRECISION).
- ***Formatted numbers*** which can be declared by using *DECIMAL(i,j)*or *DEC(i,j)* or NUMERIC(i,j)-where *i*, the ***precision,*** is the total number of decimal digits and *j*, the ***scale***, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

b) **Character-string** data types are either:

- ***fixed length***--CHAR(n) or CHARACTER(n), where *n* is the number of characters.
- ***varying length***-VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters.
- When specifying a literal **string value**, ***it is placed between single quotation marks (apostrophes)***, and it is ***case sensitive*** (a distinction is made between uppercase and lowercase.
- For **fixed-length strings**, a ***shorter string is padded with blank characters to the right***.
  For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed.
- ***Padded blanks are generally ignored when strings are compared***. For comparison purposes, strings are considered ordered in alphabetic order;
  If a string **str1** appears before another string ***str2*** in alphabetic order, then *str*1 is considered to be ***less than*** *str*2.
- There is also a **concatenation operator** denoted by || (doublevertical bar) that can concatenate two strings in SQL.
  For example, 'abc' || 'XYZ' results in a single string '**abcXYZ**'.

c) **Bit-string** data types are either of ***fixed length*** -BIT(n) or ***varying length***-BIT VARYING(n), where 'n' is the maximum number of bits.

- The ***default for 'n', the length of a character string or bit string, is 1***.

- ***Literal bit strings*** are placed between ***single quotes*** but preceded by a B to distinguish them from character strings;
  For example, **B'10101**

d) A **Boolean** data type has the traditional values of **TRUE** or **FALSE** in SQL.

- Because of the presence of **NULL** values, a ***three-valued logic*** is used, so a third possible value for a boolean data type is ***UNKNOWN***.

e) New data types for **date** and **time** were added in SQL2.

- The **DATE** data type has ***ten positions***, and its components are YEAR, MONTH, and DAY in the form **YYYY-MM-DD**.

- The **TIME** data type has ***at least eight positions***, with the components HOUR, MINUTE,and SECOND in the form **HH:MM:SS**.

- The **< (less than) comparison** can be used with **dates or times**-an *earlier* date is considered to be smaller than a later date, and similarly with time.

- Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME;
  For example, **DATE '2002-09-27'** or **TIME '09: 12:47'**.

f) A **timestamp** data type (TIMESTAMP) ***includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds.***

- **Literal values** are represented by **single-quoted strings** preceded by the keyword TIMESTAMP, with a blank space between data and time;
  For example, **TIMESTAMP'2002-09-27 09:12:47 648302'**.

g) Another data type related to DATE, TIME, and TIMESTAMP is the **INTERVAL** data type.

- This specifies an interval-a "***relative value"*** that can be used to increment or decrement an absolute value of a date, time, or timestamp.

- Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

Domain

- It is possible to specify the data type of each attribute directly, as in Figure 8.1;
- ***A domain can be declared, and the domain name can be used with the attribute specification***.
  For example, we can create a domain **SSN_TYPE** by the following statement:

  CREATE DOMAIN SSN_TYPE **AS** CHAR(9);

We can use SSN_TYPE in place of CHAR(9) in Figure 8.1 for the attributes SSN and SUPERSSN of EMPLOYEE, MGRSSN of DEPARTMENT, ESSN of WORKS_ON, and ESSN of DEPENDENT.

4.2 SPECIFYING CONSTRAINTS IN SQL:

Basic constraints can be specified in SQL as part of table creation. These include:

- ***key and referential integrity constraints***
- ***restrictions on attribute domains and NULLs***
- ***constraints on individual tuples(rows) within a relation***.

4.2.1 Specifying Attribute Constraints and Attribute Defaults:

- Because SQL allows **NULLs as attribute values**, a *constraint **NOT NULL*** may be specified if NULL is not permitted for a particular attribute.
- NOT NULL is always ***implicitly specified for the attributes that are part of the primary key*** of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 8.1.
- It is also possible to define a ***default value*** for an attribute by appending the clause **DEFAULT <value>** to an attribute definition.
- ***The default value is included in any new tuple if an explicit value is not provided for that attribute***. **Figure 8.2** illustrates examples of specifying a default values to various attributes.
- ***If no default clause is specified, the default value is NULL for attributes that do not have the NOT NULL constraint***.

```
CREATE TABLE EMPLOYEE
       ( ...,
         DNO              INT    NOT NULL    DEFAULT 1,
        CONSTRAINT EMPPK
         PRIMARY KEY (SSN),
        CONSTRAINT EMPSUPERFK
         FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
                        ON DELETE SET NULL     ON UPDATE CASCADE,
        CONSTRAINT EMPDEPTFK
         FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
                        ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPARTMENT
       ( ...,
         MGRSSN    CHAR(9)  NOT NULL DEFAULT '888665555',
         ...,
         CONSTRAINT DEPTPK
          PRIMARY KEY (DNUMBER),
         CONSTRAINT DEPTSK
          UNIQUE (DNAME),
         CONSTRAINT DEPTMGRFK
          FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
                ON DELETE SET DEFAULT     ON UPDATE CASCADE );

CREATE TABLE DEPT_LOCATIONS
       ( ...,
         PRIMARY KEY (DNUMBER, DLOCATION),
         FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
          ON DELETE CASCADE     ON UPDATE CASCADE );
```

FIGURE 8.2 Example illustrating how default attribute values and referential triggered actions are specified in SQL

- Another type of constraint can restrict attribute or domain values using the **CHECK clause** following an attribute or domain definition.

  For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of DNUMBER in the DEPARTMENT table (see Figure 8.1) to the following:

  DNUMBER **INT NOT NULL CHECK** (DNUMBER > 0 **AND** DNUMBER < 21);

- The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement.

  For example, we can write the following statement:

  CREATE DOMAIN D_NUM AS INTEGER CHECK (D_NUM > 0 AND D_NUM < 21);

  We can then use the created domain **D_NUM** as the attribute type for all attributes that refer to department numbers in Figure 8.1, such as DNUMBER of DEPARTMENT, DNUM of PROJECT, DNO ofEMPLOYEE, and so on.

4.2.2 Specifying Key and Referential Integrity Constraints:

- The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation.

- ***If a primary key has a single attribute, the clause can follow the attribute directly***. For example, the primary key of DEPARTMENT can be specified as follows

  DNUMBER **INT PRIMARY KEY;**

- The **UNIQUE** clause ***specifies alternate (secondary) keys***, as illustrated in the DEPARTMENTand PRO] ECT table declarations in Figure 8.1.

- **Referential integrity** is specified via the **FOREIGN KEY** clause

- A referential integrity constraint is violated when rows are inserted or deleted, or when a foreign key or primary key attribute value is modified.

- *The default action that SQL takes for an integrity violation is to reject the update operation that will cause a* violation.

- However, the schema designer can specify an ***alternative action*** to be taken if a referential integrity constraint is violated, by attaching a **referential triggered action** clause to any foreign key constraint.

- The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE as shown in figure 8.2.

- We illustrate this with the examples shown in Figure 8.2. Here, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE (**Figure 8.3)**
  This means that if the row for a supervising employee is *deleted*, the value of SUPERSSN is automatically set to NULL for all employee rows that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to SUPERSSN for all employee tuples referencing the updated employee tuple.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

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
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Figure 8.3: One possible database state for the COMPANY database

4.2.3 Giving Names to Constraints:

- Figure 8.2 also illustrates how a constraint may be given a constraint name, following the keyword **CONSTRAINT**.
- The names of all constraints within a particular schema must be unique.

- A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

4.2.4 Specifying Constraints on Tuples Using CHECK:

- In addition to key and referential integrity constraints, which are specified by special keywords, other ***table constraints*** can be specified through additional **CHECK** clauses at the end of a CREATE TABLE statement.

- These can be called **tuple-based constraints** ***because they apply to each tuple individually and are checked whenever a tuple is inserted or modified***.

  For example, suppose that the DEPARTMENT table in Figure 8.1 had an additional attribute DEPT_CREATE_DATE, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is greater than the department creation date:

  CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);

4.3 SCHEMA CHANGE STATEMENTS IN SQL:

Schema Change commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

4.3.1 The DROP Command:

- The **DROP** command can be used to **drop named schema elements**, such as **tables, domains, or constraints**.

- ***It is also possible to drop a schema***. For example, if a whole schema is not needed any more, the **DROP SCHEMA** command can be used.

- **There are two drop behavior options**: **CASCADE** and **RESTRICT**.

- For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the **CASCADE** option is used as follows:

- **DROP SCHEMA** COMPANY **CASCADE;**

 If the **RESTRICT** option is chosen in place of CASCADE, ***the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed***.

 If a base table within a schema is not needed any longer, the relation and its definition can be deleted by using the **DROP TABLE** command.

For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 8.1, we can get rid of the DEPENDENT relation by issuing the following command:
DROP TABLE DEPENDENT **CASCADE;**

 If the **RESTRICT** option is chosen instead of CASCADE, a ***table is dropped only if it is not referenced in any constraints*** (for example, by foreign key definitions in another relation) ***or views***.

 With the **CASCADE** option, all such ***constraints and views that reference the table are dropped automatically from the schema, along with the table itself***.

 The DROP command can also be used to drop other types of named schema elements, such as ***constraints or domains***.

4.3.2 The ALTER Command:

 The definition of a base table or of other named schema elements can be changed by using the ALTER command.

 For base tables, the possible ***alter table actions*** include :

- Adding or dropping a column (attribute)
- Changing a column definition
- Adding or dropping table constraints.

 For example, to **add an attribute for keeping track of jobs of employees** to the **EMPLOYEE** base relations in the COMPANY schema, we can use the command:

ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN** job **VARCHAR(12)**;

 We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by **Specifying a default clause** or by using the **UPDATE** command

- ***If no default clause is specified, the new attribute will have NULLs*** in all the tuples of the relation immediately after the command is executed;

- To **drop a column**, we must choose either **CASCADE** or **RESTRICT** for drop behavior.
- If **CASCADE** is chosen, ***all constraints and views that reference the column are dropped automatically from the schema, along with the column***.

- If **RESTRICT** is chosen, ***the command is successful only if no views or constraints (or other elements) reference the column***.

- The following command removes the attribute **ADDRESS** from the **EMPLOYEE** base table:
  ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE;**

- It is also possible to alter a column definition by **dropping an existing default clause or by defining a new default clause**.
  The following examples illustrate this clause:

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** MGRSSN **DROP DEFAULT;**

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** MGRSSN **SET DEFAULT** "333445555";

- ***It is possible to change the constraints specified on a table by adding or dropping a constraint***.

- To be dropped, ***a constraint must have been given a name*** when it was specified.
  For example, to drop the constraint named **EMPSUPERFK** in Figure 8.2 from the EMPLOYEE relation, we write:

- **ALTER TABLE** COMPANY.EMPLOYEE **DROP CONSTRAINT** EMPSUPERFK **CASCADE;**

- ***We can redefine a replacement constraint by adding a new constraint to the relation***, if needed. This is specified by using the **ADD** keyword in the ALTER TABLE statement followed by the new constraint.

4.4 BASIC QUERIES IN SQL:

SQL has one basic statement for ***retrieving information from a database***: the **SELECT** statement.

4.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries:

- **The basic form of the SELECT statement**, sometimes called a **mapping** or a **select- from-where block**, is formed of the three clauses **SELECT**, **FROM,** and **WHERE** and has the following form:

  SELECT <attribute list>
  FROM <table list>
  WHERE <condition>**;**
  Where

- **<attribute list>** is a list of attribute names whose values are to be retrieved by the query.
- **<table list>** is a list of the relation names required to process the query.
- **<condition>** is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

- In SQL, the basic ***logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>.***
- These correspond to the relational algebra operators =, <, ~, >, ~, and *, respectively, and to the c{c++ programming language operators =, <, <=, >, >=, and !=.
- **Examples:**

QUERY 0

Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'.

```
Q0: SELECT BDATE, ADDRESS
    FROM   EMPLOYEE
    WHERE  FNAME='John' AND MINIT='B' AND LNAME='Smith';
```

This query involves only the **EMPLOYEE** relation listed in the FROM clause. The query *selects* the EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the BDATE and ADDRESS attributes listed in the SELECT clause. **Figure 8.3a** shows the result of query QO.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT FNAME, LNAME, ADDRESS
    FROM   EMPLOYEE, DEPARTMENT
    WHERE  DNAME='Research' AND DNUMBER=DNO;
```

The result of query Ql is shown in **Figure 8.3b**

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
Q2: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM    PROJECT, DEPARTMENT, EMPLOYEE
    WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND
            PLOCATION='Stafford';
```

- ✓ The condition DNUM = DNUMBER relates a project to its controlling department.
- ✓ The condition MGRSSN = SSN relates the controlling department to the employee who manages that department.
- ✓ The condition PLOCATION='Stafford' selects the specified project location.

The result of query Q2 is shown in **Figure 8.3c**.

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

(b)

FNAME	LNAME	ADDRESS
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

PNUMBER	DNUM	LNAME	ADDRESS	BDATE
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(d)

E.FNAME	E.LNAME	S.FNAME	S.LNAME
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jenniter	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

SSN
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

SSN	DNAME
123456789	Reeearch
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(g)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

FIGURE 8.3 Results of SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C

4.4.2 Ambiguous Attribute Names, Aliasing, and Tuple Variables:

- In SQL the ***same name can be used for two (or more) attributes as long as the attributes are in different relations.***

- If this is the case, and ***if a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity***.

- This is done by *prefixing* ***the relation name to the attribute name and separating the two by a period***.
  To illustrate this, suppose that the **DNO** and **LNAME** attributes of the EMPLOYEE relation were called **DNUMBER** and **NAME**, and the **DNAME** attribute of **DEPARTMENT** was also called **NAME**; then, to prevent ambiguity. Query Ql would be rephrased as shown in QIA.

```
Q1A: SELECT  FNAME, EMPLOYEE.NAME, ADDRESS
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DEPARTMENT.NAME='Research' AND
             DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

- ***Ambiguity also arises in the case of queries that refer to the same relation twice***, as in the following example:

QUERY 8

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:  SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
     FROM    EMPLOYEE AS E, EMPLOYEE AS S
     WHERE   E.SUPERSSN=S.SSN;
```

- In this case, we are allowed to declare alternative relation names **E** and **S**, called **"aliases"** or **"tuple variables"**, for the EMPLOYEE relation.

- An alias can follow the keyword **AS**, as shown in *Q8*, or ***it can directly follow the relation name***-for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of *Q8.*

- ***It is also possible to rename the relation attributes within the query in SQL by giving them aliases***.
  For example, if we write **EMPLOYEE AS E(FN, MI, LN, SSN, SD, ADDR, SEX, SAL, SSSN, DNO)** in the FROM clause, FN becomes an alias for FNAME, MI for MINH, LN for LNAME, and so on.

- In *Q8,* we can think of **E** and S as two ***different copies*** of the **EMPLOYEE** relation;

➢ The first, **E** represents employees in the role of supervisees

- The second, **S,** represents employees in the role of supervisors. The result of query Q8 is shown in **Figure 8.3d**.

- Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query.

- We could specify query **Q1A** as in **Q1B**:

```
Q1B: SELECT  E.FNAME, E.NAME, E.ADDRESS
     FROM    EMPLOYEE E, DEPARTMENT D
     WHERE   D.NAME='Research' AND D.DNUMBER=E.DNUMBER;
```

4.4.3 Unspecified WHERE Clause and Use of the Asterisk:

- A **missing WHERE clause** indicates **no condition** on tuple selection; hence, ***all tuples of the relation specified in the FROM clause qualify and are selected for the query result***.

- ***If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT-all possible tuple combinations-of these relations is selected***.
  For example, **Query 9** selects all EMPLOYEE SSNS (**Figure 8.3e**), and **Query 10** selects all combinations of an EMPLOYEE SSN and a DEPARTMENT DNAME (**Figure 8.3f**).

QUERIES 9 AND 10

Select all EMPLOYEE SSNS (Q9), and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME (Q10) in the database.

```
Q9:  SELECT  SSN
     FROM    EMPLOYEE;
Q10: SELECT  SSN, DNAME
     FROM    EMPLOYEE, DEPARTMENT;
```

- ***It is extremely important to specify every selection and join condition in the WHERE clause***; if any such condition is overlooked then incorrect and very large relations may result.

- ***To retrieve all the attribute values of the selected tuples,*** we do not have to list the attribute names explicitly in SQL; ***we just specify an asterisk (*),*** which stands for *all the attributes.*

- Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 8.3g)

```
Q1C:  SELECT  *
      FROM    EMPLOYEE
      WHERE   DNO=5;
```

- Query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works the 'Research' department.

```
Q1D:  SELECT  *
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNAME='Research' AND DNO=DNUMBER;
```

- **Query Ql0A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations**.

```
Q10A: SELECT  *
      FROM    EMPLOYEE, DEPARTMENT;
```

4.4.4 Tables as Sets in SQL:

- ***An SQL table with a key is restricted to being a set***, since the key value must be distinct in each tuple.
- SQL usually treats a table not as a set but rather as a **multiset;**
- Duplicate tuples can appear more than once in a table, and in the result of a query.
- SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.

- The user may want to see duplicate tuples in the result of a query.

- When an aggregate function (SUM,MAX,MIN,AVG) is applied to tuples, in most cases we do not want to eliminate duplicates.

- If we do want to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result.

- In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.

- Specifying SELECT with neither ALL nor DISTINCT-as in our previous examples-is equivalent to SELECT ALL.

- **Query 11 retrieves the salary of every employee**; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in **Figure 8.4(a)**.

- **By using the keyword DISTINCT as in Q11A, we get only the distinct salary values** , as shown in **Figure 8.4(b).**

QUERY 11

Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:  SELECT ALL SALARY
      FROM   EMPLOYEE;

Q11A: SELECT DISTINCT SALARY
      FROM   EMPLOYEE;
```

(a)

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARY
30000
40000
25000
43000
38000
55000

(c)

FNAME	LNAME

(d)

FNAME	LNAME
James	Borg

FIGURE 8.4 Results of additional SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

- **SQL has directly incorporated some of the set operations of relational algebra**.

➢ There is set union (UNION) operation

➢ There is set difference (EXCEPT) operation

➢ And there is set intersection (INTERSECT) operation

- The relations resulting from these set operations are sets of tuples; that is, ***duplicate tuples are eliminated from the result.***

- Because these set operations apply only to ***union-compatible relations***, ***we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations***.

- The following example illustrates the use of **UNION**.

QUERY 4

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4:  (SELECT DISTINCT PNUMBER
      FROM   PROJECT, DEPARTMENT, EMPLOYEE
      WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND LNAME='Smith')
      UNION
      (SELECT DISTINCT PNUMBER
      FROM   PROJECT, WORKS_ON, EMPLOYEE
      WHERE  PNUMBER=PNO AND ESSN=SSN AND LNAME='Smith');
```

- ✓ The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project
- ✓ The second SELECT retrieves the projects that involve a 'Smith' as a worker on the project.
- ✓ Applying the UNION operation to the two SELECT queries gives the desired result.

- **Figure 8.5** illustrates the other multiset operations

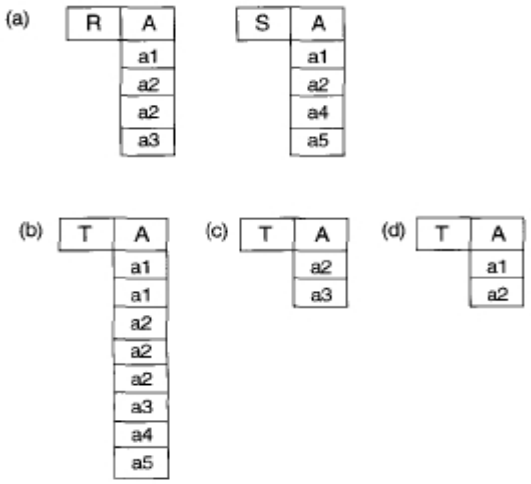


FIGURE 8.5 The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

4.4.5 Substring Pattern Matching and Arithmetic Operators:

- SQL allows **comparison conditions** on only parts of a character string, using the **LIKE** comparison operator.

- This can be used for ***string pattern matching***.

- Partial strings are specified using two reserved characters:
- ➢ **%** replaces an arbitrary number of zero or more characters and
- ➢ The underscore(_)replaces a single character.

QUERY 12

Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ADDRESS LIKE '%Houston,TX%';
```

- **To retrieve all employees who were born during the 1950s, we can use Query 12A**. Here, '5' must be the third character of the string (according to our format for date), so we use the value ' 5 ', with each underscore serving as a placeholder for an arbitrary character.

QUERY 12A

Find all employees who were born during the 1950s.

```
Q12A: SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   BDATE LIKE '__5_____';
```

- ***If an underscore or % is needed as a literal character in the string, the character should be preceded by an escape character, which is specified after the string using the keyword ESCAPE.***

  For example, **'AB_CD\%EF' ESCAPE** '\' represents the literal string **AB_CD%EF**', because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character.

- If an apostrophe (') is needed, it is represented as two consecutive apostrophes ('') so that it will not be interpreted as ending the string.

- ***The standard arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.***

QUERY 13

Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
Q13:  SELECT  FNAME, LNAME, 1.1*SALARY AS INCREASED_SAL
      FROM    EMPLOYEE, WORKS_ON, PROJECT
      WHERE   SSN=ESSN AND PNO=PNUMBER AND
              PNAME='ProductX';
```

- ***For string data types, the concatenate operator '||' can be used in a query to append two string values.***

- For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (-) a date, time, or timestamp by an interval.

- Another comparison operator that can be used for convenience is **BETWEEN**, which is illustrated in **Query 14**.

QUERY 14

Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT  *
      FROM    EMPLOYEE
      WHERE   (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
```

The condition (SALARY **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((SALARY >= 30000) **AND** (SALARY <= 40000)).

4.4.6 Ordering of Query Results:

- ***SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the* ORDER BY *clause.***

QUERY 15

Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

```
Q15:  SELECT    DNAME, LNAME, FNAME, PNAME
      FROM      DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
      WHERE     DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
      ORDER BY  DNAME, LNAME, FNAME;
```


- The default order is in ascending order of values.
- We can specify the keyword DESC if we want to see the result in a descending order of values.

4.5. INSERT, DELETE, AND UPDATE STATEMENTS IN SQL:

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

4.5.1 The INSERT Command:

- In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple.

- The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.

  For example, to add a new tuple to the EMPLOYEE relation, we can use U1:

```
U1: INSERT INTO  EMPLOYEE
    VALUES       ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                 Oak Forest,Katy,TX', 'M', 37000, '987654321', 4);
```

- A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command.

- This is useful if a relation has many attributes but only a few of those attributes are to be assigned values in the new tuple.

- However, the values must include all attributes with NOT NULL specification and no default value.

- Attributes with NULL allowed or DEFAULT values are the ones that can be left out.

  For example, to enter a tuple for a new EMPLOYEE for whom we know only the FNAME,
  LNAME, DNO, and SSN attributes, we can use U1A:

```
U1A: INSERT INTO  EMPLOYEE (FNAME, LNAME, DNO, SSN)
     VALUES       ('Richard', 'Marini', 4, '653298653');
```

  Attributes not specified in UlA are set to their DEFAULT or to NULL.

- It is also possible to insert into a relation multiple tuples separated by commas in a single INSERT command. The attribute values forming each tuple are enclosed in parentheses.

  INSERT INTO EMPLOYEE VALUES (('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest,Katy,TX', 'M', 7000, '987654321', 4) , ('Roy', 'J', 'Mathews', '653298654', '1968-11-23', '94 Fulkerson,NY', 'M', 9000, '987664351', 5));

- A DBMS that fully implements SQL-99 should support and enforce all the integrity constraints that can be specified in the DDL. However, some DBMSs do not incorporate all the constraints (like referential integrity), in order to maintain the efficiency of the DBMS and because of the complexity of enforcing all constraints.

- If a system does not support some constraint, the users or programmers must enforce the constraint.

- For example, if we issue the command in **U2** on the database shown in Table 5.1, a DBMS **not supporting referential integrity** will do the insertion even though no **DEPARTMENT** tuple exists in the database with **DNUMBER = 2**.

```
U2:  INSERT INTO   EMPLOYEE (FNAME, LNAME, SSN, DNO)
     VALUES        ('Robert', 'Hatcher', '980760540', 2);
(* U2 is rejected if referential integrity checking is provided by dbms *)
```

- It is the responsibility of the user to check that any such constraints *whose checks are not implemented by the* DBMS are not violated.
- A single INSERT command can be used for inserting multiple tuples into a relation in conjunction with creating the relation and loading the relation with the result of a query.

  For example, to create a temporary table DEPT_NAME that has the name, number of employees, and total salaries for each department, we can write the statements in U3A and U3B:

```
U3A: CREATE TABLE  DEPTS_INFO
     (DEPT_NAME    VARCHAR(15),
     NO_OF_EMPS    INTEGER,
     TOTAL_SAL     INTEGER);
U3B: INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                   TOTAL_SAL)
     SELECT        DNAME, COUNT (*), SUM (SALARY)
     FROM          (DEPARTMENT JOIN EMPLOYEE ON
                   DNUMBER=DNO)
     GROUP BY      DNAME;
```

- We can now query DEPTS_INFO as we would any other relation; when we do not need it any more, we can remove it by using the DROP TABLE command.

4.5.2 The DELETE Command:

- The DELETE command removes tuples from a relation.
- It includes a WHERE clause to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time.
- However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL.
- Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command.
- A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.
- The DELETE commands in U4A to U4D, if applied independently to the database of Table 5.1, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A: DELETE FROM  EMPLOYEE
     WHERE        LNAME='Brown';
U4B: DELETE FROM  EMPLOYEE
     WHERE        SSN='123456789';
U4C: DELETE FROM  EMPLOYEE
     WHERE        DNO IN (SELECT   DNUMBER
                          FROM     DEPARTMENT
                          WHERE    DNAME='Research');
U4D: DELETE FROM  EMPLOYEE;
```

4.5.3 The UPDATE Command:

- The UPDATE command is used to modify attribute values of one or more selected tuples.
- As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation.
- However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL.
- An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values.
- For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

```
U5: UPDATE  PROJECT
    SET     PLOCATION = 'Bellaire', DNUM = 5
    WHERE   PNUMBER=10;
```

- Several tuples can be modified with a single UPDATE command.

Example to give all employees in the 'Research' department a 10 percent rise in salary

```
U6: UPDATE  EMPLOYEE
    SET     SALARY = SALARY *1.1
    WHERE   DNO IN (SELECT   DNUMBER
                    FROM     DEPARTMENT
                    WHERE    DNAME='Research');
```

4.6 ADDITIONAL FEATURES OF SQL:

- SQL has the capability to specify more general constraints, called assertions, using the CREATE ASSERTION statement.

- SQL has language constructs for specifying views, also known as virtual tables, using the CREATE VIEW statement. Views are derived from the base tables declared through the CREATE TABLE statement.
- SQL has several different techniques for writing programs in various programming languages that can include SQL statements to access one or more databases. These include embedded SQL, dynamic SQL SQL/CLI (Call Language Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Program Stored Modules).
- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We call these commands a storage definition language (SDL).
- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- SQL has language constructs for specifying the granting and revoking of privileges to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement-such as SELECT, INSERT, DELETE, or UPDATE-to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands-called GRANT and REVOKE.
- SQL has language constructs for creating Triggers. These are generally referred to as active database techniques, since they specify actions that are automatically triggered by events such as database updates.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as object- relational. Capabilities such as creating complex-structured attributes (also called nested relations), specifying abstract data types (called DDTs or user-defined types) for attributes and tables, creating object identifiers for referencing tuples, and specifying operations on these types.
- SQL and relational databases can interact with new technologies such as XML (eXtended Markup Language) and OLAP (On Line Analytical Processing for Data Warehouses).

1.1 UNARY RELATIONAL OPERATIONS: SELECT and PROJECT

1.1.1 The SELECT Operation

- The **SELECT** operation is used to choose a ***subset* of the tuples** from a relation that satisfies a **selection condition**.
- We can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition.
- The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

  For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

  $\sigma_{Salary>30000}(EMPLOYEE)$

  In general, the SELECT operation is denoted by

  $\sigma_{<selection\ condition>}(R)$

  where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*. The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value> or
<attribute name> <comparison op> <attribute name>

- Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition.

  For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

  $\sigma_{(Dno=4\ \textbf{AND}\ Salary>25000)\ \textbf{OR}\ (Dno=5\ \textbf{AND}\ Salary>30000)}(EMPLOYEE)$

- The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:
  - (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise,it is FALSE.
  - (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
  - (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

- The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple.
- The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of *R*.

- The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ \ldots)) = \sigma_{<cond1>\ \textbf{AND}<cond2>\ \textbf{AND}\ldots\textbf{AND}\ <condn>}(R)$$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query. For example, the following operation:

$$\sigma_{Dno=4\ \textbf{AND}\ Salary>25000}(EMPLOYEE)$$

would correspond to the following SQL query:

```
SELECT   *
FROM     EMPLOYEE
WHERE    Dno=4 AND Salary>25000;
```

1.1.2 The PROJECT Operation

- The **PROJECT** operation, selects **certain *columns*** from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only.

- Therefore, the result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns.
  For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{Lname,\ Fname,\ Salary}(EMPLOYEE)$$

- The general form of the PROJECT operation is

$$\pi_{<attribute\ list>}(R)$$

where (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*.

- The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.
- The **PROJECT** operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. *This is known as* ***duplicate elimination.***

Note:

$$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

In SQL, the PROJECT attribute list is specified in the SELECT clause of a query. For example, the following operation:

$$\pi_{Sex, Salary}(EMPLOYEE)$$

would correspond to the following SQL query:

```
SELECT    DISTINCT Sex, Salary
FROM      EMPLOYEE
```

Consider the relational algebraic queries below:

1.1.3 Sequences of Operations and the RENAME Operation

- We must give names to the relations that hold the intermediate results.
- For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an in-line expression, as follows:

$$\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

- Figure 1.1(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname, Lname, Salary}(DEP5\_EMPS)$$

:

- It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to rename the attributes in the intermediate and result relations.

(a)

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 6.2
Results of a sequence of operations. (a) $\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$. (b) Using intermediate relations and renaming of attributes.

(b)
TEMP

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

- To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$TEMP \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$R(First\_name, Last\_name, Salary) \leftarrow \pi_{Fname, Lname, Salary}(TEMP)$$

- We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

$$\rho_{S(B1, B2, ..., Bn)}(R) \quad or \quad \rho_S(R) \quad or \quad \rho_{(B1, B2, ..., Bn)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and B1, B2, ..., Bn are the new attribute names.

- In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using AS, as in the following example:

SELECT E.Fname AS First_name, E.Lname AS Last_name, E.Salary AS Salary
FROM EMPLOYEE AS E
WHERE E.Dno=5,

1.2 Relational Algebra Operations from Set Theory

1.2.1 The UNION, INTERSECTION, and MINUS Operations

We can define the ***three set operations UNION, INTERSECTION, and SET DIFFERENCE*** on two union-compatible relations R and S as follows:

- **UNION:** The result of this operation, denoted by R ∪ S, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

  For example, to retrieve the Social Security numbers of all employees who either work in

department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{Ssn}(DEP5\_EMPS)$$
$$RESULT2(Ssn) \leftarrow \pi_{Super\_ssn}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 6.3), while eliminating any duplicates.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure 6.3
Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

- **INTERSECTION:** The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S.

- **SET DIFFERENCE (or MINUS or EXCEPT):** The result of this operation, denoted by R – S, is a relation that includes all tuples that are in R but not in S.

These are binary operations; that is, each is applied to two sets (of tuples).When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same type of tuples; this condition has been called ***union compatibility or type compatibility***.

Two relations R(A1, A2, ..., An) and S(B1, B2, ..., Bn) are said to be union compatible (or type compatible) if they have the same degree n and if dom(Ai) = dom(Bi) for i 1 to n This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \quad \text{and} \quad R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as *n*-ary operations applicable to any number of relations because both are also *associative operations;* that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative;* that is, in general,

$$R - S \neq S - R$$

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) **STUDENT**

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

1.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

- **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT or CROSS JOIN**—which is denoted by ×.
- This is also a binary set operation, but the relations on which it is applied do not have to be union compatible. This set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).
- In general, the result of R(A1, A2, ..., An) × S(B1, B2, ..., Bm) is a relation Q with degree n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm), in that order.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has m tuples and S has n tuples, then R × S will have m*n tuples.

Example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex='F'}}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	D
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

1.3 Binary Relational Operations: JOIN and DIVISION

1.3.1 The JOIN Operation

- The **JOIN** operation, denoted by , is used to combine *related tuples* from two relations into single "longer" tuples.
- To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. to get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.
  We do this by using the JOIN

$$DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$$
$$RESULT \leftarrow \pi_{Dname,\ Lname,\ Fname}(DEPT\_MGR)$$

- The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation.

- Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$$
$$ACTUAL\_DEPENDENTS \leftarrow \sigma_{Ssn=Essn}(EMP\_DEPENDENTS)$$

These two operations can be replaced with a single JOIN operation as follows:

$$ACTUAL\_DEPENDENTS \leftarrow EMPNAMES \bowtie_{Ssn=Essn} DEPENDENT$$

The general form of a JOIN operation on two relations[5] $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

- The result of the JOIN is a relation *Q* with *n* + *m* attributes *Q*(*A*1, *A*2, ..., *An*, *B*1, *B*2, ... , *Bm*)
  In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

Figure 6.6
Result of the JOIN operation DEPT_MGR ← DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE.

- A **general join condition is of the form**
  <condition> **AND** <condition> **AND...AND** <condition>
  where each <condition> is of the form *Ai* θ *Bj*, *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}.
  A JOIN operation with such a general join condition is called a **THETA JOIN**.

1.3.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

- The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both previous examples were EQUIJOINs.
- Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple.
- For example, in Figure 6.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—

denoted by * was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

- The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

- Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.
  In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

  PROJ_DEPT ← PROJECT * ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)

  The same query can be done in two steps by creating an intermediate table DEPT as follows:

  DEPT ← ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)
  PROJ_DEPT ← PROJECT * DEPT

  The attribute Dnum is called the join attribute for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations.

  For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write
  DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS

- *In general, the join condition for NATURAL JOIN is constructed by equating each pair of join attributes that have the same name in the two relations and combining these conditions with AND.*

- A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins.**

(a)
PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)
DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 6.7
Results of two NATURAL JOIN operations. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

- A more general, but nonstandard definition for NATURAL JOIN is

Q ← R *(<list1>),(<list2>)S

In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S.

The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an n-way join.

For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$

1.3.3 A Complete Set of Relational Algebra Operations

- It has been shown that the set of relational algebra operations {σ, π, ∪, ρ, –, ×} is a complete set; that is, any of the other original relational algebra operations can be expressed as a sequence of operations from this set.

For example, the **INTERSECTION** operation can be expressed by using UNION and MINUS as follows:

R ∩ S ≡ (R ∪ S) – ((R – S) ∪ (S – R))

JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{\text{<condition>}} S \equiv \sigma_{\text{<condition>}}(R \times S)$$

A **NATURAL JOIN** can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.

1.3.4 The DIVISION Operation

- The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications.

example is Retrieve the names of employees who work on ***all*** the projects that 'John Smith' works on.

To express this query using the DIVISION operation, proceed as follows.

First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$\text{SMITH} \leftarrow \sigma_{\text{Fname='John' AND Lname='Smith'}}(\text{EMPLOYEE})$$
$$\text{SMITH\_PNOS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} \bowtie_{\text{Essn=Ssn}} \text{SMITH})$$

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$\text{SSN\_PNOS} \leftarrow \pi_{\text{Essn, Pno}}(\text{WORKS\_ON})$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$\text{SSNS(Ssn)} \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname}}(\text{SSNS} * \text{EMPLOYEE})$$

Figure 6.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

- In general, the DIVISION operation is applied to two relations R(Z) ÷ S(X), where the attributes of R are a subset of the attributes of S; that is, X ⊆ Z. Let Y be the set of attributes of R that are not attributes of S;

- The **DIVISION operation** is defined for convenience for dealing with queries that involve ***universal quantification*** or the ***all*** condition

- The DIVISION operation can be expressed as a sequence of π, ×, and – operations as follows:

$$T1 \leftarrow \pi_Y(R)$$
$$T2 \leftarrow \pi_Y((S \times T1) - R)$$
$$T \leftarrow T1 - T2$$

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, OR $R_1 \bowtie_{(<\text{join attributes 1}>),\ (<\text{join attributes 2}>)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{<\text{join condition}>} R_2$, OR $R_1 *_{(<\text{join attributes 1}>),\ (<\text{join attributes 2}>)} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

1.3.5 Notation for Query Trees

- Here we describe about the notation typically used in relational systems to represent queries internally. The notation is called a **query tree** or sometimes it is known as a **query evaluation tree** or **query execution tree**.
- It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.
- A **query tree** is a tree data structure that corresponds to a relational algebra expression.
- It represents the *input relations of the query as leaf nodes of the tree*, and represents the *relational algebra operations as internal nodes*.

- An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation.

- The execution terminates when the root node is executed and produces the result relation for the query.

$$\pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$$

Query tree for the abobe query (Q2) . In this, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE.

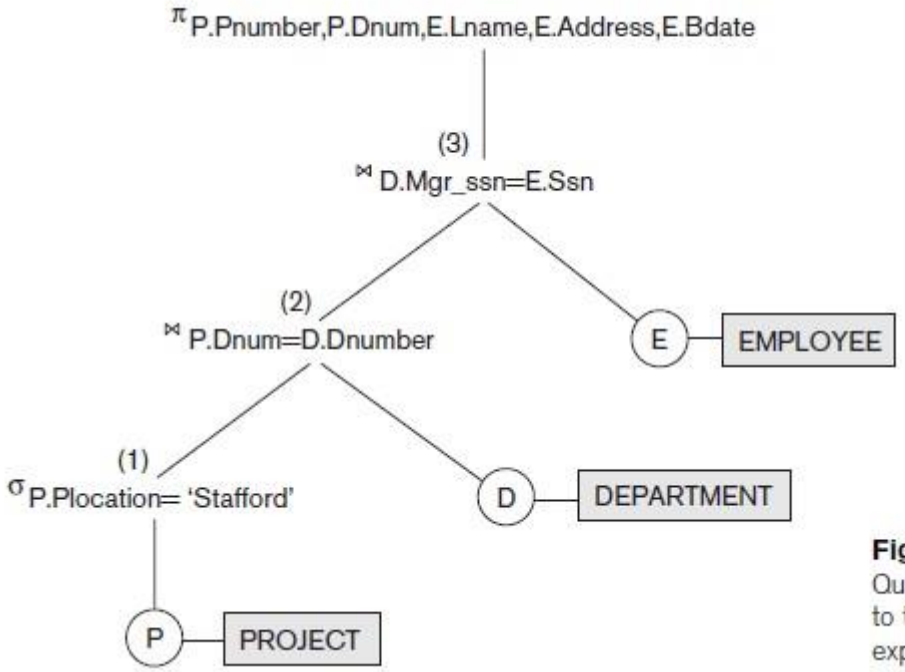


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

- In order to execute query Q2, the node marked (1) in Figure 6.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on.

- In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra.

1.4 Additional Relational Operations

1.4.1 Generalized Projection

- The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list.
- The generalized form can be expressed as:

$\pi_{F1, F2, ..., Fn}(R)$

where F1, F2, ..., Fn are functions over the attributes in relation R and may involve arithmetic operations and constant values.

Example:

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show **Net Salary = Salary – Deduction, Bonus = 2000 * Years_service, and Tax = 0.25 * Salary**.

Then a generalized projection combined with renaming may be used as follows:

$$\text{REPORT} \leftarrow \rho_{(\text{Ssn, Net\_salary, Bonus, Tax})}(\pi_{\text{Ssn, Salary – Deduction, 2000 * Years\_service, 0.25 * Salary}}(\text{EMPLOYEE})).$$

1.4.2 Aggregate Functions and Grouping

- Mathematical aggregate functions are applied on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples.
- Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.
- We can define an AGGREGATE FUNCTION operation, using the symbol $\Im$ (pronounced script F), to specify these types of requests as follows:

  <grouping attributes> $\Im$ **<function list> (R)**

  where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM,COUNT—and <attribute> is an attribute of the relation specified by R.
- The resulting relation has the grouping attributes plus one attribute for each element in the function list**.**
- **For example**, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

  $$\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(\text{Dno } \Im \text{ COUNT Ssn, AVERAGE Salary } (\text{EMPLOYEE}))$$

  In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R.

  If we do not want to rename the attributes then the above query we can write it as,

  $$\text{Dno } \Im \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$$

  Note: If no grouping attributes are specified, the functions are applied to all the tuples in the relation, so the resulting relation has a single tuple only.

For ex. $\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$

Example queries and their results:

The aggregate function operation.

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}({}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$.

b. ${}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

c. $\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

1.4.3 Recursive Closure Operations

- **Recursive closure** operation in relational algebra is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 3.5 and 3.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor).
- An example of a recursive operation is to retrieve all supervisees of an employee *e* at all levels—that is, all employees *e*' directly supervised by *e*, all employees *e*'' directly supervised by each employee *e*', all employees *e*''' directly supervised by each employee *e*'', and so on.
- For example, to specify the Ssns of all employees *e*_ directly supervised—*at level one*—by the employee *e* whose name is 'James Borg' (see Figure 3.6), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND Lname='Borg'}}(\text{EMPLOYEE}))$$
$$\text{SUPERVISION}(\text{Ssn1, Ssn2}) \leftarrow \pi_{\text{Ssn,Super\_ssn}}(\text{EMPLOYEE})$$
$$\text{RESULT1}(\text{Ssn}) \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{BORG\_SSN})$$

- To retrieve all employees supervised by Borg at level 2—that is, all employees *e*'' supervised by some employee *e*' who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

$$\text{RESULT2}(\text{Ssn}) \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{RESULT1})$$

- To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

  RESULT ← RESULT2 ∪ RESULT1

  Example result:

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn)	(Super_ssn)
Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 ∪ RESULT2)

- Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels"without utilizing a looping mechanism unless we know the maximum number of levels.
- An operation called the ***transitive closure*** of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

1.4.4 OUTER JOIN Operations

- For a NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a matching (or related) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated.
- This type of join, where tuples with no match are eliminated, is known as an inner join. The join operations we described earlier in Section 1.3 are all inner joins.
- This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

- A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values.
- **For example**, suppose that we want a list of all employee names as well as the name of the departments they manage if they happen to manage a department; if they do not manage one, we can indicate it with a NULL value.
- We can apply an operation LEFT OUTER JOIN, denoted by ⟕, to retrieve the result as follows:

$$TEMP \leftarrow (EMPLOYEE \text{ ⟕}_{Ssn=Mgr\_ssn} DEPARTMENT)$$

$$RESULT \leftarrow \pi_{Fname,\ Minit,\ Lname,\ Dname}(TEMP)$$

- The **LEFT OUTER JOIN** operation keeps every tuple in the first, or left, relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled with NULL values.

Figure 6.12
The result of a LEFT OUTER JOIN operation.

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

- A similar operation, **RIGHT OUTER JOIN**, denoted by ⟖ , keeps every tuple in the second, or right, relation S in the result of R ⟖ S.
- A third operation, **FULL OUTER JOIN**, denoted by , keeps all tuples in both the left and the right relations when no matching tuples are found, filling them with NULL values as needed.

1.4.5 The OUTER UNION Operation

- The OUTER UNION operation was developed to take the union of tuples from two relations that have some common attributes, but are not union (type) compatible.

- This operation will take the UNION of tuples in two relations R(X, Y) and S(X, Z) that are partially compatible, meaning that only some of their attributes, say X, are union compatible.
- The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation T(X, Y, Z). It is therefore the same as a FULL OUTER JOIN on the common attributes.
- Two tuples t1 in R and t2 in S are said to match if t1[X]=t2[X]. These will be combined (unioned) into a single tuple in t. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.
- For example, an OUTER UNION can be applied to two relations whose schemas are

  STUDENT(Name, Ssn, Department, Advisor)

  and INSTRUCTOR(Name, Ssn, Department, Rank).
- Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department.
- The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

  STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)
- All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result.
- Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute.
- A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.

1.5 Examples of Queries in Relational Algebra

<u>Query 1</u> Retrieve the name and address of all employees who work for the 'Research' department.

$$RESEARCH\_DEPT \leftarrow \sigma_{Dname='Research'}(DEPARTMENT)$$
$$RESEARCH\_EMPS \leftarrow (RESEARCH\_DEPT \bowtie_{Dnumber=Dno} EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname,\ Lname,\ Address}(RESEARCH\_EMPS)$$

In a single line we can write the above query as,

$$\pi_{Fname,\ Lname,\ Address}\ (\sigma_{Dname='Research'}(DEPARTMENT \bowtie_{Dnumber=Dno}(EMPLOYEE)))$$

<u>Query 2</u>. Retrieve the number of employees and the average salary of all the employees.

$$\Im_{COUNT\ Ssn,\ AVERAGE\ Salary}(EMPLOYEE)$$

Query 3. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$$STAFFORD\_PROJS \leftarrow \sigma_{Plocation='Stafford'}(PROJECT)$$
$$CONTR\_DEPTS \leftarrow (STAFFORD\_PROJS \bowtie_{Dnum=Dnumber} DEPARTMENT)$$
$$PROJ\_DEPT\_MGRS \leftarrow (CONTR\_DEPTS \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}(PROJ\_DEPT\_MGRS)$$

Query 4. Find the names of employees who work on all the projects controlled by department number 5.

$$DEPT5\_PROJS \leftarrow \rho_{(Pno)}(\pi_{Pnumber}(\sigma_{Dnum=5}(PROJECT)))$$
$$EMP\_PROJ \leftarrow \rho_{(Ssn,\ Pno)}(\pi_{Essn,\ Pno}(WORKS\_ON))$$
$$RESULT\_EMP\_SSNS \leftarrow EMP\_PROJ \div DEPT5\_PROJS$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(RESULT\_EMP\_SSNS * EMPLOYEE)$$

Query 5. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

$$SMITHS(Essn) \leftarrow \pi_{Ssn}(\sigma_{Lname='Smith'}(EMPLOYEE))$$
$$SMITH\_WORKER\_PROJS \leftarrow \pi_{Pno}(WORKS\_ON * SMITHS)$$
$$MGRS \leftarrow \pi_{Lname,\ Dnumber}(EMPLOYEE \bowtie_{Ssn=Mgr\_ssn} DEPARTMENT)$$
$$SMITH\_MANAGED\_DEPTS(Dnum) \leftarrow \pi_{Dnumber}(\sigma_{Lname='Smith'}(MGRS))$$
$$SMITH\_MGR\_PROJS(Pno) \leftarrow \pi_{Pnumber}(SMITH\_MANAGED\_DEPTS * PROJECT)$$
$$RESULT \leftarrow (SMITH\_WORKER\_PROJS \cup SMITH\_MGR\_PROJS)$$

Query 6. List the names of all employees with two or more dependents.

$$T1(Ssn,\ No\_of\_dependents) \leftarrow {}_{Essn}\Im_{COUNT\ Dependent\_name}(DEPENDENT)$$
$$T2 \leftarrow \sigma_{No\_of\_dependents>2}(T1)$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(T2 * EMPLOYEE)$$

Query 7. Retrieve the names of employees who have no dependents.

$$ALL\_EMPS \leftarrow \pi_{Ssn}(EMPLOYEE)$$
$$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$$
$$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$$

Query 8. List the names of managers who have at least one dependent.

$$MGRS(Ssn) \leftarrow \pi_{Mgr\_ssn}(DEPARTMENT)$$
$$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$$
$$MGRS\_WITH\_DEPS \leftarrow (MGRS \cap EMPS\_WITH\_DEPS)$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(MGRS\_WITH\_DEPS * EMPLOYEE)$$