

DBMS Lecture Notes (18MCA31)

MODULE 1

A **database** is a collection of related data. 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 hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications. Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalogue or dictionary; it is called meta-data. Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS. Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. Sharing a database allows multiple users and programs to access the database simultaneously.

An **application program** accesses the database by sending queries or requests for data to the DBMS. A query typically causes some data to be retrieved.

A **transaction** may cause some data to be read and some data to be written into the database.

Other important functions provided by the DBMS include protecting the database and maintaining it over a long period of time. **Protection** includes system protection against hardware or software malfunction (or crashes) and security protection against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

Characteristics of the Database Approach

In the database approach, a single repository maintains data that is defined once and then accessed by various users. In file systems, each application is free to name data elements independently. In contrast, in a database, the names or labels of data are defined once, and used repeatedly by queries, transactions, and applications. The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction processing

i. 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 DBMS catalogue, 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 catalogue is called **meta-data**, and it describes the structure of the primary database.

ii. **Insulation between programs and data, and data abstraction**

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

In some types of database systems, such as object-oriented and object-relational systems, users can define operations on data as part of the database definitions. An operation (also called a function or method) 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 DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a data model is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model hides storage and implementation details that are not of interest to most database users.

iii. **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. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views.

iv. **Sharing of data and multiuser transaction processing**

A multiuser DBMS 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 agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called **online transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently. The concept of a transaction has become central to many database applications. A transaction is an executing program or process that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction

appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are.

Actors on the Scene

In large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds of users. The people whose jobs involve the day-to-day use of a large database can be called as the actors on the scene.

i. Database Administrators

In any organization where many people 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, coordinating and monitoring its use, and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

ii. 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. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop views of the database that meet the data and processing requirements of these groups. Each view is then analyzed and integrated with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

iii. 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. The tasks that such users perform are varied:

- Bank tellers check account balances and post withdrawals and deposits.
- Reservation agents for airlines, hotels, and car rental companies check availability for a given request and make reservations.

■ **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.

■ **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

iv. **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 standard 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 should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

Workers behind the Scene

The workers behind the scene include the following categories:

■ **DBMS system designers and implementers** design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or modules, including modules for implementing the catalogue, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security. The DBMS must interface with other system software such as the operating system and compilers for various programming languages.

■ **Tool developers** design and implement tools—the software packages that facilitate database modeling and design, database system design, and improved 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. In many cases, independent software vendors develop and market these tools.

■ **Operators and maintenance personnel** (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although these categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database contents for their own purposes.

Advantages of using the DBMS Approach

i. **Controlling Redundancy**

The redundancy in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update multiple times: once for each file where the 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. Ideally, 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 is known as **data normalization**, and it ensures consistency and saves storage space. However, in practice, it is sometimes necessary to use **controlled redundancy** to improve the performance of queries.

ii. **Restricting Unauthorized Access**

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and 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. A DBMS should provide a **security and authorization subsystem**, which the DBA uses to create accounts and to specify account restrictions. Then, the DBMS should enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain privileged software, such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the predefined canned transactions developed for their use.

iii. Providing Persistent Storage for Program Objects

Databases can be used to provide persistent storage for program objects and data structures. This is one of the main reasons for object-oriented database systems. Programming languages typically have complex data structures, such as record types in Pascal or class definitions in C++ or Java. The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be persistent, since it survives the termination of program execution and can later be directly retrieved by another C++ program. The persistent storage of program objects and data structures is an important function of database systems.

iv. Providing Storage Structures and Search Techniques for Efficient Query Processing

Database systems must provide capabilities for efficiently executing queries and updates. Because the database is typically stored on disk, the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records. Auxiliary files called indexes are used for this purpose. Indexes are typically based on tree data structures or hash data structures that are suitably modified for disk search. In order to process the database records needed by a particular query, those records must be copied from disk to main memory. Therefore, the DBMS often has a buffering or caching module that maintains parts of the database in main memory buffers. In general, the operating system is responsible for disk-to-memory buffering. However, because data buffering is crucial to the DBMS performance, most DBMSs do their own data buffering. The query processing and optimization module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of physical database design and tuning, which is one of the responsibilities of the DBA staff.

v. 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 transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Alternatively, the recovery subsystem could ensure that the transaction is resumed from the point at which it was interrupted so that its full effect is recorded in the database. Disk backup is also necessary in case of a catastrophic disk failure.

vi. 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 standalone users.

Both forms-style interfaces and menu-driven interfaces are commonly known as graphical user interfaces (GUIs).

vii. Representing Complex Relationships among Data

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

viii. 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. A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. This is known as a referential integrity constraint. Another type of constraint specifies uniqueness on data item values. This is known as a key or uniqueness constraint. These constraints are derived from the meaning or semantics of the data and of the mini-world it represents. It is the responsibility of the database designers to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry. For typical large applications, it is customary to call such constraints business rules. A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of 'A' but a grade of 'C' is entered in the database, the DBMS cannot discover this error automatically because 'C' is a valid value for the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of 'Z' would be rejected automatically by the DBMS because 'Z' is not a valid value for the Grade data type.

ix. Permitting Inference 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 mini-world 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. In a traditional DBMS, an explicit procedural program code would have to be written to support such applications. But if the mini-world rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. In today's relational database systems, it is possible to associate **triggers** with tables. A trigger is a form of a rule activated by updates to the table, which results in performing some additional operations to some other tables, sending messages, and so on. More involved procedures to enforce rules are popularly called **stored procedures**; they become a part of the overall database definition and are invoked appropriately when certain conditions are met. More powerful functionality is provided by active database systems, which provide active rules that can automatically initiate actions when certain events and conditions occur.

x. Other benefits to most organizations

Potential for enforcing standards, reduced application development time, flexibility, and availability of up-to-date information and economies of scale are other benefits of DBMS approach to most organizations.

Data Models, Schemas and Instances

A data model is a collection of concepts that can be used to describe the structure of a database that provides the necessary means to achieve data abstraction.

i. Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level or conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level or physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks. Conceptual data models use concepts such as entities, attributes, and relationships. Concepts provided by low-level data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational (or implementation) data models** which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

ii. Schemas, Instances, and Database State

The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently. Most data models have certain conventions for displaying schemas as diagrams. A displayed schema is called a **schema diagram**. The following figure shows a schema diagram of a database. The diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

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
----------------	--------------------	-------

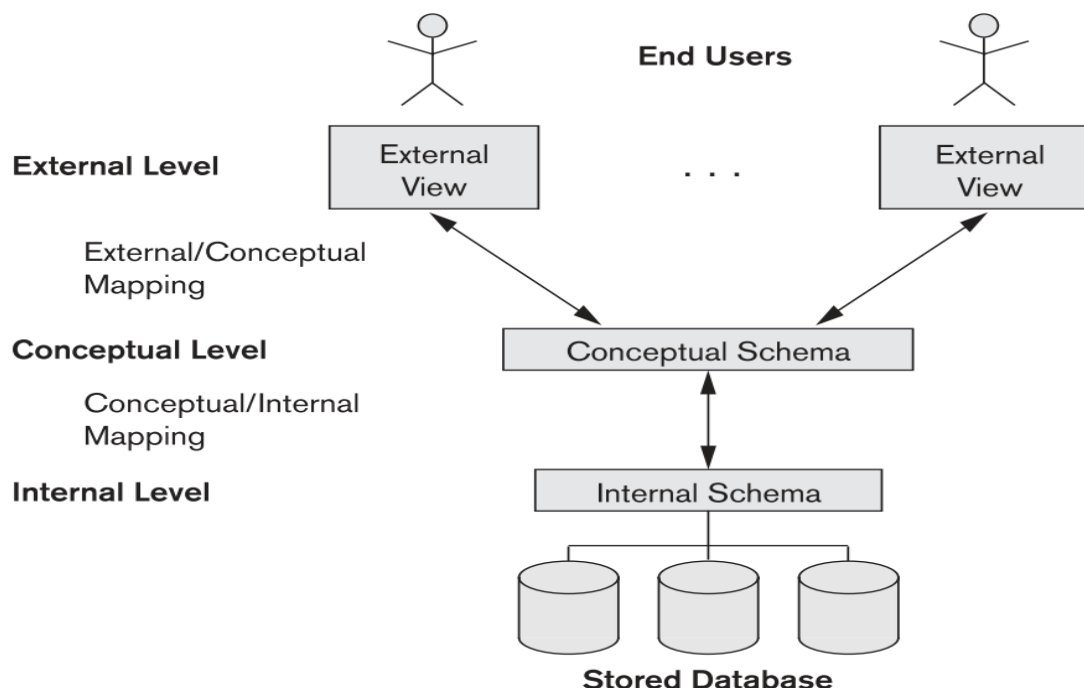
The data in the database at a particular moment in time is called a **database state or snapshot**. It is also called the current set of **occurrences or instances** in the database. In a given database state, each schema construct has its own current set of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state. We get the initial state of the database when the database is first populated or loaded with the initial

data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a current state. 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. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema. Although the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. This is known as **schema evolution**.

Three-Schema Architecture and Data Independence

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

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



Data Independence

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

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

Generally, physical data independence exists in most databases and file environments where physical details such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical data independence is harder to achieve because it allows structural and constraint changes without affecting application programs—a much stricter requirement. Whenever we have a multiple-level DBMS, its catalogue must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalogue. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the mapping between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed. The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

Database Languages and Interfaces

i. DBMS Languages

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

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. In most relational DBMSs today, there is no specific

language that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage. These permit the DBA staff to control indexing choices and mapping of data to storage. For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas. In relational DBMSs, SQL is used in the role of VDL to define user or application views as results of predefined queries.

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

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

There are two main types of DMLs. A **high-level or nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a precompiler and processed by the DBMS. A **low-level or procedural** DML must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. DL/1, a DML designed for the hierarchical model, is a low-level DML that uses commands such as GET UNIQUE, GET NEXT, or GET NEXT WITHIN PARENT to navigate from record to record within a hierarchy of records in the database. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time or set-oriented** DMLs. A query in a high-level DML often specifies which data to retrieve rather than how to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**. On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language. Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language.

ii. DBMS Interfaces

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

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

Forms-Based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages**, which are special languages that help programmers specify such forms. SQL*Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema.

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

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to understand them. A natural language interface usually has its own schema, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

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

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

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

The Database System Environment

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

i. DBMS Component Modules

The following figure illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions. The database and the DBMS catalogue are usually stored on disk. Access to the disk is controlled primarily by the operating system (OS), which schedules disk read/write. Many DBMSs have their own buffer management module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level stored data manager module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalogue. Let us consider the top part of Figure first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands. The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalogue. The catalogue includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints. In addition, the catalogue stores many other types of information that are needed by the DBMS modules, which can then look up the catalogue information as needed. Casual users and persons with occasional need for information from the database interact using some form of interface, which we call the interactive query interface in the Figure. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a query compiler that compiles them into an internal form. This internal query is subjected to query optimization. Among other things, the query optimizer is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution. It consults the system catalogue for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

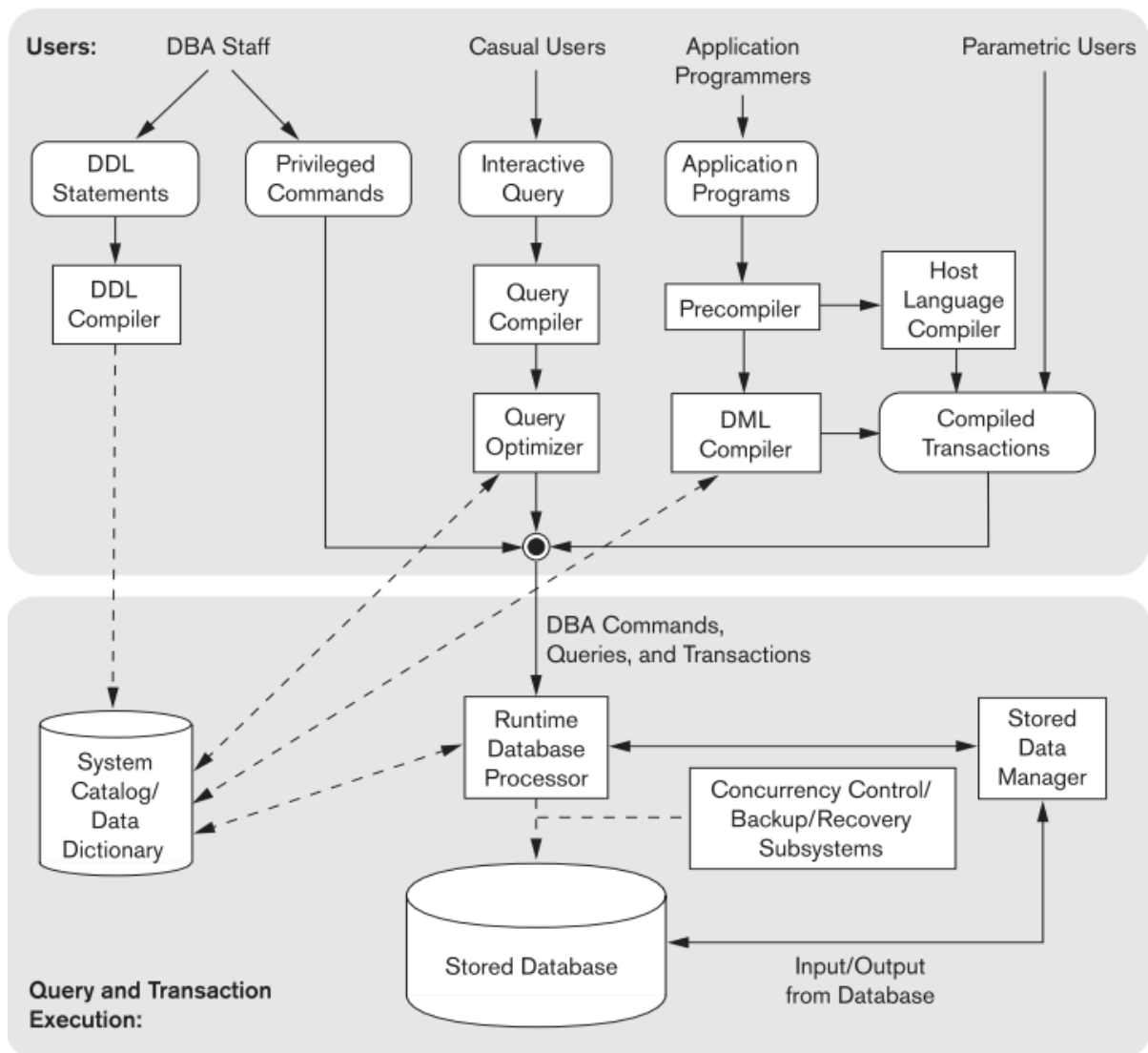


Figure: Component modules of a DBMS and their interactions

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The precompiler extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. Canned transactions are executed repeatedly by parametric users, who simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank withdrawal transaction where the account number and the amount may be supplied as parameters. In the lower part of Figure, the runtime database processor executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the system catalogue and may update it with statistics. It also works with the stored data manager, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module while others depend on the OS for buffer management. It is now common to have the client program that accesses the DBMS running on a separate computer from the computer on which the database resides. The former is called the client computer running a DBMS client software and the latter is called the database server. In some cases, the client accesses a middle

computer, called the application server, which in turn accesses the database server. The DBMS interacts with the operating system when disk accesses—to the database or to the catalogue—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory buffering of disk pages. The DBMS also interfaces with compilers for general purpose host programming languages, and with application servers and client programs running on separate machines through the system network interface.

ii. Database System Utilities

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

- **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called conversion tools.

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

- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations, and create new access paths to improve performance.

- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

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

iii. Tools, Application Environments, and Communications Facilities

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

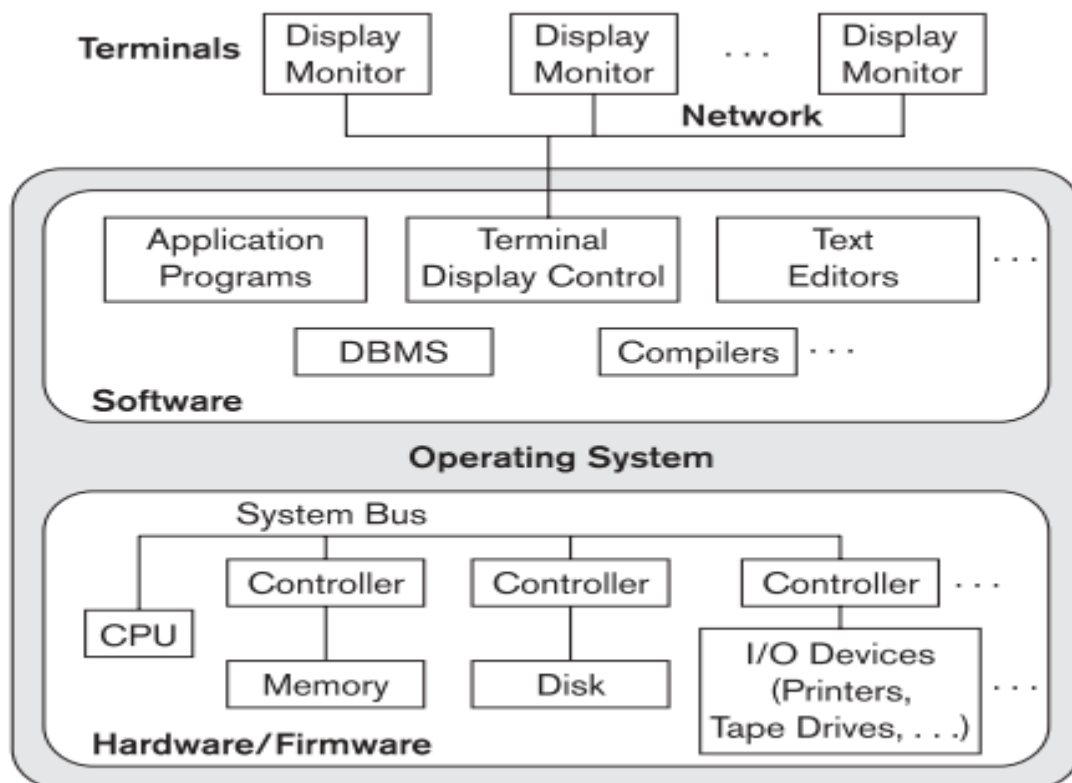
Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development,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

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

Centralized and Client/Server Architectures for DBMSs

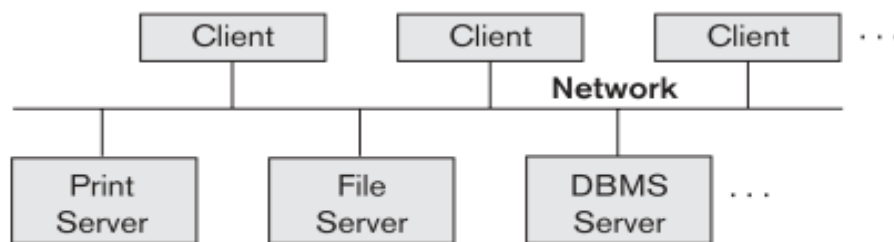
i. Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks. As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a centralized DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Following figure illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

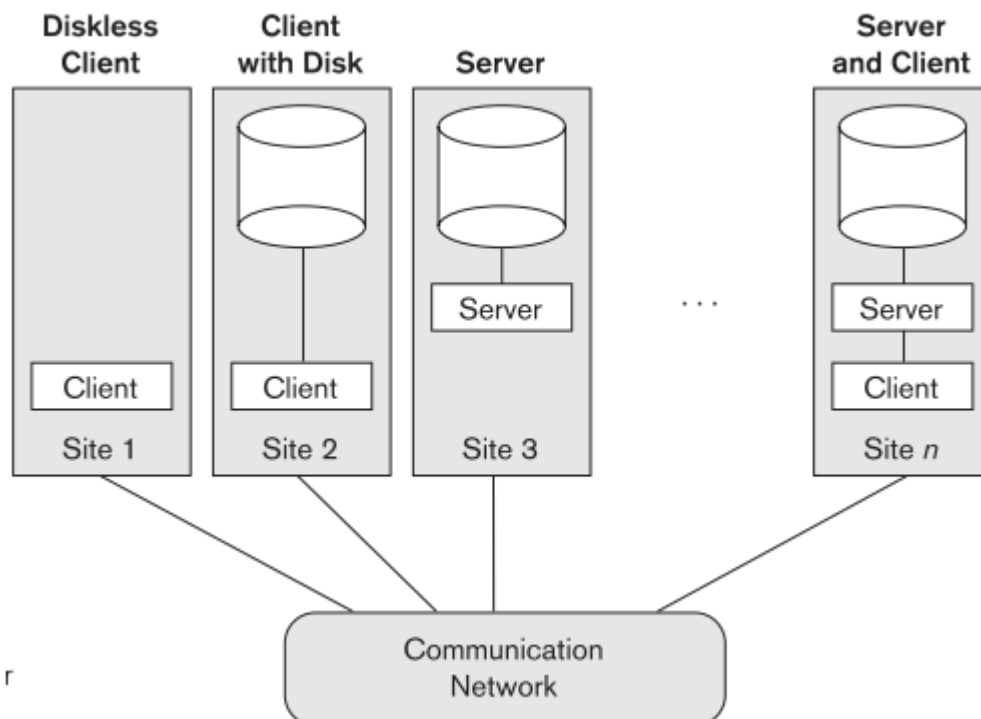


ii. Basic Client/Server Architectures

The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, data- base servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. Following figure illustrates client/server architecture at the logical level.



The next Figure is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless workstations or workstations/PCs with disks that have only client software installed).



Other machines would be dedicated servers, and others would have both client and server functionality. The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A client in this framework is typically a user machine

that provides user interface capabilities and local processing. When a client requires access to additional functionality— such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A server is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in the Figure. However, it is more common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: two-tier and three-tier.

iii. Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server or transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

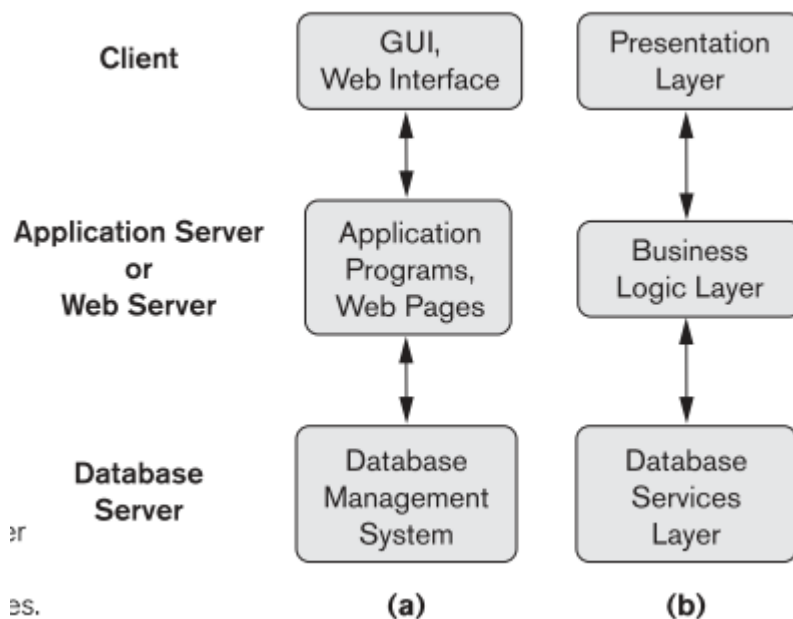
The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database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

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

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

iv. Three-Tier and n-Tier Architectures for Web Applications

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



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

It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to n-tier architectures, where n may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, n-tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently.

Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**. The **object data model** has been implemented in some commercial systems but has not had widespread use. Many legacy applications still run on database systems based on the **hierarchical** and

network data models. We can 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 PCs. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous** DDBMSs use the same DBMS software at all the sites, whereas heterogeneous DDBMSs can use different DBMS software at each site. It is also possible to develop middleware software to access several autonomous pre-existing databases stored under heterogeneous DBMSs. This leads to a federated DBMS (or multidatabase system), in which the participating DBMSs are loosely coupled and have a degree of local autonomy.

The fourth criterion is **cost**. It is difficult to propose a classification of DBMSs based on cost. The giant systems are being sold in modular form with components to handle distribution, replication, parallel processing, mobile capability, and so on, and with a large number of parameters that must be defined for the configuration. Furthermore, they are sold in the form of licenses—site licenses allow unlimited use of the database system with any number of copies running at the customer site. Another type of license limits the number of concurrent users or the number of user seats at a location.

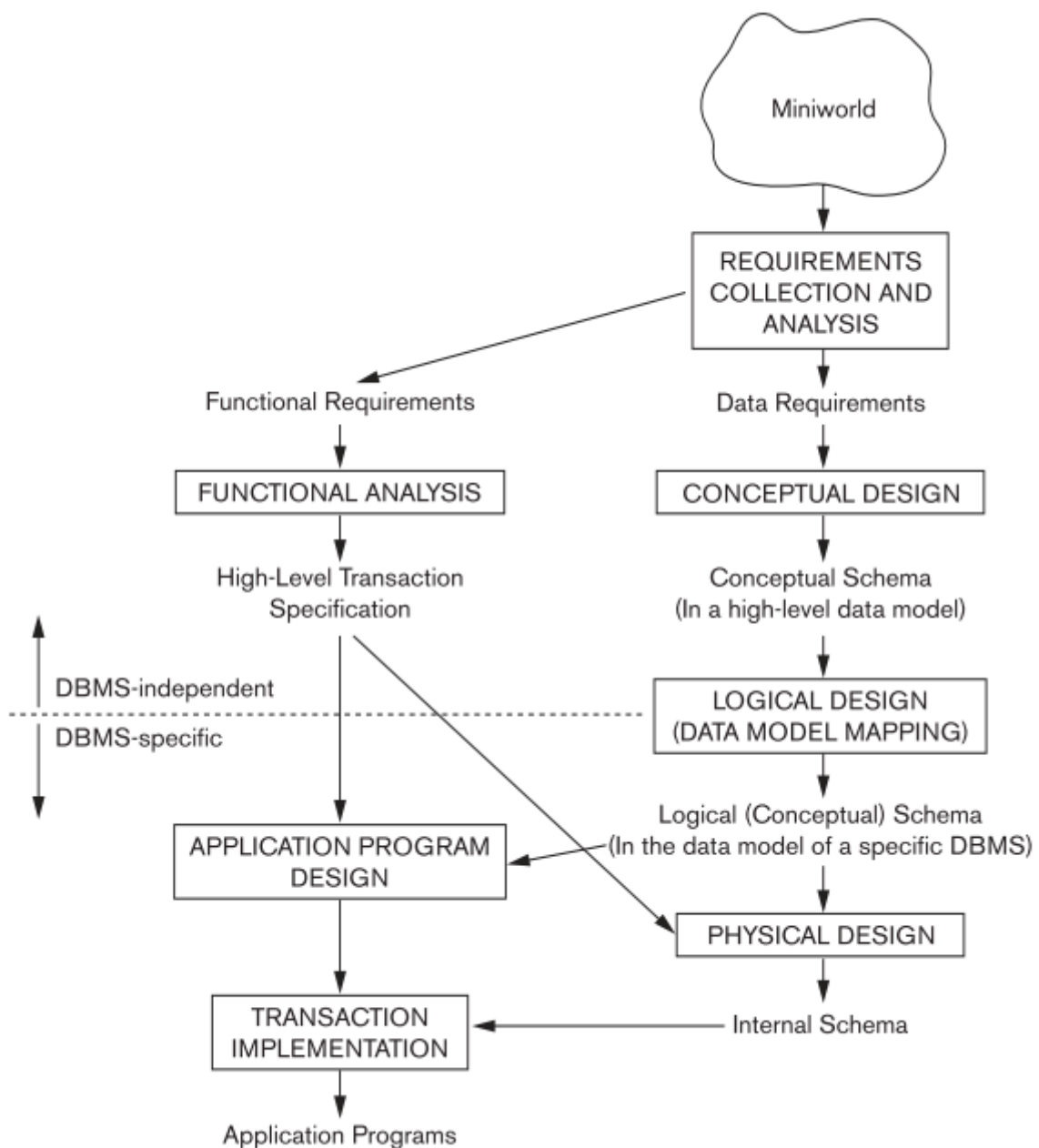
The fifth criterion is on the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed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

Entity-Relationship model

i. Using High-Level conceptual data models for database design

The following Figure shows a simplified overview 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, including both retrievals and updates. In software design, it is common to use data flow diagrams, sequence diagrams, scenarios, and other techniques to specify functional requirements.

Once 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. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it is easier to create a good conceptual database design.



During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and 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 using 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-relational 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**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semi-automated within the database design tools.

The last step is the physical design phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters 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.

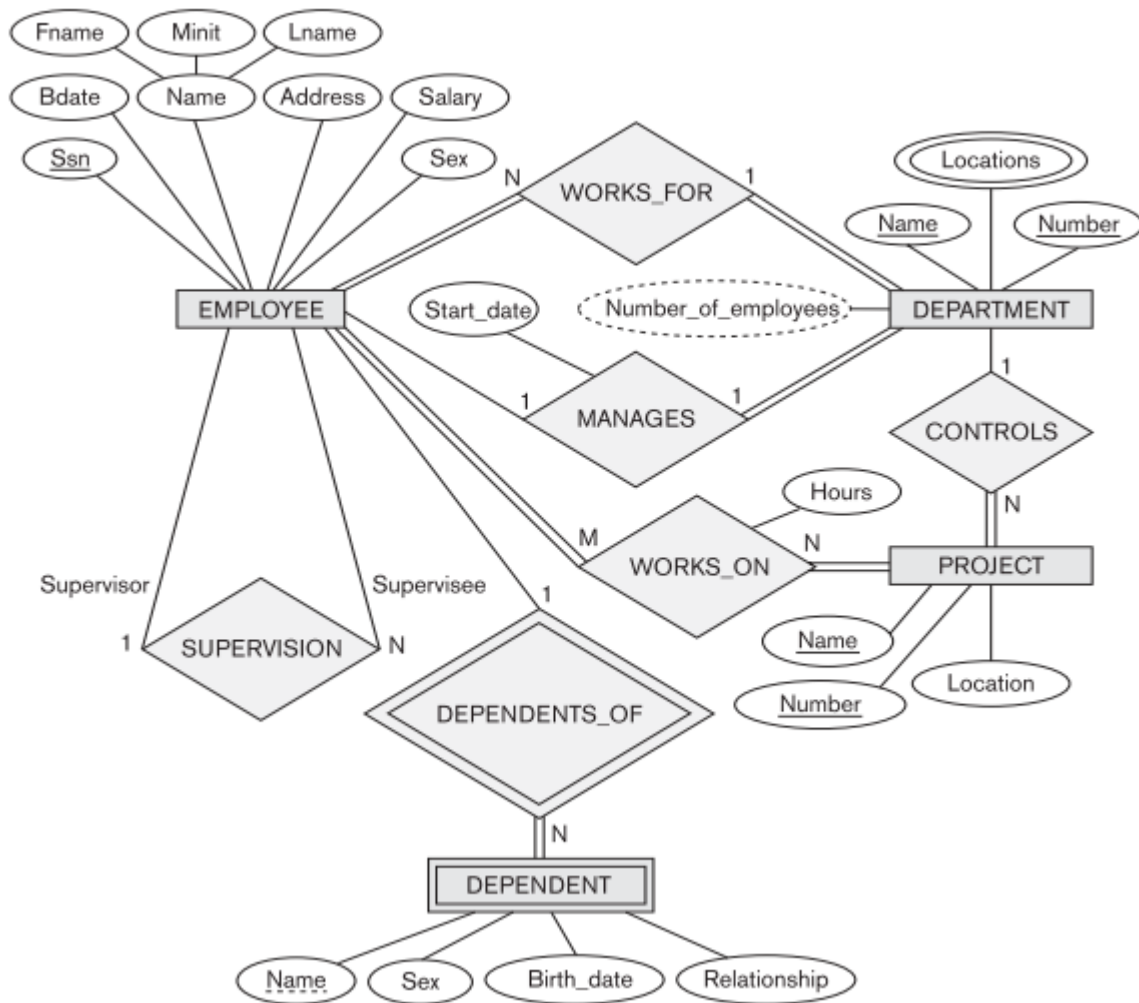
ii. A Sample Database Application

In this section we describe a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the *miniworld*—the part of the company that will be represented in the database.

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.

- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number, address, salary, sex (gender), 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 current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
- 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.

The following Figure shows how the schema for this database application can be displayed by means of the graphical notation known as ER diagrams.



iii. Entity Types, Entity Sets, Attributes, and Keys

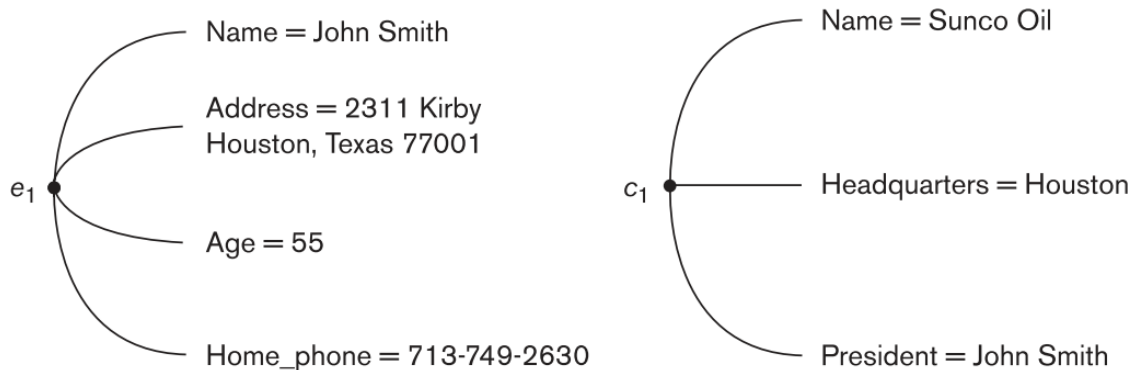
The ER model describes data as entities, relationships, and attributes.

a. Entities and Attributes

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 (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). 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.

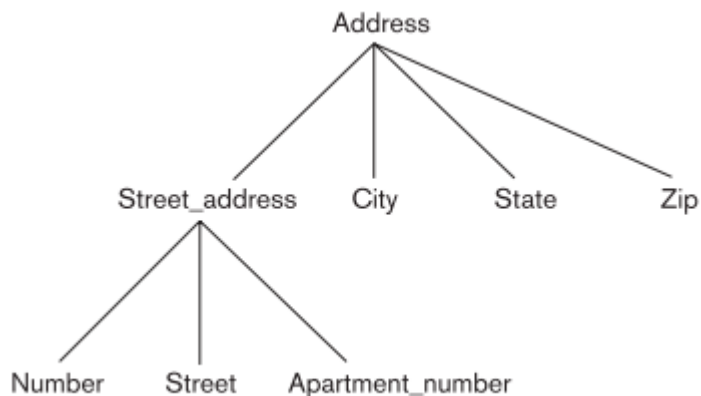
The following Figure shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respectively. The COMPANY entity c_1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.

Several types of attributes occur in the ER model: simple versus composite, single-valued versus multivalued, and stored versus derived. First we define these attribute types and illustrate their use via examples. Then we discuss the concept of a NULL value for an attribute.



Composite versus Simple (Atomic) Attributes

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in the above Figure can be subdivided into Street_address, City, State, and Zip, with the values '2311 Kirby', 'Houston', 'Texas', and '77001.' Attributes that are not divisible are called **simple or atomic attributes**. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in the next Figure. The value of a composite attribute is the concatenation of the values of its component simple attributes.



Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip Code, street, and so on), then the whole address can be designated as a simple attribute.

Single-Valued versus Multivalued Attributes

Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values.

Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different numbers of values for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

Stored versus Derived Attributes

In some cases, two (or more) attribute values are related—for example, the Age and Birth_date 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 Birth_date. The Age attribute is hence called a **derived attribute** and is said to be **derivable** from the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from related entities; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

NULL Values

In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number 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. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. 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 number 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 instance, 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 Home_phone attribute of a person is NULL.

Complex Attributes

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**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in the Figure. Both Phone and Address are themselves composite attributes.

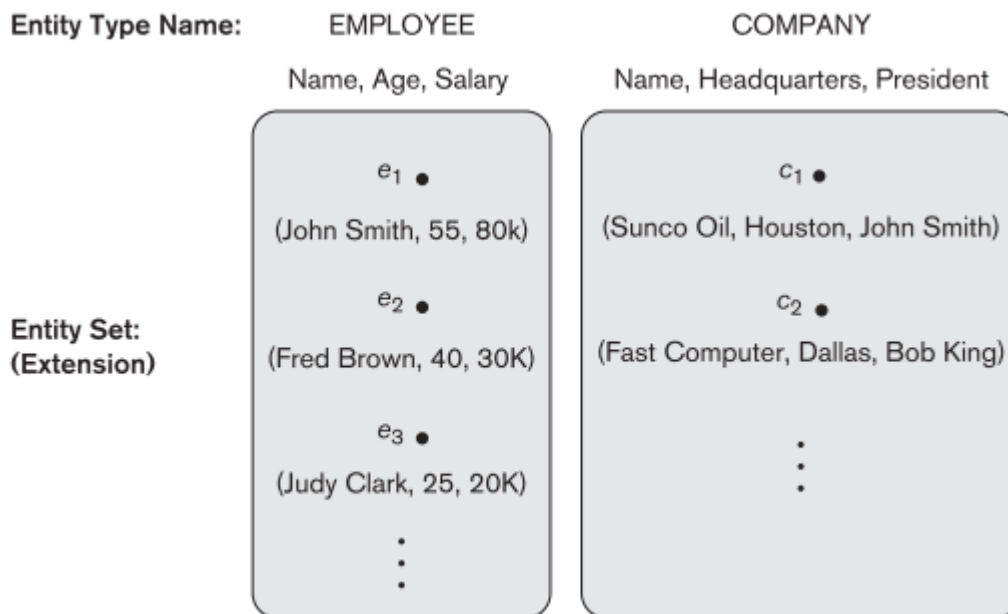
```
{Address_phone( {Phone(Area_code,Phone_number)},Address(Street_address  
(Number,Street,Apartment_number),City,State,Zip) )}
```

b. Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets

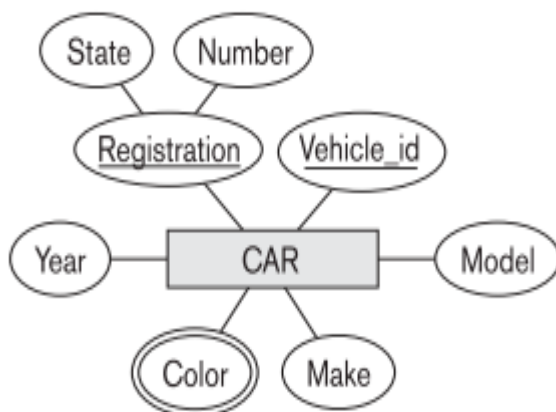
A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own value(s) for each

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. The following Figure shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for each.



A few individual entities of each type are also illustrated, along with the values of their attributes. 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. The following Figure shows a CAR entity type in this notation.



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 is 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 one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is SSN (Social Security Number). 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, the proper way to represent this in the ER model that we describe here is to define a composite attribute and designate it as a key attribute of the entity type. Notice that such a composite key must be minimal; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. 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 entity set 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. It is not the property of a particular entity set; rather, it is a constraint on any entity set of the entity type at any point in time. This key constraint is derived from the constraints of the mini-world that the database represents.

Some entity types have more than one key attribute. For example, each of the Vehicle_id 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, State and Number, 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. In our diagrammatic notation, if two attributes are underlined separately, then each is a key on its own. Unlike the relational model, there is no concept of primary key in the ER model that we present here; the primary key will be chosen during mapping to a relational schema.

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. If 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 to be the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are not displayed in ER diagrams, and are typically specified using the basic data types available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. Additional data types to represent common database types such as date, time, and other concepts are also employed.

Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set P(V) of V:

$$A : E \rightarrow P(V)$$

We refer to the value of attribute A for entity e as A(e). The previous definition covers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the empty set. For single-valued attributes, A(e) is restricted to being a singleton set for each entity e in E, whereas there is no restriction on multivalued attributes. For a composite attribute A, the value set V is the power set of the Cartesian product of $P(V_1), P(V_2), \dots, P(V_n)$, where V_1, V_2, \dots, V_n are the value sets of the simple component attributes that form A:

$$V = P(P(V_1) \times P(V_2) \times \dots \times P(V_n))$$

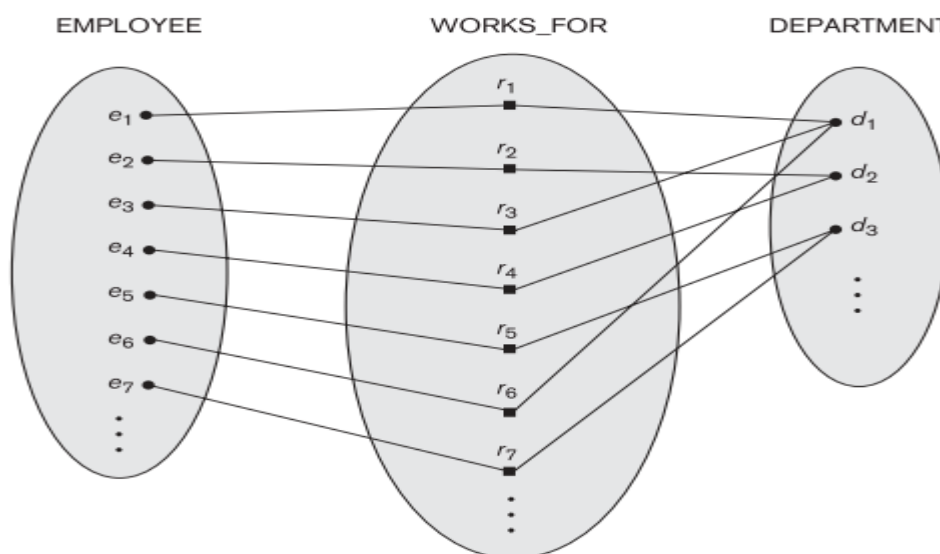
The value set provides all possible values. Usually only a small number of these values exist in the database at a particular time. Those values represent the data from the current state of the mini-world. They correspond to the data as it actually exists in the mini-world.

iv. Relationship Types, Relationship Sets, Roles, and Structural Constraints

a. Relationship Types, Sets, and Instances

A relationship type R among n entity types E_1, E_2, \dots, E_n defines a set of associations—or a relationship set—among entities from these entity types. As for the case of 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 r_i , where each r_i associates n individual entities (e_1, e_2, \dots, e_n) , and each entity e_j in r_i is a member of entity set E_j , $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on E_1, E_2, \dots, E_n ; alternatively, it can be defined as a subset of the Cartesian product of the entity sets $E_1 \times E_2 \times \dots \times E_n$. Each of the entity types E_1, E_2, \dots, E_n is said to participate in the relationship type R ; similarly, each of the individual entities e_1, e_2, \dots, e_n is said to **participate** in the relationship instance $r_i = (e_1, e_2, \dots, e_n)$.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i 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 for which the employee works in the corresponding entity set. Each relationship instance in the relationship set `WORKS_FOR` associates one `EMPLOYEE` entity and one `DEPARTMENT` entity. The following Figure illustrates this example, where each relationship instance r_i is shown connected to the `EMPLOYEE` and `DEPARTMENT` entities that participate in r_i . In the miniworld represented by the following Figure, employees e_1, e_3 , and e_6 work for department d_1 ; employees e_2 and e_4 work for department d_2 ; and employees e_5 and e_7 work for department d_3 .

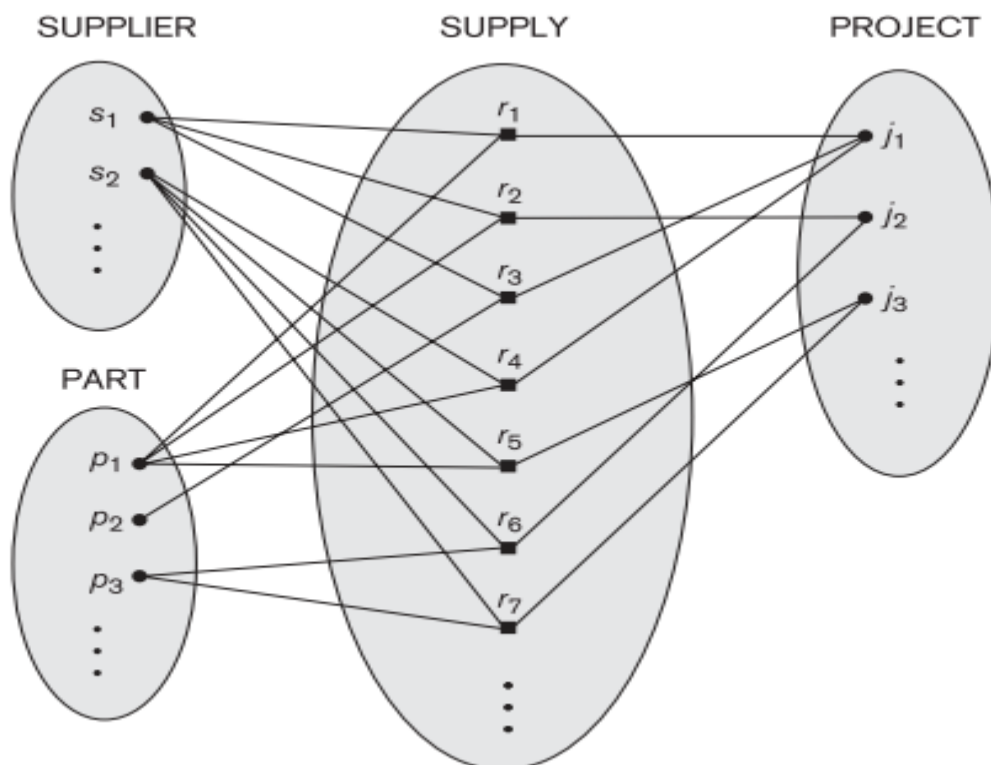


In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box.

b. Relationship Degree, Role Names, and Recursive Relationships

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 the following Figure where each relationship instance r_i 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 binary relationship type in terms of attributes. Consider the WORKS_FOR relationship type. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of all DEPARTMENT entities, which is the DEPARTMENT entity set.

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 participating 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 the role that each participating entity plays. Such relationship types are called recursive relationships. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. 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).

c. Constraints on Binary Relationship types

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 miniworld 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 binary relationship constraints: **cardinality ratio and participation**.

Cardinality Ratios for Binary Relationships

The cardinality ratio for a binary relationship specifies the maximum 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 (that is, employs) any number of employees, but an employee can be related to (work for) only one department. This means that for this particular relationship WORKS_FOR, a particular department entity can be related to any number of employees (N indicates there is no maximum number). On the other hand, an employee can be related to a maximum of one 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—at any point in time—an employee can manage one department only and a department can have one manager only. The relationship type WORKS_ON is of cardinality ratio M:N, because the mini-world rule is that an employee can work on several projects and a project can have several employees.

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. This constraint specifies the minimum number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint**. There are two types of participation constraints—total and partial. If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one 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**. 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 some department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type. In ER diagrams, total participation (or existence dependency) is

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

d. 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 Start_date 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 Start_date 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 Start_date 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 on the N-side of the relationship. For example, if the WORKS_FOR relationship also has an attribute Start_date 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 works for only one department, and hence participates in at most one relationship instance in WORKS_FOR.












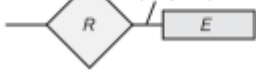
v. Weak Entity Types

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 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 one 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. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (License_number) and hence is not a weak entity.

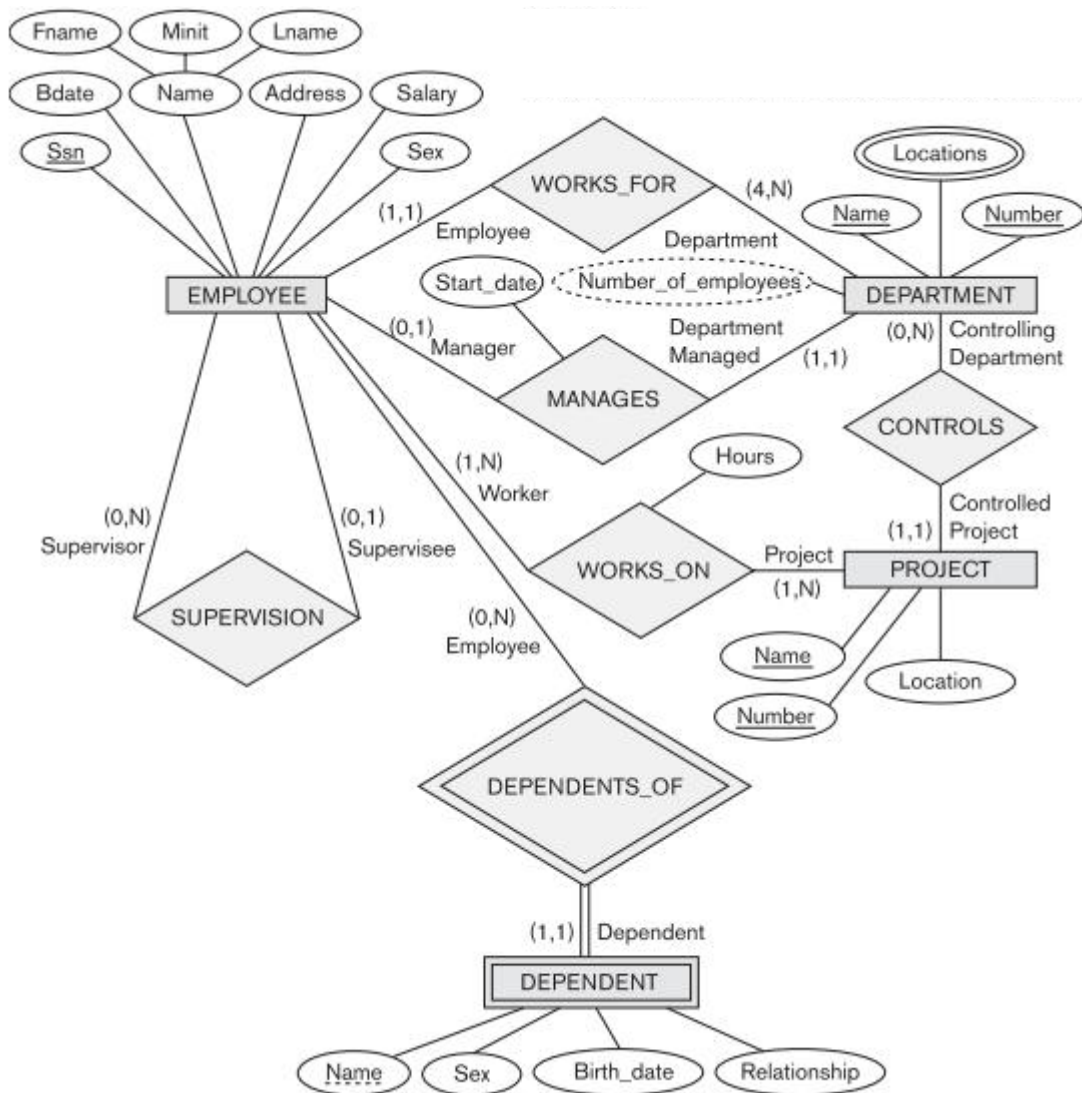
Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship. In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of two distinct employees may, by chance, have the same values for Name, Birth_date, 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 attribute 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.

Summary of the notation for ER diagrams

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1 : N for E_1 : E_2 in R
	Structural Constraint (min, max) on Participation of E in R

ER diagrams for the COMPANY schema with structural constraints specified using (min,max) notation and role names



DBMS Lecture Notes (18MCA31)

MODULE 2

Relational Model Concepts

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a flat file of records. It is called a flat file because each record has a simple linear or flat structure. There are important differences between relations and files.

When a relation is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is represented by a domain of possible values. We now define these terms—domain, tuple, attribute, and relation—formally.

i. Domains, Attributes, Tuples, and Relations

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- **Usa_phone_numbers**. The set of ten-digit phone numbers valid in the United States.
- **Local_phone_numbers**. The set of seven-digit phone numbers valid within a particular area code in the United States.
- **Social_security_numbers**. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- **Names**: The set of character strings that represent names of persons.

The preceding are called logical definitions of domains. A **data type or format** is also specified for each domain. For example, the data type for the domain **Usa_phone_numbers** can be declared as a character string of the form $(ddd)ddddddd$, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for **Employee_ages** is an integer number between 15 and 80. For **Academic_department_names**, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as **Person_weights** should have the units of measurement, such as pounds or kilograms.

A **relation schema** R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n . Each attribute A_i is the name of a role played by some domain D in the relation schema R . D is called the domain of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to describe a relation; R is called the name of this relation. The degree (or arity) of a relation is the number of attributes n of its relation schema.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Using the data type of each attribute, the definition is sometimes written as:

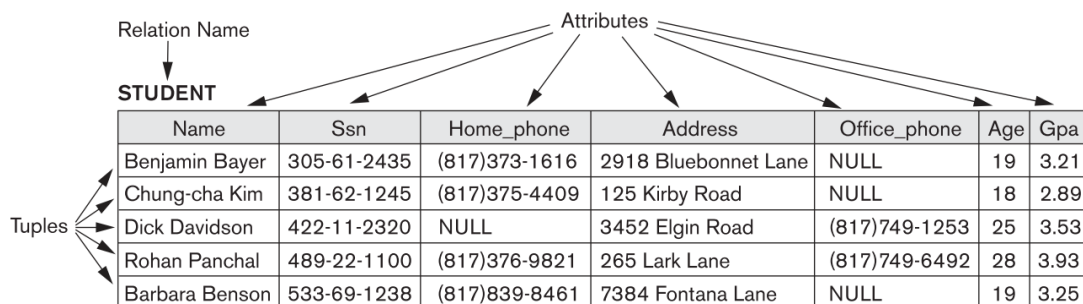
STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the preceding definition, we showed assignment of generic types such as string or integer to the attributes. More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation:

dom(Name) = Names; dom(Ssn) = Social_security_numbers; dom(HomePhone) =USA_phone_numbers, dom(Office_phone) = USA_phone_numbers, and dom(Gpa) =Grade_point_averages. It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn, whereas the fourth attribute is Address.

A **relation (or relation state)** r of the relation schema $R (A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$. Each **n -tuple** t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value. The i^{th} value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

The following Figure shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column header indicating a role or interpretation of the values in that column. NULL values represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.



The earlier definition of a relation can be restated more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $\text{dom}(A_1)$, $\text{dom}(A_2)$, ..., $\text{dom}(A_n)$, which is a subset of the **Cartesian product** (denoted by \times) of the domains that define R :

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality**, in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \dots \times |\text{dom}(A_n)|$$

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation state, by being transformed into another relation state. However, the schema R is relatively static and changes very infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to have the same domain. The attribute names indicate different **roles**, or interpretations, for the domain. For example, in the *STUDENT* relation, the same domain *USA_phone_numbers* plays the role of *Home_phone*, referring to the home phone of a student, and the role of *Office_phone*, referring to the office phone of the student. A third possible attribute (not shown) with the same domain could be *Mobile_phone*.

ii. Characteristics of Relations

Ordering of Tuples in a Relation A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, i^{th} , and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Ordering of Values within a Tuple and an Alternative Definition of a Relation According to the preceding definition of a relation, an n -tuple is an ordered list of n values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is not that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple unnecessary. In this definition, a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes (instead of a list), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \dots, t_m\}$, where each tuple t_i is a mapping from R to D , and D is the union (denoted by \cup) of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r . Each mapping t_i is called a tuple. According to this definition of tuple as a mapping, a tuple can be considered as a set of $\langle \text{attribute}, \text{value} \rangle$ pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is not important, because the attribute name appears with its value. By this definition, the two tuples shown in the following Figure are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple.

Two identical tuples when the order of attributes and values is not part of relation definition.

$$t = \langle (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Home_phone, NULL}), (\text{Address, 3452 Elgin Road}), (\text{Office_phone, (817)749-1253}), (\text{Age, 25}), (\text{Gpa, 3.53}) \rangle$$
$$t = \langle (\text{Address, 3452 Elgin Road}), (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Age, 25}), (\text{Office_phone, (817)749-1253}), (\text{Gpa, 3.53}), (\text{Home_phone, NULL}) \rangle$$

Values and NULLs in the Tuples Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption. Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone does not apply to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is unknown). In general, we can have several meanings for NULL values, such as **value unknown**, **value exists but is not available**, or **attribute does not apply** to this tuple (also known as **value undefined**). An example of the last type of NULL will occur if we add an attribute *Visa_status* to the STUDENT relation that applies only to tuples representing foreign students. It is possible to devise different codes for different meanings of NULL values.

The exact meaning of a NULL value governs how it fares during arithmetic aggregations or comparisons with other values. For example, a comparison of two NULL values leads to ambiguities— if both Customer A and B have NULL addresses, it does not mean they have the same address. During database design, it is best to avoid NULL values as much as possible.

Interpretation of a Relation The relation schema can be interpreted as a declaration or a type of assertion. For example, the schema of the STUDENT relation asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a fact or a particular instance of the assertion. For example, the first tuple in Figure asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about entities, whereas other relations may represent facts about relationships. For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline. Hence, the relational model represents facts about both entities and relationships uniformly as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type.

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that satisfy the predicate. For example, the predicate STUDENT (Name, Ssn, ...) is true for the five tuples in relation STUDENT. These tuples represent five different propositions or facts in the real world. An assumption called the closed world assumption states that

the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false.

iii. Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, \dots, A_n)$.
- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as $STUDENT$ also indicates the current set of tuples in that relation—the current relation state—whereas $STUDENT(\text{Name}, \text{Ssn}, \dots)$ refers only to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$ —for example, $STUDENT.\text{Name}$ or $STUDENT.\text{Age}$. This is because the same name may be used for two attributes in different relations. However, all attribute names in a particular relation must be distinct.
- An n -tuple t in a relation $r(R)$ is denoted by $t = \langle v_1, v_2, \dots, v_n \rangle$, where v_i is the value corresponding to attribute A_i . The following notation refers to **component values** of tuples:
 - Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i .
 - Both $t[A_u, A_w, \dots, A_z]$ and $t.(A_u, A_w, \dots, A_z)$, where A_u, A_w, \dots, A_z is a list of attributes from R , refer to the subtuple of values $\langle v_u, v_w, \dots, v_z \rangle$ from t corresponding to the attributes specified in the list.

As an example, consider the tuple $t = \langle \text{'Barbara Benson'}, \text{'533-69-1238'}, \text{'(817)8398461'}, \text{'7384 Fontana Lane'}, \text{NULL}, 19, 3.25 \rangle$ from the $STUDENT$ relation; we have $t[\text{Name}] = \langle \text{'Barbara Benson'} \rangle$, and $t[\text{Ssn}, \text{Gpa}, \text{Age}] = \langle \text{'533-69-1238'}, 3.25, 19 \rangle$.

Relational Model Constraints and Relational Database Schemas

In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or constraints on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents. In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints or implicit constraints**.
2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL (data definition language). We call these **schema-based constraints or explicit constraints**.

3. Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based or semantic constraints or business rules**.

The characteristics of relations are the inherent constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL. Constraints in the third category are more general, relate to the meaning as well as behavior of attributes, and are difficult to express and enforce within the data model, so they are usually checked within the application programs that perform database updates.

Another important category of constraints is data dependencies, which include functional dependencies and multivalued dependencies. They are used mainly for testing the “goodness” of the design of a relational database and are utilized in a process called normalization.

The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

i. Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

ii. Key Constraints and Constraints on NULL Values

In the formal relational model, a relation is defined as a set of tuples. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK ; then for any two distinct tuples t_1 and t_2 in a relation state r of R , we have the constraint that:

$$t_1[SK] \neq t_2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema R . A superkey SK specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value for SK . Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a key, which has no redundancy. A key K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.

2. It is a minimal superkey—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold. This property is not required by a superkey.

Whereas the first property applies to both keys and superkeys, the second property is required only for keys. Hence, a key is also a superkey but not vice versa. Consider the STUDENT relation. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn. Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require all its attributes together to have the uniqueness property.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on every valid relation state of the schema. A key is determined from the meaning of the attributes, and the property is time-invariant: It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 3.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to identify tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined. Notice that when a relation schema has several candidate keys, the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as unique keys, and are not underlined.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

iii. Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema.

A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of **integrity constraints** IC . A **relational database state** DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC . The following Figure shows a relational database schema that we call $COMPANY = \{\text{EMPLOYEE}, \text{DEPARTMENT}, \text{DEPT_LOCATIONS}, \text{PROJECT}, \text{WORKS_ON}, \text{DEPENDENT}\}$. The underlined attributes represent primary keys.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

The next Figure shows a relational database state corresponding to the COMPANY schema. When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called **an invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state**.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	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	<u>Dnumber</u>	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

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

<u>Essn</u>	<u>Pno</u>	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

<u>Pname</u>	<u>Pnumber</u>	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

<u>Essn</u>	<u>Dependent_name</u>	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

The Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different realworld concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have identical attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation: once in the role of the employee's SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

iv. Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. For example, in the above Figure, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a foreign key. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2. A set of attributes FK in relation schema R1 is a **foreign key** of R1 that **references** relation R2 if it satisfies the following rules:

- The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.
- A value of FK in a tuple t1 of the current state r1(R1) either occurs as a value of PK for some tuple t2 in the current state r2(R2) or is NULL. In the former case, we have $t1[FK] = t2[PK]$, and we say that the tuple t1 **references or refers** to the tuple t2.

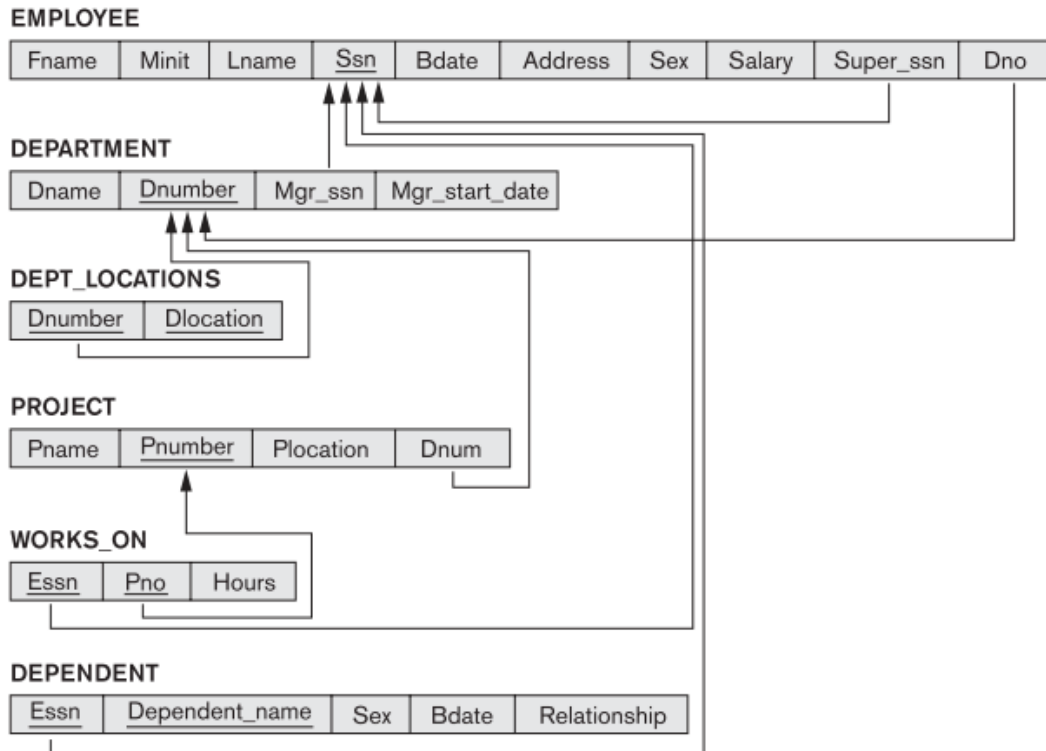
In this definition, R1 is called the **referencing relation** and R2 is the **referenced relation**. If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas. For example, consider the database shown in the above Figure. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t2 of the DEPARTMENT relation, or the value of Dno can be NULL if the employee does not belong to a department or will be assigned to a department later. For example, in the above Figure, the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.

Notice that a foreign key can refer to its own relation. For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure, the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith.'

We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. The next Figure shows the schema with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.



v. Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. However, they do not include a large class of general constraints, sometimes called semantic integrity constraints, which may have to be specified and enforced on a relational database. Examples of such constraints are the salary of an employee should not exceed the salary of the employee's supervisor and the maximum number of hours an employee can work on all projects per week is 56. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers and assertions** can be used. In SQL, CREATE ASSERTION and CREATE TRIGGER statements can be used for this purpose.

Another type of constraint is the functional dependency constraint, which establishes a functional relationship among two sets of attributes X and Y. This constraint specifies that the value of X determines a unique value of Y in all states of a relation.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a valid state of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database. An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers.

Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into retrievals and updates. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a

query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. That **result relation** becomes the answer to (or result of) the user's query.

In this section, we concentrate on the database **modification or update** operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update (or Modify)** is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in above Figure for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in the following Figure. For each type of operation, we give some examples and discuss any constraints that each operation may violate.

i. The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R . Insert can violate any of the constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation $r(R)$. Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

■ Operation:

Insert $\langle \text{'Cecilia'}, \text{'F'}, \text{'Kolonsky'}, \text{NULL}, \text{'1960-04-05'}, \text{'6357 Windy Lane, Katy, TX'}, \text{F}, \text{28000}, \text{NULL}, \text{4} \rangle$ into EMPLOYEE.

Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

■ Operation:

Insert $\langle \text{'Alicia'}, \text{'J'}, \text{'Zelaya'}, \text{'999887777'}, \text{'1960-04-05'}, \text{'6357 Windy Lane, Katy, TX'}, \text{F}, \text{28000}, \text{'987654321'}, \text{4} \rangle$ into EMPLOYEE.

Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

■ Operation:

Insert $\langle \text{'Cecilia'}, \text{'F'}, \text{'Kolonsky'}, \text{'677678989'}, \text{'1960-04-05'}, \text{'6357 Windswept, Katy, TX'}, \text{F}, \text{28000}, \text{'987654321'}, \text{7} \rangle$ into EMPLOYEE.

Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

■ Operation:

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.

Result: This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to reject the insertion. In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to correct the reason for rejecting the insertion, but this is typically not used for violations caused by Insert; rather, it is used more often in correcting violations for Delete and Update. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In operation 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for Mgr_ssn that does not exist in the EMPLOYEE relation.

ii. The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

■ Operation:

Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.

Result: This deletion is acceptable and deletes exactly one tuple.

■ Operation:

Delete the EMPLOYEE tuple with Ssn = '999887777'.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

■ Operation:

Delete the EMPLOYEE tuple with Ssn = '333445555'.

Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option, called restrict, is to reject the deletion. The second option, called cascade, is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = '999887777'. A third option, called set null or set default, is to modify the referencing attribute values that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple. Notice that if a referencing attribute that causes a violation is part of the primary key, it cannot be set to NULL; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = '333445555'. Tuples in EMPLOYEE with Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples

that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to specify which of the options applies in case of a violation of the constraint.

iii. The Update Operation

The **Update (or Modify)** operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation R. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

- Operation:
Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
Result: Acceptable.
- Operation:
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
Result: Acceptable.
- Operation:
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
Result: Unacceptable, because it violates referential integrity.
- Operation:
Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is neither part of a primary key nor of a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation. In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a violation caused by Update.

iv. The Transaction Concept

A database application program running against a relational database typically executes one or more transactions. A transaction is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations, and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

A large number of commercial applications running against relational databases in **online transaction processing (OLTP)** systems are executing transactions at rates that reach several hundred per second.

Relational Algebra

The basic set of operations for the relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as relational algebra expressions. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs). Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs. Although most commercial RDBMSs in use today do not provide user interfaces for relational algebra queries, the core operations and functions in the internal modules of most relational systems are based on relational algebra operations.

i. Unary Relational Operations: SELECT and PROJECT

• The SELECT Operation

The SELECT operation is used to choose a subset of the tuples from a relation that satisfies a **selection condition**. One can consider the SELECT operation to be a filter that keeps only those tuples that satisfy a qualifying condition. Alternatively, 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:

$$\sigma_{Dno=4}(EMPLOYEE)$$
$$\sigma_{Salary>3000}(EMPLOYEE)$$

In general, the SELECT operation is denoted by

$$\sigma_{\langle \text{selection condition} \rangle}(R)$$

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. Notice that R is generally a relational algebra expression whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the same attributes as R.

The Boolean expression specified in $\langle \text{selection condition} \rangle$ is made up of a number of **clauses** of the form

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{constant value} \rangle$

or

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{attribute name} \rangle$

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators {=,<,≤,>,≥,≠}, and <constant value> is a constant value from the attribute domain. 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 SE LECT operation:

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

The result is shown in the following figure (a).

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
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

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Notice that all the comparison operators in the set {=,<,≤,>,≥,≠} can apply to attributes whose domains are ordered values, such as numeric or date domains. Domains of strings of characters are also considered to be ordered based on the collating sequence of the characters. If the domain of an attribute is a set of unordered values, then only the comparison operators in the set {=,≠} can be used. An example of an unordered domain is the domain Color={ 'red', 'blue', 'green', 'white', 'yellow', ...}, where no order is specified among the various colors.

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. That is, $|\sigma_C(R)| \leq |R|$ for any condition C. 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,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(R)) = \sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond1} \rangle}(R))$$

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,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\dots(\sigma_{\langle \text{condn} \rangle}(R)) \dots)) = \sigma_{\langle \text{cond1} \rangle \text{ AND} \langle \text{cond2} \rangle \text{ AND} \dots \text{ AND} \langle \text{condn} \rangle}(R)$$

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

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

would correspond to the following SQL query:

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

- **The PROJECT Operation**

If we think of a relation as a table, the SELECT operation chooses some of the rows from the table while discarding other rows. The PROJECT operation, on the other hand, 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:

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

The resulting relation is shown in Figure (b). The general form of the PROJECT operation is

$$\pi_{\langle \text{attribute list} \rangle}(R)$$

where π (π) is the symbol used to represent the PROJECT operation, and $\langle \text{attribute list} \rangle$ is the desired sublist of attributes from the attributes of relation R . Again, notice that R is, in general, a relational algebra expression whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in $\langle \text{attribute list} \rangle$ in the same order as they appear in the list. Hence, its degree is equal to the number of attributes in $\langle \text{attribute list} \rangle$.

If the attribute list includes only non-key attributes of R , duplicate tuples are likely to occur. 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**. For example, consider the following PROJECT operation:

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

The result is shown in Figure (c). Notice that the tuple $\langle 'F', 25000 \rangle$ appears only once in Figure (c), even though this combination of values appears twice in the EMPLOYEE relation.

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R . If the projection list is a superkey of R —that is, it includes some key of R —the resulting relation has the same number of tuples as R . Moreover,

$$\pi_{\langle \text{list1} \rangle}(\pi_{\langle \text{list2} \rangle}(R)) = \pi_{\langle \text{list1} \rangle}(R)$$

as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$; 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:

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

would correspond to the following SQL query:

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

Notice that if we remove the keyword DISTINCT from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra.

- **Sequences of Operations and the RENAME Operation**

The relations shown in the above Figure that depict operation results do not have any names. In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, 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:

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

The next Figure (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 follows:

```
DEP5_EMPS ←  $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$ 
RESULT ←  $\pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$ 
```

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. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

```
TEMP ←  $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$ 
R(First_name, Last_name, Salary) ←  $\pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$ 
```

These two operations are illustrated in Figure (b).

(a)

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

(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

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

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:

$$\rho_{S(B_1, B_2, \dots, B_n)}(R) \text{ or } \rho_S(R) \text{ or } \rho_{(B_1, B_2, \dots, B_n)}(R)$$

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. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are (A1, A2, ..., An) in that order, then each Ai is renamed as Bi.

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;
```

ii. Relational Algebra Operations from Set Theory

- The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations are the standard mathematical operations on sets. 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:

$$\begin{aligned} \text{DEP5_EMPS} &\leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE}) \\ \text{RESULT1} &\leftarrow \pi_{\text{Ssn}}(\text{DEP5_EMPS}) \\ \text{RESULT2}(\text{Ssn}) &\leftarrow \pi_{\text{Super_ssn}}(\text{DEP5_EMPS}) \\ \text{RESULT} &\leftarrow \text{RESULT1} \cup \text{RESULT2} \end{aligned}$$

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 next figure), while eliminating any duplicates. Thus, the Ssn value '333445555' appears only once in the result.

RESULT1	RESULT2	RESULT
Ssn	Ssn	Ssn
123456789	333445555	123456789
333445555	888665555	333445555
666884444		666884444
453453453		453453453
		888665555

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION, INTERSECTION, and SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). 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 $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be union compatible (or type compatible) if they have the same degree n and if $\text{dom}(A_i) = \text{dom}(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

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

- **UNION:** The result of this operation, denoted by $R \cup S$, 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.
- **INTERSECTION:** The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S .
- **SET DIFFERENCE (or MINUS):** The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S .

We will adopt the convention that the resulting relation has the same attribute names as the first relation R . It is always possible to rename the attributes in the result using the rename operator.

The next Figure illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure (a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. The result of the UNION operation in Figure (b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure (c)) includes only those who are both students and instructors.

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

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

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,

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

The MINUS operation is not commutative; that is, in general,
 $R - S \neq S - R$

(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

Figure (d) shows the names of students who are not instructors, and Figure (e) shows the names of instructors who are not students.

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

$$R \cap S = ((R \cup S) - (R - S)) - (S - R)$$

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates.

- **The CARTESIAN PRODUCT (CROSS PRODUCT) Operation**

Next, we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by \times . This is also a binary set operation, but the relations on which it is applied do not have to be union compatible. In its binary form, 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 $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, 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 n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

The n -ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from n underlying relations.

In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

```
FEMALE_EMPS  $\leftarrow$   $\sigma_{Sex='F'}(EMPLOYEE)$ 
EMP_NAMES  $\leftarrow$   $\pi_{Fname, Lname, Ssn}(FEMALE_EMPS)$ 
EMP_DEPENDENTS  $\leftarrow$  EMP_NAMES  $\times$  DEPENDENT
ACTUAL_DEPENDENTS  $\leftarrow$   $\sigma_{Ssn=Essn}(EMP_DEPENDENTS)$ 
RESULT  $\leftarrow$   $\pi_{Fname, Lname, Dependent\_name}(ACTUAL_DEPENDENTS)$ 
```

The resulting relations from this sequence of operations are shown in next Figure. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMP_NAMES from Figure with DEPENDENT from Figure. In EMP_DEPENDENTS, every tuple from EMP_NAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful (every dependent is combined with every female employee). We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value match the Ssn value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. It is the responsibility of the user to make sure to apply only meaningful operations to relations.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

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

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT related tuples only from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables. Alternatively, if there are two tables in the WHERE clause and there is no corresponding join condition in the query, the result will also be the CARTESIAN PRODUCT of the two tables.

iii. **Binary Relational Operations: JOIN and DIVISION**

- **The JOIN Operation**

The **JOIN** operation, denoted by \bowtie , is used to combine related tuples from two relations into single “longer” tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations. 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 operation and then projecting the result over the necessary attributes, as follows:

$$\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE}$$

$$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT_MGR})$$

The first operation is illustrated in the next Figure.

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	...

Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$\text{EMP_DEPENDENTS} \leftarrow \text{EMP_NAMES} \times \text{DEPENDENT}$$

$$\text{ACTUAL_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP_DEPENDENTS})$$

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

$$\text{ACTUAL_DEPENDENTS} \leftarrow \text{EMP_NAMES} \bowtie_{\text{Ssn}=\text{Essn}} \text{DEPENDENT}$$

The general form of a JOIN operation on two relations R(A1, A2, ..., An) and S(B1, B2, ..., Bm) is

$$R \bowtie_{\langle \text{join condition} \rangle} S$$

The result of the JOIN is a relation Q with n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm) in that order; Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition. This is the main difference between CARTESIAN PRODUCT and JOIN. 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. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q as a single combined tuple.

A general join condition is of the form

<condition> AND <condition> AND...AND <condition>

where each <condition> is of the form $A_i \theta B_j$, A_i is an attribute of R , B_j is an attribute of S , A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a THETA JOIN. Tuples whose join attributes are NULL or for which the join condition is FALSE do not appear in the result. In that sense, the JOIN operation does not necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

- **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 the above Figure, 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:

$PROJ_DEPT \leftarrow PROJECT * \rho_{(Dname, Dnum, Mgr_ssn, Mgr_start_date)}(DEPARTMENT)$

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

$DEPT \leftarrow \rho_{(Dname, Dnum, Mgr_ssn, Mgr_start_date)}(DEPARTMENT)$
 $PROJ_DEPT \leftarrow PROJECT * DEPT$

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. The resulting relation is illustrated in Figure (a). In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but only one join attribute value is kept.

If the attributes on which the natural join is specified already have the same names in both relations, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

$DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS$

The resulting relation is shown in Figure (b), which combines each department with its locations and has one tuple for each location. 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. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

(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

A more general, but nonstandard definition for NATURAL JOIN is

$$Q \leftarrow R *_{\langle \text{list1} \rangle, \langle \text{list2} \rangle} S$$

In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S. The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R—<list1>— is kept in the result Q.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has nR tuples and S has nS tuples, the result of a JOIN operation $R \bowtie_{\langle \text{join condition} \rangle} S$ will have between zero and nR * nS tuples. The expected size of the join result divided by the maximum size nR * nS leads to a ratio called **join selectivity**, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

As we can see, 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**, to distinguish them from a different join variation called outer joins. Informally, an inner join is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself. 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:

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

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

- **A Complete Set of Relational Algebra Operations**

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ 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:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{\langle \text{condition} \rangle} S \equiv \sigma_{\langle \text{condition} \rangle}(R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also not strictly necessary for the expressive power of the relational algebra. However, they are important to include as separate operations because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the basic relational algebra for convenience rather than necessity.

DBMS Lecture Notes (18MCA31)

MODULE 3

Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints.

i. Overview of the SQL Query Language

The SQL language has several parts:

- **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language (DML).** The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition.** The SQL DDL includes commands for defining views.
- **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

ii. SQL Data Definition

The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

a. Basic Types

The SQL standard supports a variety of built-in types, including:

- **char(n):** A fixed-length character string with user-specified length n. The full form, character, can be used instead.

- **varchar(n)**: A variable-length character string with user-specified maximum length n. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric(p,d)**: A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, numeric (3, 1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float(n)**: A floating-point number, with precision of at least n digits.

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered.

The **char** data type stores fixed length strings. Consider, for example, an attribute A of type **char(10)**. If we store a string “Avi” in this attribute, 7 spaces are appended to the string to make it 10 characters long. In contrast, if attribute B were of type **varchar(10)**, and we store “Avi” in attribute B, no spaces would be added. When comparing two values of type **char**, if they are of different lengths extra spaces are automatically added to the shorter one to make them the same size, before comparison.

When comparing a **char** type with a **varchar** type, one may expect extra spaces to be added to the **varchar** type to make the lengths equal, before comparison; however, this may or may not be done, depending on the database system. As a result, even if the same value “Avi” is stored in the attributes A and B above, a comparison A=B may return false.

b. Basic Schema Definition

We define an SQL relation by using **the create table** command. The following command creates a relation department in the database.

```
Create table department
(dept_name varchar (20),
building varchar (15),
budget numeric (12,2),
primary key (dept_name));
```

The relation created above has three attributes, dept_name, which is a character string of maximum length 20, building, which is a character string of maximum length 15, and budget, which is a number with 12 digits in total, 2 of which are after the decimal point. The create table command also specifies that the dept_name attribute is the primary key of the department relation.

The semicolon shown at the end of the create table statements, as well as at the end of other SQL statements later in this chapter, is optional in many SQL implementations.

SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

- **primary key** ($A_{j1}, A_{j2}, \dots, A_{jm}$): The **primary-key** specification says that attributes $A_{j1}, A_{j2}, \dots, A_{jm}$ form the primary key for the relation. The primary key attributes are required to be non-null and unique; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

- **foreign key** ($A_{k1}, A_{k2}, \dots, A_{kn}$) **references** s: The **foreign key** specification says that the values of attributes ($A_{k1}, A_{k2}, \dots, A_{kn}$) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relations.

Figure 3.1 presents a partial SQL DDL definition of the university database we use.

```

Create table department
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2),
primary key(dept_name));

create table course
(course_id varchar(7),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0),
primary key(course_id),
foreign key(dept_name) references department);

create table instructor
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2),
primary key(ID),
foreign key(dept_name) references department);

create table section
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
building varchar(15),
room number varchar(7),
time_slot_id varchar(4),
primary key(course_id, sec_id, semester, year),
foreign key(course_id) references course);

create table teaches
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key(ID, course_id, sec_id, semester, year),
foreign key(course_id, sec_id, semester, year) references section,
foreign key(ID) references instructor);

```

Figure 3.1 SQL data definition for part of the university database.

The definition of the course table has a declaration “**foreign key** (dept_name) **references** department”. This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the primary key attribute (dept_name) of the department relation. Without this constraint, it is possible for a course to specify a non-existent department name. Figure 3.1 also shows foreign key constraints on tables section, instructor and teaches.

- **not null:** The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute. For example, in Figure 3.1, the **not null** constraint on the name attribute of the instructor relation ensures that the name of an instructor cannot be null.

SQL prevents any update to the database that violates an integrity constraint. For example, if a newly inserted or modified tuple in a relation has null values for any primary-key attribute, or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevents the update. Similarly, an insertion of a course tuple with a dept_name value that does not appear in the department relation would violate the foreign-key constraint on course, and SQL prevents such an insertion from taking place.

A newly created relation is empty initially. We can use the insert command to load data into the relation. For example, if we wish to insert the fact that there is an instructor named Smith in the Biology department with instructor_id 10211 and a salary of \$66,000, we write:

```
insert into instructor
values (10211, 'Smith', 'Biology', 66000);
```

The values are specified in the order in which the corresponding attributes are listed in the relation schema.

We can use the **delete** command to delete tuples from a relation. The command

```
delete from student;
```

would delete all tuples from the student relation. Other forms of the delete command allow specific tuples to be deleted.

To remove a relation from an SQL database, we use the **drop table** command. The drop table command deletes all information about the dropped relation from the database. The command

```
drop table r;
```

is a more drastic action than

```
delete from r;
```

The latter retains relation r, but deletes all tuples in r. The former deletes not only all tuples of r, but also the schema for r. After r is dropped, no tuples can be inserted into r unless it is re-created with the create table command.

We use the **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned null as the value for the new attribute. The form of the alter table command is

```
alter table r add A D;
```

where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute. We can drop attributes from a relation by the command

```
alter table r drop A;
```

where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation.

iii. Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**. The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.

a. Queries on a Single Relation

Let us consider a simple query using our university example, “Find the names of all instructors.” Instructor names are found in the instructor relation, so we put that relation in the **from** clause. The instructor’s name appears in the name attribute, so we put that in the **select** clause.

```
Select name from instructor;
```

The result is a relation consisting of a single attribute with the heading name.

Now consider another query, “Find the department names of all instructors,” which can be written as:

```
Select dept_name from instructor;
```

Since more than one instructor can belong to a department, a department name could appear more than once in the instructor relation. The result of the above query is a relation containing the department names, shown in Figure 3.3.

<i>dept_name</i>
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of “select *dept_name* from *instructor*”.

In the formal, mathematical definition of the relational model, a relation is a set. Thus, duplicate tuples would never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL allows duplicates in relations as well as in the results of SQL expressions. Thus, the preceding SQL query lists each department name once for every tuple in which it appears in the instructor relation.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as:

```
Select distinct dept_name from instructor;
```

If we want duplicates removed. The result of the above query would contain each department name at most once.

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

```
Select all dept_name from instructor;
```

The **select** clause may also contain arithmetic expressions involving the operators **+**, **-**, *****, and **/** operating on constants or attributes of tuples. For example, the query:

```
select ID, name, dept_name, salary *1.1 from instructor;
```

returns a relation that is the same as the instructor relation, except that the attribute salary is multiplied by 1.1. This shows what would result if we gave a 10% raise to each instructor; note, however, that it does not result in any change to the instructor relation.

SQL also provides special data types, such as various forms of the date type, and allows several arithmetic functions to operate on these types.

The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate. Consider the query “Find the names of all instructors in the Computer Science department who have salary greater than \$70,000.” This query can be written in SQL as:

```
Select name from instructor where dept_name = 'Comp. Sci.' and salary > 70000;
```

SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators **<**, **<=**, **>**, **>=**, **=**, and **<>**. SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

b. Queries on Multiple Relations

Queries often need to access information from multiple relations. We now study how to write such queries.

An example, suppose we want to answer the query “Retrieve the names of all instructors, along with their department names and department building name.”

Looking at the schema of the relation instructor, we realize that we can get the department name from the attribute dept_name, but the department building name is present in the attribute building of the relation department. To answer the query, each tuple in the instructor relation must be

matched with the tuple in the department relation whose dept_name value matches the dept_name value of the instructor tuple.

In SQL, to answer the above query, we list the relations that need to be accessed in the from clause, and specify the matching condition in the where clause. The above query can be written in SQL as

```
Select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name=department.dept_name;
```

The result of this query is shown in Figure 3.5.

<i>name</i>	<i>dept_name</i>	<i>building</i>
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.5 The result of “Retrieve the names of all instructors, along with their department names and department building name.”

Note that the attribute dept_name occurs in both the relations instructor and department, and the relation name is used as a prefix (in instructor.dept_name, and department.dept name) to make clear to which attribute we are referring. In contrast, the attributes name and building appear in only one of the relations, and therefore do not need to be prefixed by the relation name.

This naming convention requires that the relations that are present in the from clause have distinct names. This requirement causes problems in some cases, such as when information from two different tuples in the same relation needs to be combined.

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the select clause, the from clause, and the where clause. The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form

```
select A1, A2, ..., An
```

```
from r1, r2, ..., rm
where P;
```

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**.

Although the clauses must be written in the order **select, from, where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first from, then where, and then select.

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of set theory, but is perhaps best understood as an iterative process that generates tuples for the result relation of the **from** clause.

```
For each tuple t1 in relation r1
  For each tuple t2 in relation r2
    ...
    For each tuple tm in relation rm
      Concatenate t1, t2, ..., tm into a single tuple t
      Add t into the result relation
```

The result relation has all attributes from all the relations in the from clause. Since the same attribute name may appear in both r_i and r_j , as we saw earlier, we prefix the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations instructor and teaches is:

```
(instructor_ID, instructor.name, instructor.dept_name, instructor.salary, teaches.ID,
  teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)
```

With this schema, we can distinguish instructor.ID from teaches.ID. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema as:

```
(instructor.ID, name, dept_name, salary
  teaches.ID, courseid, secid, semester, year)
```

Consider the instructor relation and the teaches relation. Their Cartesian product is shown in Figure 3.6, which includes only a portion of the tuples that make up the Cartesian product result.

inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...
...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...
...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...
...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...
...

Figure 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

The Cartesian product by itself combines tuples from *instructor* and *teaches* that are unrelated to each other. Each tuple in *instructor* is combined with every tuple in *teaches*, even those that refer to a different instructor. The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.

Instead, the predicate in the *where* clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer. We would expect a query involving *instructor* and *teaches* to combine a particular tuple *t* in *instructor* with only those tuples in *teaches* that refer to the same instructor to which *t* refers. That is, we wish only to match *teaches* tuples with *instructor* tuples that have the same ID value. The following SQL query ensures this condition, and outputs the instructor name and course identifiers from such matching tuples.

```
Select name, course_id
from instructor, teaches
where instructor.ID=teaches.ID;
```

Note that the above query outputs only instructors who have taught some course. Instructors who have not taught any course are not output; if we wish to output such tuples, we could use an operation called the **outer join**.

The relation that results from the preceding query is shown in Figure 3.7. Observe that instructors Gold, Califiori, and Singh, who have not taught any course, do not appear in the above result.

<i>name</i>	<i>course_id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.7 Result of “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”

If we only wished to find instructor names and course identifiers for instructors in the ComputerScience department, we could add an extra predicate to the where clause, as shown below.

```
Select name, courseid
From instructor, teaches
Where instructor.ID=teaches.ID and instructor.dept_name= 'Comp. Sci.';
```

Note that since the dept_name attribute occurs only in the instructor relation, we could have used just dept_name, instead of instructor.dept_name in the above query.

c. The Natural Join

In our example query that combined information from the instructor and teaches table, the matching condition required instructor. ID to be equal to teaches.ID. These are the only attributes in the two relations that have the same name. In fact this is a common case; that is, the matching condition in the from clause most often requires all attributes with matching names to be equated.

To make the life of an SQL programmer easier for this common case, SQL supports an operation called the natural join, which we describe below. We have already seen how a Cartesian product along with a where clause predicate can be used to join information from multiple relations.

The natural join operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations. So, going back to the example of the relations *instructor* and *teaches*, computing ***instructor natural join teaches*** considers only those pairs of tuples where both the tuple from *instructor* and the tuple from *teaches* have the same value on the common attribute, ID.

The result is shown in the Figure 3.8.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.8 The natural join of the *instructor* relation with the *teaches* relation.

The result relation has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

Consider the query “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught” which we wrote earlier as:

```
Select name, courseid
From instructor, teaches
Where instructor.ID=teaches.ID;
```

This query can be written more concisely using the natural-join operation in SQL as:

```
Select name, courseid
from instructor natural join teaches;
```

Both of the above queries generate the same result.

iv. Additional Basic Operations

There are number of additional basic operations that are supported in SQL.

a. The Rename Operation

Consider again the query that we used earlier:

```
Select name, courseid
from instructor, teaches
where instructor.ID=teaches.ID;
```

The result of this query is a relation with the following attributes:

```
name, courseid
```

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We cannot, however, always derive names in this way, for several reasons: First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result. Second, if we used an arithmetic expression in the select clause, the resultant attribute does not have a name. Third, even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result. Hence, SQL provides a way of renaming the attributes of a result relation. It uses the as clause, taking the form:

```
old-name as new-name
```

The as clause can appear in both the **select** and **from** clauses.

For example, if we want the attribute name **name** to be replaced with the name `instructor_name`, we can rewrite the preceding query as:

```
select name as instructor name, courseid
from instructor,teaches
where instructor.ID=teaches.ID;
```

The **as** clause is particularly useful in renaming relations. One reason to rename a relation is to replace along relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”

```
select T.name, S.courseid
from instructor as T, teaches as S
where T.ID=S.ID;
```

b. String Operations

SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. A single quote character that is part of a string can be specified by using two single quote characters; for example, the string "It's right" can be specified by "It''s right".

SQL also permits a variety of functions on character strings, such as concatenating (using "||"), extracting substrings, finding the length of strings, converting strings to uppercase (using the function upper(s) where s is a string) and lowercase (using the function lower(s)), removing spaces at the end of the string (using trim(s)) and so on. There are variations on the exact set of string functions supported by different database systems. See your database system's manual for more details on exactly what string functions it supports.

Pattern matching can be performed on strings, using the operator like. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore (_): The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. To Computer Science', and 'Computational Biology'.
- '___' matches any string of exactly three characters.
- '___%' matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

```
Select dept_name
From department
Where building like '%Watson%';
```

For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a like comparison using the escape keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- like 'ab\%cd%' escape '\' matches all strings beginning with "ab%cd".
- like 'ab\\cd%' escape '\' matches all strings beginning with "ab\cd".

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator.

c. Attribute Specification in Select Clause

The asterisk symbol " * " can be used in the **select** clause to denote "all attributes." Thus, the use of instructor.* in the **select** clause of the query:


```
Select instructor.*
From instructor, teaches
Where instructor.ID=teaches.ID;
```

indicates that all attributes of instructor are to be selected. A **select** clause of the form **select *** indicates that all attributes of the result relation of the **from** clause are selected.

d. Ordering the Display of Tuples

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire instructor relation in descending order of salary. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

```
select *
from instructor
order by salary desc, name asc;
```

e. Where Clause Predicates

SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the names of instructors with salary amounts between \$90,000 and \$100,000, we can use the **between** comparison to write:

```
select name
from instructor
where salary between 90000 and 100000;
```

instead of:

```
select name
from instructor
where salary <= 100000 and salary >= 90000;
```

Similarly, we can use the **not between** comparison operator.

We can extend the preceding query that finds instructor names along with course identifiers, which we saw earlier, and consider a more complicated case in which we require also that the instructors be from the Biology department: "Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course." To write this query, we can

modify either of the SQL queries we saw earlier, by adding an extra condition in the where clause. We show below the modified form of the SQL query that does not use natural join.

```
select name, course_id
from instructor, teaches
where instructor.ID=teaches.ID and dept_name= 'Biology';
```

v. Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations \cup , \cap , and $-$. We shall now construct queries involving the union, intersect, and except operations over two sets.

- The set of all courses taught in the Fall 2009 semester:

```
select course_id
from section
where semester = 'Fall' and year= 2009;
```

- The set of all courses taught in the Spring2010 semester:

```
select course_id
from section
where semester = 'Spring' and year= 2010;
```

We shall refer to the relations obtained as the result of the preceding queries as c_1 and c_2 , respectively, and show the results when these queries are run on the section relation.

a. The Union Operation

To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both, we write:

```
(select course_id
from section
where semester = 'Fall' and year= 2009)
union
(select course_id
from section
where semester = 'Spring' and year = 2010);
```

The union operation automatically eliminates duplicates, unlike the select clause. If we want to retain all duplicates, we must write **union all** in place of **union**.

b. The Intersect Operation

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

```
(select course_id
from section
where semester = 'Fall' and year= 2009)
intersect
(select course_id
```

from section
where semester = 'Spring' and year= 2010);

The intersect operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of **intersect**.

c. The Except Operation

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

```
(select course_id
from section
where semester = 'Fall' and year= 2009)
except
(select course_id
from section
where semester = 'Spring' and year= 2010);
```

The except operation outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. If we want to retain duplicates, we must write **except all** in place of **except**.

vi. Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example +, -, *, or /) is null if any of the input values is null. For example, if a query has an expression $r.A+5$, and $r.A$ is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison " $1 < \text{null}$ ". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, " $\text{not}(1 < \text{null})$ " would evaluate to true, which does not make sense. SQL therefore treats as unknown the result of any comparison involving a null value. This creates a third logical value in addition to true and false.

Since the predicate in a where clause can involve Boolean operations such as and, or, and not on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value unknown.

- **and**: The result of true and unknown is unknown, false and unknown is false, while unknown and unknown is unknown.
- **or**: The result of true or unknown is true, false or unknown is unknown, while unknown or unknown is unknown.
- **not**: The result of not unknown is unknown.

You can verify that if $r.A$ is null, then " $1 < r.A$ " as well as "**not** ($1 < r.A$)" evaluate to unknown.

If the where clause predicate evaluates to either false or unknown for a tuple, that tuple is not added to the result.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the instructor relation with null values for salary, we write:

```
select name
from instructor
where salary is null;
```

The predicate **is not null** succeeds if the value on which it is applied is not null.

Some implementations of SQL also allow us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses **is unknown** and **is not unknown**.

When a query uses the **select distinct** clause, duplicate tuples must be eliminated. For this purpose, when comparing values of corresponding attributes from two tuples, the values are treated as identical if either both are non-null and equal in value, or both are null. Thus two copies of a tuple, such as {(‘A’,null), (‘A’,null)}, are treated as being identical, even if some of the attributes have a null value. Using the distinct clause then retains only one copy of such identical tuples. Note that the treatment of null above is different from the way nulls are treated in predicates, where a comparison “null=null” would return unknown, rather than true.

The above approach of treating tuples as identical if they have the same values for all attributes, even if some of the values are null, is also used for the set operations union, intersection and except.

vii. Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

a. Basic Aggregation

Consider the query “Find the average salary of instructors in the Computer Science department.” We write this query as follows:

```
select avg (salary)
from instructor
where dept_name= 'Comp.Sci.';
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an arbitrary name to the result relation attribute that is generated by aggregation; however, we can give a meaningful name to the attribute by using the as clause as follows:

```
select avg (salary) as avg_salary
from instructor
where dept_name= 'Comp.Sci.';
```

In the instructor relation, the salaries in the Computer Science department are \$75,000, \$65,000, and \$92,000. The average balance is $\$232,000/3 = \$77,333.33$.

Retaining duplicates is important in computing an average. Suppose the Computer Science department adds a fourth instructor whose salary happens to be \$75,000. If duplicates were eliminated, we would obtain the wrong answer ($\$232,000/4 = \$58,000$) rather than the correct answer of \$76,750.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2010 semester." In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation teaches, and we write this query as follows:

```
select count(distinct ID)
from teaches
where semester = 'Spring' and year= 2010;
```

Because of the keyword **distinct** preceding ID, even if an instructor teaches more than one course, she is counted only once in the result.

We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is count (*). Thus, to find the number of tuples in the course relation, we write

```
select count(*)
from course;
```

b. Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

```
Select dept_name, avg(salary) as avg_salary
From instructor
Group by dept_name;
```

Figure 3.14 shows the tuples in the instructor relation grouped by the dept_name attribute, which is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 3.15.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 3.14 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.15 The result relation for the query “Find the average salary in each department”.

In contrast, consider the query “Find the average salary of all instructors.” We write this query as follows:

```
Select avg (salary)
From instructor;
```

In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.

c. The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than \$42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the having clause of SQL. SQL applies predicates in the having clause after groups have been formed, so aggregate functions may be used. We express this query in SQL as follows:

```
select dept_name, avg(salary) as avg_salary
from instructor
```

```
group by dept_name
having avg(salary)> 42000;
```

The result is shown in Figure 3.17.

<i>dept_name</i>	<i>avg(avg_salary)</i>
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.17 The result relation for the query “Find the average salary of instructors in those departments where the average salary is more than \$42,000.”

As was the case for the select clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.
4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

d. Aggregation with Null and Boolean Values

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the instructor relation have a null value for salary. Consider the following query to total all salary amounts:

```
select sum (salary)
from instructor;
```

The values to be summed in the preceding query include null values, since some tuples have a null value for salary. Rather than say that the overall sum is itself null, the SQL standard says that the **sum** operator should ignore null values in its input.

In general, aggregate functions treat nulls according to the following rule: All aggregate functions except **count (*)** ignore null values in their input collection. As a result of null values being ignored,

the collection of values may be empty. The count of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection. The effect of null values on some of the more complicated SQL constructs can be subtle.

viii. Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause.

a. Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a select clause. The **not in** connective tests for the absence of set membership.

As an illustration, reconsider the query “Find all the courses taught in the both the Fall 2009 and Spring2010 semesters.” Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2009 and the set of courses taught in Spring 2010. We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring2010. Clearly, this formulation generates the same results as the previous one did, but it leads us to write our query using the in connective of SQL. We begin by finding all courses taught in Spring2010, and we write the subquery

```
(select course_id
  From section
  Where semester = 'Spring' and year= 2010)
```

We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the where clause of an outer query. The resulting query is

```
Select distinct course_id
  From section
  Where semester = 'Fall' and year= 2009 and
        course_id in (select course_id
                      From section
                      Where semester = 'Spring' and year= 2010);
```

This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way that seems most natural. We shall see that there is a substantial amount of redundancy in SQL.

We use the **not in** construct in a way similar to the **in** construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_id
  from section
  where semester= 'Fall' and year= 2009 and
        course_id not in (select course_id
                          from section where semester = 'Spring' and year= 2010);
```


b. Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query “Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.” Before, we wrote this query as follows:

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase “greater than at least one” is represented in SQL by **> some**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
Select name
From instructor
Where salary > some (select salary
                    From instructor
                    Where dept_name = 'Biology');
```

The subquery:

```
(select salary
 From instructor
 Where dept_name = 'Biology')
```

generates the set of all salary values of all instructors in the Biology department. The **> some** comparison in the **where** clause of the outer **select** is true if the salary value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

Now we modify our query slightly. Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct **> all** corresponds to the phrase “greater than all.” Using this construct, we write the query as follows:

```
Select name
From instructor
Where salary > all (select salary
                  From instructor
                  Where dept_name = 'Biology');
```

SQL also allows **< all**, **<= all**, **>= all**, **= all**, and **<> all** comparisons.

c. Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value true if the argument subquery is nonempty. Using the **exists** construct, we can write the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester” in still another way:

```
Select course_id
```

```

From section as S
Where semester = 'Fall and year = 2009 and
  Exists (select *
    From section as T
    Where semester = 'Spring' and year = 2010 and
      S.course_id = T.course_id);

```

The above query also illustrates a feature of SQL where a correlation name from an outer query (S in the above query), can be used in a subquery in the where clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. We can use the **not exists** construct to simulate the set containment (that is, superset) operation: We can write "relation A contains relation B" as "not exists (B except A)." To illustrate the **not exists** operator, consider the query "Find all students who have taken all courses offered in the Biology department." Using the **except** construct, we can write the query as follows:

```

Select distinct S.ID, S.name
From student as S
Where not exists ((select course_id
  From course
  Where dept_name = 'Biology')
  except
  (select T.course_id
  From takes as T
  Where S.ID=T.ID));

```

Here, the subquery:

```

(select course_id
From course
Where dept_name = 'Biology')

```

finds the set of all courses offered in the Biology department. The subquery:

```

(select T.course_id
From takes as T
Where S.ID=T.ID)

```

finds all the courses that student.ID has taken. Thus, the outer **select** takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

d. Test for the Absence of Duplicate Tuples

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query "Find all courses that were offered at most once in 2009" as follows:

```

Select T.course_id

```

```

From course as T
Where unique (select R.course_id
              From section as R
              Where T.course_id = R.course_id and
              R.year = 2009);

```

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

An equivalent version of the above query not using the **unique** construct is:

```

Select T.course_id
From course as T
where 1 <= (select count(R.course_id)
           from section as R
           where T.course_id = R.course_id and
           R.year = 2009);

```

We can test for the existence of duplicate tuples in a subquery by using the not unique construct. To illustrate this construct, consider the query “Find all courses that were offered at least twice in 2009” as follows:

```

Select T.course_id
From course as T
Where not unique (select R.course_id
                  From section as R
                  Where T.course_id = R.course_id and
                  R.year = 2009);

```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two tuples t_1 and t_2 such that $t_1 = t_2$. Since the test $t_1 = t_2$ fails if any of the fields of t_1 or t_2 are null, it is possible for unique to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

e. Subqueries in the From Clause

SQL allows a subquery expression to be used in the from clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.

Consider the query “Find the average instructors’ salaries of those departments where the average salary is greater than \$42,000.” We wrote this query before by using the having clause. We can now rewrite this query, without using the having clause, by using a subquery in the from clause, as follows:

```

Select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
     from instructor
     group by dept_name)
where avg_salary > 42000;

```

The subquery generates a relation consisting of the names of all departments and their corresponding average instructors' salaries. The attributes of the subquery result can be used in the outer query, as can be seen in the above example.

Note that we do not need to use the having clause, since the subquery in the from clause computes the average salary, and the predicate that was in the having clause earlier is now in the where clause of the outer query.

We can give the subquery result relation a name, and rename the attributes, using the as clause, as illustrated below.

```
Select dept_name, avg_salary
From (select dept_name, avg (salary)
      From instructor
      Group by dept_name)
As dept_avg (dept_name, avg_salary)
Where avg_salary>42000;
```

The subquery result relation is named dept_avg, with the attributes dept_name and avg_salary.

Nested subqueries in the **from** clause are supported by most but not all SQL implementations. However, some SQL implementations, notably Oracle, do not support renaming of the result relation in the **from** clause.

As another example, suppose we wish to find the maximum across all departments of the total salary at each department. The **having** clause does not help us in this task, but we can write this query easily by using a subquery in the **from** clause, as follows:

```
Select max(tot_salary)
From (select dept_name, sum(salary)
      From instructor
      Group by dept_name) as dept_total (dept_name,tot_salary);
```

f. The with clause

The with clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget(value) as
  (select max(budget)
   From department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

The **with** clause defines the temporary relation max_budget, which is used in the immediately following query.

We could have written the above query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and

understand. The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.

For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments. We can write the query using the **with** clause as follows.

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     From instructor
     Group by dept_name),
Dept_total_avg(value) as
    (select avg(value)
     From dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

We can create an equivalent query without the **with** clause, but it would be more complicated and harder to understand.

g. Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called scalar subqueries. For example, a subquery can be used in the **select** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
Select dept_name,
    (select count(*)
     From instructor
     Where department.dept_name = instructor.dept_name)
As num_instructors
From department;
```

The subquery in the above example is guaranteed to return only a single value since it has a **count(*)** aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the from clause of the outer query, such as `department.dept_name` in the above example.

Scalar subqueries can occur in **select**, **where**, and **having** clauses. Scalar subqueries may also be defined without aggregates. It is not always possible to figure out at compile time if a subquery can return more than one tuple in its result; if the result has more than one tuple when the subquery is executed, a run-time error occurs.

Note that technically the type of a scalar subquery result is still a relation, even if it contains a single tuple. However, when a scalar subquery is used in an expression where a value is expected, SQL implicitly extracts the value from the single attribute of the single tuple in the relation, and returns that value.

ix. Modification of the Database

Let us see how to add, remove, or change information with SQL.

a. Deletion

A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

```
Delete from r
Where P;
```

where P represents a predicate and r represents a relation. The **delete** statement first finds all tuples t in r for which P(t) is true, and then deletes them from r. The **where** clause can be omitted, in which case all tuples in r are deleted.

Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request

```
Delete from instructor;
```

deletes all tuples from the instructor relation. The instructor relation itself still exists, but it is empty. Here are examples of SQL delete requests:

- Delete all tuples in the instructor relation pertaining to instructors in the Finance department.

```
Delete from instructor
Where dept_name = 'Finance';
```

- Delete all instructors with a salary between \$13,000 and \$15,000.

```
Delete from instructor
Where salary between 13000 and 15000;
```

- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.

```
Delete from instructor
Where dept_name in (select dept_name
                    From department
                    where building= 'Watson');
```

This delete request first finds all departments located in Watson, and then deletes all instructor tuples pertaining to those departments.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a select-from-where nested in the where clause of a delete. The delete request can contain a nested select that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all instructors with salary below the average

at the university. We could write:

```
delete from instructor
where salary < (select avg (salary)
               from instructor);
```

The delete statement first tests each tuple in the relation instructor to check whether the salary is less than the average salary of instructors in the university. Then, all tuples that fail the test—that is, represent an instructor with a lower-than-average salary—are deleted. Performing all the tests before performing any deletion is important—if some tuples are deleted before other tuples have been tested, the average salary may change, and the final result of the delete would depend on the order in which the tuples were processed!

b. Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes.

The simplest insert statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title "Database Systems", and 4 credit hours. We write:

```
Insert into course
values ('CS-437','Database Systems','Comp.Sci.',4);
```

In this example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the insert statement. For example, the following SQL insert statements are identical in function to the preceding one:

```
Insert into course (course id,title,dept name,credits)
values ('CS-437','Database Systems','Comp.Sci.',4);

insert into course (title,courseid,credits,dept name)
values ('Database Systems','CS-437', 4,'Comp.Sci.');
```

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of \$18,000. We write:

```
Insert into instructor
select ID,name,dept_name, 18000
from student
where dept_name = 'Music' and tot_cred>144;
```

Instead of specifying a tuple as we did earlier in this section, we use a select to specify a set of tuples. SQL evaluates the select statement first, giving a set of tuples that is then inserted into the instructor relation. Each tuple has an ID, a name, a dept_name (Music), and a salary of \$18,000.

It is important that we evaluate the select statement fully before we carry out any insertions. If we carry out some insertions even as the select statement is being evaluated, a request such as:

```
insert into student
  select * from student;
```

might insert an infinite number of tuples, if the primary key constraint on student were absent. Without the primary key constraint, the request would insert the first tuple in student again, creating a second copy of the tuple. Since this second copy is part of student now, the select statement may find it, and a third copy would be inserted into student. The select statement may then find this third copy and insert a fourth copy, and soon, forever. Evaluating the select statement completely before performing insertions avoids such problems. Thus, the above insert statement would simply duplicate every tuple in the student relation, if the relation did not have a primary key constraint.

Our discussion of the insert statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by null. Consider the request:

```
insert into student
  values ('3003', 'Green', 'Finance', null);
```

The tuple inserted by this request specified that a student with ID "3003" is in the Finance department, but the tot_cred value for this student is not known. Consider the query:

```
select student
  from student
  Where tot_cred > 45;
```

Since the tot_cred value of student "3003" is not known, we cannot determine whether it is greater than 45.

Most relational database products have special "bulk loader" utilities to insert a large set of tuples into a relation. These utilities allow data to be read from formatted text files, and can execute much faster than an equivalent sequence of insert statements.

c. Updates

In certain situations, we may wish to change a value in a tuple without changing all values in the tuple. For this purpose, the **update** statement can be used. As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.

Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent. We write:

```
update instructor
  set salary = salary * 1.05;
```

The preceding update statement is applied once to each of the tuples in instructor relation.

If a salary increase is to be paid only to instructors with salary of less than \$70,000, we can write:


```

update instructor
set salary = salary *1.05
where salary < 70000;

```

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **selects**). As with **insert** and **delete**, a nested select within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and carries out the updates afterward. For example, we can write the request “Give a 5 percent salary raise to instructors whose salary is less than average” as follows:

```

update instructor
set salary = salary *1.05
where salary < (select avg (salary)
                from instructor);

```

Let us now suppose that all instructors with salary over \$100,000 receive a 3 percent raise, whereas all others receive a 5 percent raise. We could write two **update** statements:

```

update instructor
set salary = salary *1.03 where salary > 100000;

update instructor
set salary = salary *1.05 where salary <= 100000;

```

Note that the order of the two **update** statements is important. If we changed the order of the two statements, an instructor with a salary just under \$100,000 would receive an over 8 percent raise.

SQL provides a **case** construct that we can use to perform both the updates with a single **update** statement, avoiding the problem with the order of updates.

```

update instructor
set salary = case
                When salary <= 100000 then salary *1.05
                Else salary *1.03
            End

```

The general form of the case statement is as follows.

```

case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end

```

The operation returns $result_i$, where i is the first of $pred_1, pred_2, \dots, pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$. Case statements can be used in any place where a value is expected.

Scalar subqueries are also useful in SQL update statements, where they can be used in the set clause. Consider an update where we set the tot_cred attribute of each student tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is not 'F' or null. To specify this update, we need to use a subquery in the set clause, as shown below:

```
Update student S
set tot_cred = ( select sum(credits)
                from takes natural join course
                where S.ID=takes.ID and
                      takes.grade <> 'F' and
                      takes.grade is not null);
```

Observe that the subquery uses a correlation variable S from the **update** statement. In case a student has not successfully completed any course, the above update statement would set the tot_cred attribute value to null. To set the value to 0 instead, we could use another **update** statement to replace null values by 0; a better alternative is to replace the clause “**select sum(credits)**” in the preceding subquery by the following **select** clause using a **case** expression:

```
select case
        when sum(credits) is not null then sum(credits)
        else 0
        end
```

x. JOIN Expressions

The examples in this section involve the two relations **student** and **takes**, shown in Figures 4.1 and 4.2, respectively. Observe that the attribute **grade** has a value **null** for the student with ID 98988, for the course BIO-301, section 1, taken in Summer 2010. The null value indicates that the grade has not been awarded yet.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.1 The *student* relation.

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	<i>null</i>

Figure 4.2 The *takes* relation.

a. Join Conditions

We saw the **join ... using** clause, which is a form of natural join that only requires values to match on specified attributes. SQL supports another form of join, in which an arbitrary join condition can be specified.

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression.

Consider the following query, which has a join expression containing the **on** condition.

```
select *
from student join takes on student.ID=takes.ID;
```

The **on** condition above specifies that a tuple from student matches a tuple from takes if their ID values are equal. The join expression in this case is almost the same as the join expression student **natural join** takes, since the natural join operation also requires that for a student tuple and a takes tuple to match. The one difference is that the result has the ID attribute listed twice, in the join result, once for student and once for takes, even though their ID values must be the same.

In fact, the above query is equivalent to the following query (in other words, they generate exactly the same results):

```

select *
from student, takes
where student.ID = takes.ID;

```

As we have seen earlier, the relation name is used to disambiguate the attribute name ID, and thus the two occurrences can be referred to as student.ID and takes.ID respectively. A version of this query that displays the ID value only once is as follows:

```

select student.ID as ID, name, dept_name, tot_cred,
course_id, sec_id, semester, year, grade
from student join takes on student.ID=takes.ID;

```

The result of the above query is shown in Figure 4.3.

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null

Figure 4.3 The result of *student join takes on student.ID= takes.ID* with second occurrence of ID omitted.

The **on** condition can express any SQL predicate, and thus a join expressions using the **on** condition can express a richer class of join conditions than **natural join**. However, as illustrated by our receding example, a query using a join expression with an **on** condition can be replaced by an equivalent expression without the **on** condition, with the predicate in the **on** clause moved to the **where** clause. Thus, it may appear that the **on** condition is a redundant feature of SQL.

However, there are two good reasons for introducing the **on** condition. First, we shall see shortly that for a kind of join called an outer join, **on** conditions do behave in a manner different from **where** conditions. Second, an SQL query is often more readable by humans if the join condition is specified in the **on** clause and the rest of the conditions appear in the **where** clause.

b. Outer Joins

Suppose we wish to display a list of all students, displaying their ID, and name, dept_name, and tot_cred, along with the courses that they have taken. The following SQL query may appear to retrieve the required information:

```
select *  
from student natural join takes;
```

Unfortunately, the above query does not work quite as intended. Suppose that there is some student who takes no courses. Then the tuple in the student relation for that particular student would not satisfy the condition of a natural join with any tuple in the takes relation, and that student's data would not appear in the result. We would thus not see any information about students who have not taken any courses. For example, in the **student** and **takes** relations of Figures 4.1 and 4.2, note that student Snow, with ID 70557, has not taken any courses. Snow appears in student, but Snow's ID number does not appear in the ID column of takes. Thus, Snow does not appear in the result of the natural join. More generally, some tuples in either or both of the relations being joined may be "lost" in this way. The **outer join** operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.

For example, to ensure that the student named Snow from our earlier example appears in the result, a tuple could be added to the join result with all attributes from the student relation set to the corresponding values for the student Snow, and all the remaining attributes which come from the takes relation, namely course_id, sec_id, semester, and year, set to null. Thus the tuple for the student Snow is preserved in the result of the outer join.

There are in fact three forms of outer join:

- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve nonmatched tuples are called **inner join** operations, to distinguish them from the outer-join operations.

We can compute the left outer-join operation as follows. First, compute the result of the inner join as before. Then, for every tuple *t* in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple *r* to the result of the join constructed as follows:

- The attributes of tuple *r* that are derived from the left-hand-side relation are filled in with the values from tuple *t*.
- The remaining attributes of *r* are filled with null values.

Figure 4.4 shows the result of:

```
select *  
from student natural left outer join takes;
```

That result includes student Snow (ID 70557), unlike the result of an inner join, but the tuple for Snow includes nulls for the attributes that appear only in the schema of the takes relation.

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	<i>null</i>

Figure 4.4 Result of *student* natural left outer join *takes*.

As another example of the use of the outer-join operation, we can write the query “Find all students who have not taken a course” as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

The **right outer join** is symmetric to the **left outer join**. Tuples from the right hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite our above query using a right outer join and swapping the order in which we list the relations as follows:

```
select *
from takes natural right outer join student;
```

we get the same result except for the order in which the attributes appear in the result (see Figure 4.5).

ID	course_id	sec_id	semester	year	grade	name	dept_name	tot_cred
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2009	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2010	B	Brandt	History	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	<i>null</i>	Tanaka	Biology	120

Figure 4.5 The result of *takes* natural right outer join *student*.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand side relation, and adds them to the result. Similarly, it extends with nulls those tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

As an example of the use of full outer join, consider the following query: “Display a list of all students in the Comp.Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section.” This query can be written as:

```

select *
from (select *
      from student
      where dept_name = 'Comp.Sci')
natural full outer join
(select *
 from takes
 where semester = 'Spring' and year = 2009);

```

The **on** clause can be used with outer joins. The following query is identical to the first query we saw using “student **natural left outer join** takes,” except that the attribute ID appears twice in the result.

```

select *
from student left outer join takes on student.ID=takes.ID;

```

As we noted earlier, **on** and **where** behave differently for outer join. The reason for this is that outer join adds null-padded tuples only for those tuples that do not contribute to the result of the corresponding inner join. The **on** condition is part of the outer join specification, but a **where** clause is not. In our example, the case of the student tuple for student “Snow” with ID 70557, illustrates this distinction. Suppose we modify the preceding query by moving the on clause predicate to the **where** clause, and instead using an **on** condition of true.

```

select *
from student left outer join takes on true
where student.ID = takes.ID;

```

The earlier query, using the left outer join with the on condition, includes a tuple (70557, Snow, Physics, 0, null, null, null, null, null), because there is no tuple in takes with ID=70557. In the latter query, however, every tuple satisfies the join condition true, so no null-padded tuples are generated by the outer join. The outer join actually generates the Cartesian product of the two relations. Since there is no tuple in takes with ID=70557, every time a tuple appears in the outer join with name=“Snow”, the values for student.ID and takes.ID must be different, and such tuples would be eliminated by the where clause predicate. Thus student Snow never appears in the result of the latter query.

c. Join Types and Conditions

To distinguish normal joins from outer joins, normal joins are called inner joins in SQL. A join clause can thus specify **inner join** instead of **outer join** to specify that a normal join is to be used. The keyword **inner** is, however, optional. The default join type, when the **join** clause is used without the **outer** prefix is the **inner join**. Thus,

```

select *
from student join takes using (ID);

```

is equivalent to:

```

select *
from student inner join takes using (ID);

```

Similarly, **natural join** is equivalent to **natural inner join**.

Figure 4.6 shows a full list of the various types of join that we have discussed. As can be seen from the figure, any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).

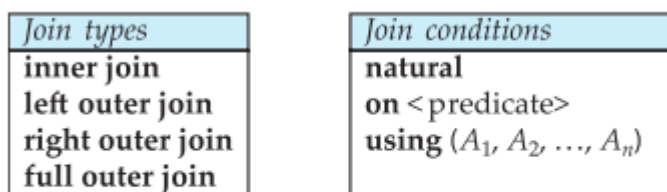


Figure 4.6 Join types and join conditions.

xi. Views

In our examples up to this point, we have operated at the logical-model level. That is, we have assumed that the relations in the collection we are given are the actual relations stored in the database.

It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. Consider a clerk who needs to know an instructor's ID, name and department name, but does not have authorization to see the instructor's salary amount. This person should see a relation described in SQL, by:

```
select ID, name, dept_name
from instructor;
```

Aside from security concerns, we may wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the logical model. We may want to have a list of all course sections offered by the Physics department in the Fall 2009 semester, with the building and room number of each section. The relation that we would create for obtaining such a list is:

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
      and course.deptname = 'Physics'
      and section.semester = 'Fall'
      and section.year = '2009';
```

It is possible to compute and store the results of the above queries and then make the stored relations available to users. However, if we did so, and the underlying data in the relations instructor, course, or section changes, the stored query results would then no longer match the result of reexecuting the query on the relations. In general, it is a bad idea to compute and store query results such as those in the above examples.

Instead, SQL allows a "virtual relation" to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.

Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

a. View Definition

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is:

```
create view v as <queryexpression>;
```

where <query expression> is any legal query expression. The view name is represented by v.

Consider again the clerk who needs to access all data in the instructor relation, except salary. The clerk should not be authorized to access the instructor relation. Instead, a view relation **faculty** can be made available to the clerk, with the view defined as follows:

```
create view faculty as
select ID, name, dept_name
from instructor;
```

The view relation conceptually contains the tuples in the query result, but is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation. Whenever the view relation is accessed, its tuples are created by computing the query result. Thus, the view relation is created whenever needed, on demand.

To create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section, we write:

```
Create view physics_fall_2009 as
Select course.course_id, sec_id, building, room_number
From course, section
Where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall'
and section.year = '2009';
```

b. Using Views in SQL Queries

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view physics fall 2009, we can find all Physics courses offered in the Fall 2009 semester in the Watson building by writing:

```
Select course_id
From physics_fall_2009
Where building = 'Watson';
```

View names may appear in a query any place where a relation name may appear. The attribute names of a view can be specified explicitly as follows:

```
Create view departments_total_salary(dept_name, total_salary) as
Select dept_name, sum (salary)
From instructor
Group by dept_name;
```

The preceding view gives for each department the sum of the salaries of all the instructors at that department. Since the expression `sum(salary)` does not have a name, the attribute name is specified explicitly in the view definition.

Intuitively, at any given time, the set of tuples in the view relation is the result of evaluation of the query expression that defines the view. Thus, if a view relation is computed and stored, it may become out of date if the relations used to define it are modified. To avoid this, views are usually implemented as follows. When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the query expression that defines the view. Wherever a view relation

appears in a query, it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation is recomputed.

One view may be used in the expression defining another view. For example, we can define a view `physics_fall_2009_watson` that lists the course ID and room number of all Physics courses offered in the Fall 2009 semester in the Watson building as follows:

```
create view physics_fall_2009_watson as
  select course_id, room_number
  from physics_fall_2009
  where building = 'Watson';
```

where `physics_fall_2009_watson` is itself a view relation. This is equivalent to:

```
create view physics_fall_2009_watson as
  (select course_id, room_number
   from (select course.course_id, building, room_number
         from course, section
         where course.course_id = section.course_id
              and course.dept_name = 'Physics'
              and section.semester = 'Fall'
              and section.year = '2009'))
  where building = 'Watson';
```

c. Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

For example, consider the view `departments_total_salary`. If the above view is materialized, its results would be stored in the database. However, if an instructor tuple is added to or deleted from the instructor relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated. Similarly, if an instructor's salary is updated, the tuple in `departments_total_salary` corresponding to that instructor's department must be updated.

The process of keeping the materialized view up-to-date is called **materialized view maintenance**. View maintenance can be done immediately when any of the relations on which the view is defined is updated. Some database systems, however, perform view maintenance lazily, when the view is accessed. Some systems update materialized views only periodically; in this case, the contents of the materialized view may be stale, that is, not up-to-date, when it is used, and should not be used if the application needs up-to-date data. And some database systems permit the database administrator to control which of the above methods is used for each materialized view.

Applications that use a view frequently may benefit if the view is materialized. Applications that demand fast response to certain queries that compute aggregates over large relations can also benefit greatly by creating materialized views corresponding to the queries. In this case, the aggregated result is likely to be much smaller than the large relations on which the view is defined; as a result the materialized view can be used to answer the query very quickly, avoiding reading the large underlying relations. Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

SQL does not define a standard way of specifying that a view is materialized, but many database systems provide their own SQL extensions for this task. Some database systems always keep materialized views up-to-date when the underlying relations change, while others permit them to become out of date, and periodically recompute them.

d. Update of a View

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

Suppose the view `faculty`, which we saw earlier, is made available to a clerk. Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

```
Insert into faculty
values ('30765', 'Green', 'Music');
```

This insertion must be represented by an insertion into the relation `instructor`, since `instructor` is the actual relation from which the database system constructs the view `faculty`. However, to insert a tuple into `instructor`, we must have some value for salary. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple ('30765', 'Green', 'Music', null) into the `instructor` relation.

Another problem with modification of the database through views occurs with a view such as:

```
create view instructor_info as
select ID,name,building
from instructor, department
where instructor.dept_name = department.dept_name;
```

This view lists the ID, name, and building-name of each instructor in the university. Consider the following insertion through this view:

```
insert into instructor_info
values ('69987', 'White', 'Taylor');
```

Suppose there is no instructor with ID 69987, and no department in the Taylor building. Then the only possible method of inserting tuples into the `instructor` and `department` relations is to insert ('69987', 'White', null, null) into `instructor` and (null, 'Taylor', null) into `department`. Then, we obtain the relations shown in Figure 4.7. However, this update does not have the desired effect, since the view relation `instructor_info` still does not include the tuple ('69987', 'White', 'Taylor'). Thus, there is no way to update the relations `instructor` and `department` by using nulls to get the desired update on `instructor_info`.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	<i>null</i>	<i>null</i>

instructor

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
<i>null</i>	Painter	<i>null</i>

department

Figure 4.7 Relations *instructor* and *department* after insertion of tuples.

Because of problems such as these, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations; see the database system manuals for details.

In general, an SQL view is said to be updatable (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The from clause has only one database relation.
- The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- Any attribute not listed in the select clause can be set to null; that is, it does not have a not null constraint and is not part of a primary key.
- The query does not have a group by or having clause.

Under these constraints, the update, insert, and delete operations would be allowed on the following view:

```
create view history_instructors as
select *
from instructor
```

where dept_name = 'History';

Even with the conditions on updatability, the following problem still remains. Suppose that a user tries to insert the tuple ('25566', 'Brown', 'Biology', 100000) into the history_instructors view. This tuple can be inserted into the instructor relation, but it would not appear in the history_instructors view since it does not satisfy the selection imposed by the view.

By default, SQL would allow the above update to proceed. However, views can be defined with a **with check option** clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's where clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the where clause conditions.

xii. Transactions

A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

The keyword **work** is optional in both the statements.

Transaction rollback is useful if some error condition is detected during execution of a transaction. Commit is similar, in a sense, to saving changes to a document that is being edited, while rollback is similar to quitting the edit session without saving changes. Once a transaction has executed commit work, its effects can no longer be undone by rollback work. The database system guarantees that in the event of some failure, such as an error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled back if it has not yet executed commit work. In the case of power outage or other system crash, the rollback occurs when the system restarts.

For instance, consider a banking application, where we need to transfer money from one bank account to another in the same bank. To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other. If the system crashes after subtracting the amount from the first account, but before adding it to the second account, the bank balances would be inconsistent. A similar problem would occur, if the second account is credited before subtracting the amount from the first account, and the system crashes just after crediting the amount.

As another example, consider our running example of a university application. We assume that the attribute tot_cred of each tuple in the student relation is kept up-to-date by modifying it whenever the student successfully completes a course. To do so, whenever the takes relation is updated to record successful completion of a course by a student (by assigning an appropriate grade) the corresponding student tuple must also be updated. If the application performing these two updates crashes after one update is performed, but before the second one is performed, the data in the database would be inconsistent.

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being atomic, that is, indivisible. Either all the effects of the transaction are reflected in the database, or none are (after rollback).

Applying the notion of transactions to the above applications, the update statements should be executed as a single transaction. An error while a transaction executes one of its statements would result in undoing of the effects of the earlier statements of the transaction, so that the database is not left in a partially updated state.

If a program terminates without executing either of these commands, the updates are either committed or rolled back. The standard does not specify which of the two happens, and the choice is implementation dependent.

xiii. Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database. Examples of integrity constraints are:

- An instructor name cannot be null.
- No two instructors can have the same instructor ID.
- Every department name in the course relation must have a matching department name in the department relation.
- The budget of a department must be greater than \$0.00.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify integrity constraints that can be tested with minimal overhead.

Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations. However, integrity constraints can also be added to an existing relation by using the command **alter table** table-name **add** constraint, where constraint can be any constraint on the relation. When such a command is executed, the system first ensures that the relation satisfies the specified constraint. If it does, the constraint is added to the relation; if not, the command is rejected.

a. Constraints on a Single Relation

The **create table** command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command. The allowed integrity constraints include

- **not null**
- **unique**
- **check** (<predicate>)

b. Not Null Constraint

The null value is a member of all domains, and as a result is a legal value for every attribute in SQL by default. For certain attributes, however, null values may be inappropriate. Consider a tuple in the

student relation where name is null. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be null. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes name and budget to exclude null values, by declaring it as follows:

```
name varchar(20) not null
budget numeric(12,2) not null
```

The not null specification prohibits the insertion of a null value for the attribute. Any database modification that would cause a null to be inserted in an attribute declared to be not null generates an error diagnostic.

There are many situations where we want to avoid null values. In particular, SQL prohibits null values in the primary key of a relation schema. Thus, in our university example, in the department relation, if the attribute dept_name is declared as the primary key for department, it cannot take a null value. As a result it would not need to be declared explicitly to be not null.

c. Unique Constraint

SQL also supports an integrity constraint:

```
unique (Aj1, Aj2, ..., Ajm)
```

The **unique** specification says that attributes Aj₁, Aj₂, ..., Aj_m form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be not null. Recall that a null value does not equal any other value.

d. The check Clause

When applied to a relation declaration, the clause **check(P)** specifies a predicate P that must be satisfied by every tuple in a relation.

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause **check (budget >0)** in the **create table** command for relation department would ensure that the value of **budget** is nonnegative.

As another example, consider the following:

```
Create table section
(course_id    varchar(8),
Sec_id       varchar(8),
semester     varchar(6),
year         numeric(4,0),
building     varchar(15),
room_number  varchar(7),
time_slot_id varchar(4),
primary key (courseid, secid, semester, year),
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```


Here, we use the **check** clause to simulate an enumerated type, by specifying that semester must be one of 'Fall', 'Winter', 'Spring', or 'Summer'. Thus, the **check** clause permits attribute domains to be restricted in powerful ways that most programming-language type systems do not permit.

e. Referential Integrity

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause. We illustrate foreign-key declarations by using the SQL DDL definition of part of our university database, shown in Figure 4.8. The definition of the course table has a declaration "**foreign key (dept_name) references department**". This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the department relation. Without this constraint, it is possible for a course to specify a non-existent department name.

By default, in SQL a foreign key references the primary-key attributes of the referenced table. SQL also supports a version of the references clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must, however, be declared as a candidate key of the referenced relation, using either a **primary key** constraint, or a **unique** constraint.

We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

```
dept_name varchar(20) references department
```

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back). However, a foreign key clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation course:

```
create table course
  (...
  foreign key (dept_name) references department
    on delete cascade
    on update cascade,
  ...);
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in department results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the course relation, deleting the tuple that refers to the department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field dept_name in the referencing tuples in course to the new value as well. SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, dept_name) can be set to null (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

```

create table classroom
  (building    varchar (15),
   room_number varchar (7),
   capacity    numeric (4,0),
   primary key (building, room_number))

create table department
  (dept_name  varchar (20),
   building    varchar (15),
   budget      numeric (12,2) check (budget > 0),
   primary key (dept_name))

create table course
  (course_id  varchar (8),
   title       varchar (50),
   dept_name   varchar (20),
   credits     numeric (2,0) check (credits > 0),
   primary key (course_id),
   foreign key (dept_name) references department)

create table instructor
  (ID         varchar (5),
   name       varchar (20),
   dept_name  varchar (20),
   salary    numeric (8,2), check (salary > 29000),
   primary key (ID),
   foreign key (dept_name) references department)

create table section
  (course_id  varchar (8),
   sec_id     varchar (8),
   semester   varchar (6), check (semester in
                                   ('Fall', 'Winter', 'Spring', 'Summer'),
   year       numeric (4,0), check (year > 1759 and year < 2100),
   building   varchar (15),
   room_number varchar (7),
   time_slot_id varchar (4),
   primary key (course_id, sec_id, semester, year),
   foreign key (course_id) references course,
   foreign key (building, room_number) references classroom)

```

Figure 4.8 SQL data definition for part of the university database.

f. Integrity Constraint Violation During a Transaction

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation *person* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *person*. That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the above relation, with the *spouse* attributes set to Mary and John, respectively. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted the foreign-key constraint would hold again.

To handle such situations, the SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction, and not at intermediate steps. A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default, but can be deferred when desired. For constraints declared as deferrable, executing a statement **set constraints *constraint-list* deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction.

However, you should be aware that the default behaviour is to check constraints immediately, and many database implementations do not support deferred constraint checking.

We can work around the problem in the above example in another way, if the spouse attribute can be set to null: We set the spouse attributes to null when inserting the tuples for John and Mary, and we update them later. However, this technique requires more programming effort, and does not work if the attributes cannot be set to null.

g. Complex Check Conditions and Assertions

The SQL standard supports additional constructs for specifying integrity constraints that are described in this section. However, you should be aware that these constructs are not currently supported by most database systems.

As defined by the SQL standard, the predicate in the **check** clause can be an arbitrary predicate, which can include a subquery. If a database implementation supports subqueries in the **check** clause, we could specify the following referential-integrity constraint on the relation section:

```
check (time_slot_id in (select time_slot_id from time_slot))
```

The **check** condition verifies that the `time_slot_id` in each tuple in the section relation is actually the identifier of a time slot in the `time_slot` relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in section, but also when the relation timeslot changes (in this case, when a tuple is deleted or modified in relation `time_slot`).

Another natural constraint on our university schema would be to require that every section has at least one instructor teaching the section. In an attempt to enforce this, we may try to declare that the attributes (`course_id`, `sec_id`, `semester`, `year`) of the section relation form a foreign key referencing the corresponding attributes of the teaches relation. Unfortunately, these attributes do not form a candidate key of the relation teaches. A check constraint similar to that for the timeslot attribute can be used to enforce this constraint, if check constraints with subqueries were supported by a database system.

Complex **check** conditions can be useful when we want to ensure integrity of data, but may be costly to test. For example, the predicate in the **check** clause would not only have to be evaluated when a

```
create assertion credits_earned_constraint check  
(not exists (select ID  
            from student  
            where tot_cred <> (select sum(credits)  
            from takes natural join course  
            where student.ID=takes.ID  
            and grade is not null and grade <> 'F' )
```

Figure 4.9 An assertion example.

Modification is made to the section relation, but may have to be checked if a modification is made to the timeslot relation because that relation is referenced in the subquery.

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertions because they are easily tested and apply to a wide range of database applications. However, there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

- For each tuple in the student relation, the value of the attribute tot cred must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same timeslot.

An assertion in SQL takes the form:

```
create assertion<assertion-name> check <predicate>;
```

In Figure 4.9, we show how the first example of constraints can be written in SQL. Since SQL does not provide a “for all X, P(X)” construct (where P is a predicate), we are forced to implement the constraint by an equivalent construct, “not exists X such that not P(X)”, that can be expressed in SQL.

When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made. Hence, assertions should be used with great care. The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertion that are easier to test.

Currently, none of the widely used database systems supports either subqueries in the check clause predicate, or the **create assertion** construct. However, equivalent functionality can be implemented using triggers.

xiv. SQL Data Types and Schemas

a. Date and Time Types in SQL

In addition to the basic data types, the SQL standard supports several data types relating to dates and times:

- **date**: A calendar date containing a (four-digit) year, month, and day of the month.
- **time**: The time of day, in hours, minutes, and seconds. A variant, **time(p)**, can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying **time with time zone**.
- **timestamp**: A combination of date and time. A variant, **timestamp(p)**, can be used to specify the number of fractional digits for seconds (the default here being 6). Time-zone information is also stored if with time zone is specified.

Date and time values can be specified like this:

```
date '2001-04-25'
```

time '09:30:00'
timestamp '2001-04-25 10:29:01.45'

Dates must be specified in the format year followed by month followed by day, as shown. The seconds field of **time** or **timestamp** can have a fractional part, as in the timestamp above.

We can use an expression of the form **cast e as t** to convert a character string (or string valued expression) e to the type t, where t is one of date, time, or timestamp. The string must be in the appropriate format as illustrated at the beginning of this paragraph. When required, time-zone information is inferred from the system settings.

To extract individual fields of a **date** or **time** valued, we can use **extract** (field from d), where field can be one of **year, month, day, hour, minute, or second**. Timezone information can be extracted using **timezone_hour** and **timezone_minute**.

SQL defines several functions to get the current date and time. For example, **current_date** returns the currentdate, **current_time** returns the current time (with time zone), and **localtime** returns the current local time (without time zone). Timestamps (date plus time) are returned by **current_timestamp** (withtimezone) and **localtimestamp** (local date and time without timezone).

SQL allows comparison operations on all the types listed here, and it allows both arithmetic and comparison operations on the various numeric types. SQL also provides a data type called interval, and it allows computations based on dates and times and on intervals. For example, if x and y are of type date, then $x - y$ is an interval whose value is the number of days from date x to date y. Similarly, adding or subtracting an interval to a date or time gives back a date or time, respectively.

b. Default Values

SQL allows a default value to be specified for an attribute as illustrated by the following create table statement:

```
create table student
  (ID          varchar(5),
   name        varchar(20) not null,
   dept_name   varchar(20),
   tot_cred    numeric(3,0) default 0,
   primary key (ID));
```

The default value of the tot_cred attribute is declared to be 0. As a result, when a tuple is inserted into the student relation, if no value is provided for the tot_cred attribute, its value is set to 0. The following insert statement illustrates how an insertion can omit the value for the tot_cred attribute.

```
insert into student(ID, name, dept_name)
  values ('12789', 'Newman', 'Comp. Sci.');
```

c. Index Creation

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all instructors in the Physics department" or "Find the tot_cred value of the student with ID 22201" references only a fraction of the student records. It is inefficient for the system to read every record and to check ID field for the ID "32556," or the building field for the value "Physics".

An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation. For example, if we create an index on attribute ID of relation student, the database system can find the record with any specified ID value, such as 22201, or 44553, directly, without reading all the tuples of the student relation. An index can also be created on a list of attributes, for example on attributes name, and dept_name of student.

Although the SQL language does not formally define any syntax for creating indices, many databases support index creation using the syntax illustrated below.

```
create index student_ID_index on student(ID);
```

The above statement creates an index named student_ID_index on the attribute ID of the relation student.

When a user submits an SQL query that can benefit from using an index, the SQL query processor automatically uses the index. For example, given an SQL query that selects the student tuple with ID 22201, the SQL query processor would use the index student_ID_index defined above to find the required tuple without reading the whole relation.

d. Large-Object Types

Many current-generation database applications need to store attributes that can be large (of the order of many kilobytes), such as a photograph, or very large (of the order of many megabytes or even gigabytes), such as a high-resolution medical image or video clip. SQL therefore provides large-object data types for character data (**clob**) and binary data (**blob**). The letters “lob” in these data types stand for “Large Object.” For example, we may declare attributes

```
book review clob(10KB)  
image blob(10MB)  
movie blob(2GB)
```

For result tuples containing large objects (multiple megabytes to gigabytes), it is inefficient or impractical to retrieve an entire large object into memory. Instead, an application would usually use an SQL query to retrieve a “locator” for a large object and then use the locator to manipulate the object from the host language in which the application itself is written. For instance, the JDBC application program interface permits a locator to be fetched instead of the entire large object; the locator can then be used to fetch the large object in small pieces, rather than all at once, much like reading data from an operating system file using a read function call.

e. User-Defined Types

SQL supports two forms of user-defined data types. The first form is called **distinct types**. The other form, called **structured data types**, allows the creation of complex data types with nested record structures, arrays, and multisets.

It is possible for several attributes to have the same data type. For example, the name attributes for student name and instructor name might have the same domain: the set of all person names. However, the domains of budget and dept_name certainly ought to be distinct. It is perhaps less clear whether name and dept_name should have the same domain. At the implementation level, both

instructor names and department names are character strings. However, we would normally not consider the query “Find all instructors who have the same name as a department” to be a meaningful query. Thus, if we view the database at the conceptual, rather than the physical, level, name and dept_name should have distinct domains.

More importantly, at a practical level, assigning an instructor’s name to a department name is probably a programming error; similarly, comparing a monetary value expressed in dollars directly with a monetary value expressed in pounds is also almost surely a programming error. A good type system should be able to detect such assignments or comparisons. To support such checks, SQL provides the notion of **distinct types**.

The **create type** clause can be used to define new types. For example, the statements:

```
create type Dollars as numeric(12,2) final;  
create type Pounds as numeric(12,2) final;
```

define the user-defined types Dollars and Pounds to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point. The newly created types can then be used, for example, as types of attributes of relations. For example, we can declare the department table as:

```
create table department  
  (dept_name  varchar(20),  
   building   varchar(15),  
   budget     Dollars);
```

An attempt to assign a value of type Dollars to a variable of type Pounds results in a compile-time error, although both are of the same numeric type. Such an assignment is likely to be due to a programmer error, where the programmer forgot about the differences in currency. Declaring different types for different currencies helps catch such errors.

As a result of strong type checking, the expression (department.budget+20) would not be accepted since the attribute and the integer constant 20 have different types. Values of one type can be cast (that is, converted) to another domain, as illustrated below:

```
cast (department.budget to numeric(12,2))
```

We could do addition on the numeric type, but to save the result back to an attribute of type Dollars we would have to use another cast expression to convert the type back to Dollars.

SQL provides **drop type** and **alter type** clauses to drop or modify types that have been created earlier.

Even before user-defined types were added to SQL (in SQL:1999), SQL had a similar but subtly different notion of domain (introduced in SQL-92), which can add integrity constraints to an underlying type. For example, we could define a domain DDollars as follows.

```
create domain DDollars as numeric(12,2) not null;
```

The domain DDollars can be used as an attribute type, just as we used the type Dollars. However, there are two significant differences between types and domains:

1. Domains can have constraints, such as **not null**, specified on them, and can have default values defined for variables of the domain type, whereas user defined types cannot have constraints or

default values specified on them. User-defined types are designed to be used not just for specifying attribute types, but also in procedural extensions to SQL where it may not be possible to enforce constraints.

2. Domains are not strongly typed. As a result, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

When applied to a domain, the **check** clause permits the schema designer to specify a predicate that must be satisfied by any attribute declared to be from this domain. For instance, a **check** clause can ensure that an instructor's salary domain allows only values greater than a specified value:

```
create domain YearlySalary numeric(8,2)
constraint salary_value_test check(value >= 29000.00);
```

The domain YearlySalary has a constraint that ensures that the YearlySalary is greater than or equal to \$29,000.00. The clause **constraint** salary_value_test is optional, and is used to give the name salary_value_test to the constraint. The name is used by the system to indicate the constraint that an update violated.

As another example, a domain can be restricted to contain only a specified set of values by using the **in** clause:

```
create domain degree_level varchar(10)
constraint degree_level_test
check (value in ('Bachelors', 'Masters', or 'Doctorate'));
```

f. Create Table Extensions

Applications often require creation of tables that have the same schema as an existing table. SQL provides a **create table like** extension to support this task:

```
create table temp_instructor like instructor;
```

The above statement creates a new table temp_instructor that has the same schema as instructor.

When writing a complex query, it is often useful to store the result of a query as a new table; the table is usually temporary. Two statements are required, one to create the table (with appropriate columns) and the second to insert the query result into the table. SQL:2003 provides a simpler technique to create a table containing the results of a query. For example the following statement creates a table t1 containing the results of a query.

```
create table t1 as
  (select *
   from instructor
   where dept_name= 'Music')
with data;
```

By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name.

As defined by the SQL:2003 standard, if the **with data** clause is omitted, the table is created but not populated with data. However many implementations populate the table with data by default even if the **with data** clause is omitted.

Note that several implementations support the functionality of **create table... like** and **create table...as** using different syntax; see the respective system manuals for further details.

The above **create table...as** statement closely resembles the create view statement and both are defined by using queries. The main difference is that the contents of the table are set when the table is created, whereas the contents of a view always reflect the current query result.

g. Schemas, Catalogs, and Environments

To understand the motivation for schemas and catalogs, consider how files are named in a file system. Early file systems were flat; that is, all files were stored in a single directory. Current file systems, of course, have a directory (or, synonymously, folder) structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, */users/avi/db-book/chapter3.tex*.

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**. (Some database implementations use the term “database” in place of the term catalog.)

In order to perform any actions on a database, a user (or a program) must first connect to the database. The user must provide the user name and usually, a password for verifying the identity of the user. Each user has a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for the connection; this corresponds to the current directory being set to the user’s home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name may be used, for example,

catalog5.univ_schema.course

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus if catalog5 is the default catalog, we can use univ_schema.course to identify the same relation uniquely.

If a user wishes to access a relation that exists in a different schema than the default schema for that user, the name of the schema must be specified. However, if a relation is in the default schema for a particular user, then even the schema name may be omitted. Thus we can use just course if the default catalog is catalog5 and the default schema is univ_schema.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application— one a production version, other test versions—can run on the same database system.

The default catalog and schema are part of an SQL environment that is set up for each connection. The environment additionally contains the user identifier (also referred to as the authorization

identifier). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema.

We can create and drop schemas by means of **create schema** and **drop schema** statements. In most database systems, schemas are also created automatically when user accounts are created, with the schema name set to the user account name. The schema is created in either a default catalog, or a catalog specified in creating the user account. The newly created schema becomes the default schema for the user account.

Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

xv. Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a privilege. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected.

In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations. A user who has some form of authorization may be allowed to pass on (grant) this authorization to other users, or to withdraw (revoke) an authorization that was granted earlier. In this section, we see how each of these authorizations can be specified in SQL.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a **superuser**, administrator, or operator for an operating system.

a. Granting and Revoking of Privileges

The SQL standard includes the privileges select, insert, update, and delete. The privilege **all privileges** can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically.

The SQL data-definition language includes commands to grant and revoke privileges. The grant statement is used to confer authorization. The basic form of this statement is:

```
grant <privilege list>  
on <relation name or view name>  
to <user/role list>;
```

The privilege list allows the granting of several privileges in one command.

The **select** authorization on a relation is required to read tuples in the relation. The following **grant** statement grants database users Amit and Satoshi **select** authorization on the department relation:

```
grant select on department to Amit, Satoshi;
```

This allows those users to run queries on the department relation.

The **update** authorization on a relation allows a user to update any tuple in the relation. The update authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users Amit and Satoshi update authorization on the budget attribute of the department relation:

```
grant update (budget) on department to Amit, Satoshi;
```

The **insert** authorization on a relation allows a user to insert tuples into the relation. The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to null.

The **delete** authorization on a relation allows a user to delete tuples from a relation.

The user name **public** refers to all current and future users of the system. Thus, privileges granted to public are implicitly granted to all current and future users.

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. SQL allows a privilege grant to specify that the recipient may further grant the privilege to another user.

It is worth noting that the SQL authorization mechanism grants privileges on an entire relation, or on specified attributes of a relation. However, it does not permit authorizations on specific tuples of a relation.

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

```
revoke <privilege list>  
on <relation name or view name>  
from <user/role list>;
```

Thus, to revoke the privileges that we granted previously, we write

```
revoke select on department from Amit, Satoshi;  
revoke update (budget) on department from Amit, Satoshi;
```

Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user.

b. Roles

Consider the real-world roles of various people in a university. Each instructor must have the same types of authorizations on the same set of relations. Whenever a new instructor is appointed, she will have to be given all these authorizations individually.

A better approach would be to specify the authorizations that every instructor is to be given, and to identify separately which database users are instructors. The system can use these two pieces of information to determine the authorizations of each instructor. When a new instructor is hired, a user identifier must be allocated to him, and he must be identified as an instructor. Individual permissions given to instructors need not be specified again.

The notion of roles captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that she is authorized to perform.

In our university database, examples of roles could include instructor, teaching_assistant, student, dean, and department_chair.

A less preferable alternative would be to create an instructor userid, and permit each instructor to connect to the database using the instructor userid. The problem with this approach is that it would not be possible to identify exactly which instructor carried out a database update, leading to security risks. The use of roles has the benefit of requiring users to connect to the database with their own userid.

Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are.

Roles can be created in SQL as follows:

```
create role instructor;
```

Roles can then be granted privileges just as the users can, as illustrated in this statement:

```
grant select on takes to instructor;
```

Roles can be granted to users, as well as to other roles, as these statements show:

```
grant dean to Amit;  
create role dean;  
grant instructor to dean;  
grant dean to Satoshi;
```

Thus the privileges of a user or a role consist of:

- All privileges directly granted to the user/role.
- All privileges granted to roles that have been granted to the user/role.

Note that there can be a chain of roles; for example, the role `teaching_assistant` may be granted to all instructors. In turn the role `instructor` is granted to all deans. Thus, the dean role inherits all privileges granted to the roles `instructor` and to `teaching assistant` in addition to privileges granted directly to dean.

When a user logs in to the database system, the actions executed by the user during that session have all the privileges granted directly to the user, as well as all privileges granted to roles that are granted (directly or indirectly via other roles) to that user. Thus, if a user Amit has been granted the role `dean`, user Amit holds all privileges granted directly to Amit, as well as privileges granted to `dean`, plus privileges granted to `instructor`, and `teaching_assistant` if, as above, those roles were granted (directly or indirectly) to the role `dean`.

It is worth noting that the concept of role-based authorization is not specific to SQL, and role-based authorization is used for access control in a wide variety of shared applications.

c. Authorization on Views

In our university example, consider a staff member who needs to know the salaries of all faculty in a particular department, say the Geology department. This staff member is not authorized to see information regarding faculty in other departments. Thus, the staff member must be denied direct access to the `instructor` relation. But, if he is to have access to the information for the Geology department, he might be granted access to a view that we shall call `geo_instructor`, consisting of only those `instructor` tuples pertaining to the Geology department. This view can be defined in SQL as follows:

```
create view geo_instructor as
  (select *
   From instructor
   Where dept_name = 'Geology');
```

Suppose that the staff member issues the following SQL query:

```
select *
from geo_instructor;
```

Clearly, the staff member is authorized to see the result of this query. However, when the query processor translates it into a query on the actual relations in the database, it produces a query on `instructor`. Thus, the system must check authorization on the clerk's query before it begins query processing.

A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user who creates a view cannot be given update authorization on a view without having update authorization on the relations used to define the view. If a user creates a view on which no authorization can be granted, the system will deny the view creation request. In our `geo_instructor` view example, the creator of the view must have select authorization on the `instructor` relation.

SQL supports the creation of functions and procedures, which may in turn contain queries and updates. The execute privilege can be granted on a function or procedure, enabling a user to execute the function/procedure. By default, just like views, functions and procedures have all the privileges that the creator of the function or procedure had. In effect, the function or procedure runs as if it were invoked by the user who created the function.

Although this behavior is appropriate in many situations, it is not always appropriate. Starting with SQL:2003, if the function definition has an extra clause **sql security invoker**, then it is executed under the privileges of the user who invokes the function, rather than the privileges of the definer of the function. This allows the creation of libraries of functions that can run under the same authorization as the invoker.

d. Authorizations on Schema

The SQL standard specifies a primitive authorization mechanism for the database schema: Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices.

However, SQL includes a **references** privilege that permits a user to declare foreign keys when creating relations. The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user Mariano to create relations that reference the key `branch_name` of the `branch` relation as a foreign key:

```
grant references (dept_name) on department to Mariano;
```

Initially, it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, recall that foreign key constraints restrict deletion and update operations on the referenced relation. Suppose Mariano creates a foreign key in a relation `r` referencing the `dept_name` attribute of the `department` relation and then inserts a tuple into `r` pertaining to the Geology department. It is no longer possible to delete the Geology department from the `department` relation without also modifying relation `r`. Thus, the definition of a foreign key by Mariano restricts future activity by other users; therefore, there is a need for the **references** privilege.

Continuing to use the example of the `department` relation, the **references** privilege on `department` is also required to create a **check** constraint on a relation `r` if the constraint has a subquery referencing `department`. This is reasonable for the same reason as the one we gave for foreign-key constraints; a check constraint that references a relation limits potential updates to that relation.

e. Transfer of Privileges

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users. By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, if we wish to allow Amit the **select** privilege on `department` and allow Amit to grant this privilege to others, we write:

```
grant select on department to Amit with grant option;
```

The creator of an object (relation/view/role) holds all privileges on the object, including the privilege to grant privileges to others.

Consider, as an example, the granting of update authorization on the `teaches` relation of the university database. Assume that, initially, the database administrator grants update authorization on `teaches` to users `U1`, `U2`, and `U3`, who may in turn pass on this authorization to other users. The passing of a

specific authorization from one user to another can be represented by an authorization graph. The nodes of this graph are the users.

Consider the graph for update authorization on teaches. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on teaches to U_j . The root of the graph is the database administrator. In the sample graph in Figure 4.10 observe that user U_5 is granted authorization by both U_1 and U_2 ; U_4 is granted authorization by only U_1 .

A user has an authorization if and only if there is a path from the root of the authorization graph (the node representing the database administrator) down to the node representing the user.

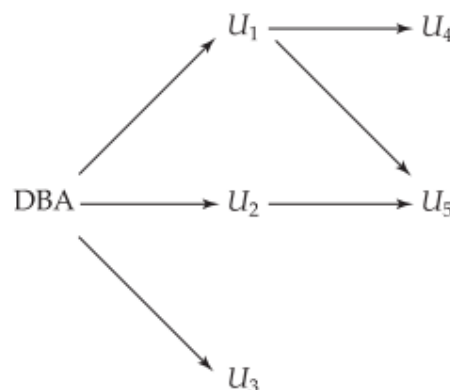


Figure 4.10 Authorization-grant graph (U_1, U_2, \dots, U_5 are users and DBA refers to the database administrator).

f. Revoking of Privileges

Suppose that the database administrator decides to revoke the authorization of user U_1 . Since U_4 has authorization from U_1 , that authorization should be revoked as well. However, U_5 was granted authorization by both U_1 and U_2 . Since the database administrator did not revoke update authorization on teaches from U_2 , U_5 retains update authorization on teaches. If U_2 eventually revokes authorization from U_5 , then U_5 loses the authorization.

A pair of devious users might attempt to defeat the rules for revocation of authorization by granting authorization to each other. For example, if U_2 is initially granted an authorization by the database administrator, and U_2 further grants it to U_3 . Suppose U_3 now grants the privilege back to U_2 . If the database administrator revokes authorization from U_2 , it might appear that U_2 retains authorization through U_3 . However, note that once the administrator revokes authorization from U_2 , there is no path in the authorization graph from the root to either U_2 or to U_3 . Thus, SQL ensures that the authorization is revoked from both the users.

As we just saw, revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called cascading revocation. In most database systems, cascading is the default behavior. However, the **revoke** statement may specify **restrict** in order to prevent cascading revocation:

revoke select on department from Amit, Satoshi restrict;

In this case, the system returns an error if there are any cascading revocations, and does not carry out the revoke action.

The keyword **cascade** can be used instead of **restrict** to indicate that revocation should cascade; however, it can be omitted, as we have done in the preceding examples, since it is the default behavior.

The following **revoke** statement revokes only the grant option, rather than the actual **select** privilege:

```
revoke grant option for select on department from Amit;
```

Note that some database implementations do not support the above syntax; instead, the privilege itself can be revoked, and then granted again without the grant option.

Cascading revocation is inappropriate in many situations. Suppose Satoshi has the role of dean, grants instructor to Amit, and later the role dean is revoked from Satoshi (perhaps because Satoshi leaves the university); Amit continues to be employed on the faculty, and should retain the instructor role.

To deal with the above situation, SQL permits a privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default, the current role associated with a session is null (except in some special cases). The current role associated with a session can be set by executing `set role role name`. The specified role must have been granted to the user, else the `setrole` statement fails.

To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

```
granted by current_role
```

to the grant statement, provided the current role is not null.

Suppose the granting of the role instructor (or other privileges) to Amit is done using the **granted by current_role** clause, with the current role set to dean), instead of the grantor being the user Satoshi. Then, revoking of roles/privileges (including the role dean) from Satoshi will not result in revoking of privileges that had the grantor set to the role dean, even if Satoshi was the user who executed the grant; thus, Amit would retain the instructor role even after Satoshi's privileges are revoked.

xvi. Database Programming: Techniques and Issues

We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL databases from application programs. Our presentation of SQL in Chapters 4 and 5 focused on the language constructs for various database operations—from schema definition and constraint specification to querying, updating, and specifying views. Most database systems have an interactive interface where these SQL commands can be typed directly into a monitor for execution by the database system. For example, in a computer system where the Oracle RDBMS is installed, the command `SQLPLUS` starts the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is, “; <cr>”). Alternatively, a file of commands can be created and executed through the interactive interface by typing `@<filename>`. The system will execute the commands written in the file and display the results, if any.

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, in practice, the majority of database interactions are executed through programs that have been carefully designed and tested. These programs are generally known as **application programs** or **database applications**, and are used as canned transactions by the end users, as discussed in Section 1.4.3. Another common use of database programming is to access a database through an application program that implements a **Web interface**, for example, when making airline reservations or online purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands.

a. Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. **Embedding database commands in a general-purpose programming language.** In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program. A **precompiler** or **preprocessor** scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as **embedded SQL**.

2. **Using a library of database functions.** A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands and any other necessary information are included as parameters in the function calls. This approach provides what is known as an **application programming interface (API)** for accessing a database from application programs.

3. **Designing a brand-new language.** A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full fledged programming language. An example of this approach is Oracle's PL/SQL.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is impedance mismatch, which does not occur in the third approach.

b. Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records). The first problem that may occur is that the data types of the programming language differ from the attribute data types that are available in the data model. Hence, it is necessary to have a **binding** for each host programming language that specifies for each attribute type the compatible programming language types. A different binding is needed for each programming language because different languages have different data types. For example, the data types available in C/C++ and Java are different, and both differ from the SQL data types, which are the standard data types for relational databases.

Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. In the program, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a **query result** in order to access a single tuple at a time and to extract individual values from the tuple. The extracted attribute values are typically copied to appropriate program variables for further processing by the program. A **cursor** or **iterator variable** is typically used to loop over the tuples in a query result. Individual values within each tuple are then extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is Oracle's PL/SQL. The SQL standard also has a proposal for such a database programming language, known as SQL/PSM. For object databases, the object data model is quite similar to the data model of the Java programming language, so the impedance mismatch is greatly reduced when Java is used as the host language for accessing a Java-compatible object database. Several database programming languages have been implemented as research prototypes.

c. Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. A common architecture for database access is the client/server model, where a **client program** handles the logic of a software application, but includes some calls to one or more **database servers** to access or update the data. When writing such a program, a common sequence of interaction is the following:

1. When the client program requires access to a particular database, the program must first establish or open a connection to the database server. Typically, this involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.
2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.
3. When the program no longer needs access to a particular database, it should terminate or close the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established simultaneously.

xvii. Embedded SQL

In this section, we give an overview of the technique for how SQL statements can be embedded in a general-purpose programming language. The examples used with the C language are known as **embedded SQL**. In this embedded approach, the programming language is called the **host language**. Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program.

a. Retrieving Single Tuples with Embedded SQL

To illustrate the concepts of embedded SQL, we will use C as the host programming language. When using C as the host language, an embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded SQL statements from the host language code. The SQL statements within a program are terminated by a matching END-EXEC or by a semicolon (;). Similar rules apply to embedding SQL in other programming languages.

Within an embedded SQL command, we may refer to specially declared C program variables. These are called **shared variables** because they are used in both the C program and the embedded SQL statements. Shared variables are prefixed by a colon (:) when they appear in an SQL statement. This distinguishes program variable names from the names of database schema constructs such as attributes (column names) and relations (table names). It also allows program variables to have the same names as attribute names, since they are distinguishable by the colon (:) prefix in the SQL statement. Names of database schema constructs—such as attributes and relations—can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the colon (:) prefix.

Suppose that we want to write C programs to process the **COMPANY** database in Figure 3.5. We need to declare program variables to match the types of the database attributes that the program will process. The programmer can choose the names of the program variables; they may or may not have names that are identical to their corresponding database attributes. We will use the C program variables declared in Figure 13.1 for all our examples and show C program segments without variable declarations. Shared variables are declared within a declare section in the program, as shown in Figure 13.1 (lines 1 through 7).

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

Figure 13.1
C program variables used in the
embedded SQL examples E1 and E2.

A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types long, short, float, and double, respectively. Fixed-length and varying-length strings (CHAR[i], VARCHAR[i]) in SQL can be mapped to arrays of characters (char [i+1], varchar [i+1]) in C that are one character longer than the SQL type because strings in C are terminated by a NULL character (\0), which is not part of the character string itself.⁶ Although varchar is not a standard C data type, it is permitted when C is used for SQL database programming.

Notice that the only embedded SQL commands in Figure 13.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the COMPANY database in Figure 3.5 that was declared by the SQL DDL in Figure 4.1. The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the executing program. Line 0 shows a program variable loop that will not be used in any embedded SQL statement, so it is declared outside the SQL declare section.

Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

```
CONNECT TO <server name> AS <connection name>  
AUTHORIZATION <user account name and password> ;
```

In general, since a user or program can access several database servers, several connections can be established, but only one connection can be active at any point in time. The programmer or user can use the <connection name> to change from the currently active connection to a different one by using the following command:

```
SET CONNECTION <connection name> ;
```

Once a connection is no longer needed, it can be terminated by the following command:

```
DISCONNECT <connection name> ;
```

In the examples in this chapter, we assume that the appropriate connection has already been established to the COMPANY database, and that it is the currently active connection.

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE.

The two special **communication variables** that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE. The SQLCODE variable shown in Figure 13.1 is an integer variable. After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result. If SQLCODE < 0, this indicates some error has occurred. In some systems—for example, in the Oracle RDBMS— SQLCODE is a field in a record structure called SQLCA (SQL communication area), so it is referenced as SQLCA.SQLCODE. In this case, the definition of SQLCA must be included in the C program by including the following line:

```
EXEC SQL include SQLCA ;
```

In later versions of the SQL standard, a communication variable called **SQLSTATE** was added, which is a string of five characters. A value of '00000' in SQLSTATE indicates no error or exception; other values indicate various errors or exceptions. For example, '02000' indicates 'no more data' when using SQLSTATE. Currently, both SQLSTATE and SQLCODE are available in the SQL standard. Many of the error and exception codes returned in SQLSTATE are supposed to be standardized for all SQL vendors and platforms,⁷ whereas the codes returned in SQLCODE are not standardized but are defined by the DBMS vendor. Hence, it is generally better to use SQLSTATE because this makes error handling in the application programs independent of a particular DBMS. As an exercise, the reader should rewrite the examples given later in this chapter using SQLSTATE instead of SQLCODE.

Example of Embedded SQL Programming. Our first example to illustrate embedded SQL programming is a repeating program segment (loop) that takes as input a Social Security number of an employee and prints some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 13.2. The program reads (inputs) an Ssn value and then retrieves the EMPLOYEE tuple with that Ssn from the database via the embedded SQL command. The **INTO** clause (line 5) specifies the program variables into which attribute values from

the database record are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier. The INTO clause can be used in this way only when the query result is a single record; if multiple records are retrieved, an error will be generated.

Line 7 in E1 illustrates the communication between the database and the program through the special variable SQLCODE. If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions. Line 7 checks this and assumes that if an error occurred, it was because no EMPLOYEE tuple existed with the given Ssn; therefore it outputs a message to that effect (line 8).

```
//Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2)   prompt("Enter a Social Security Number: ", ssn) ;
3)   EXEC SQL
4)     select Fname, Minit, Lname, Address, Salary
5)     into :fname, :minit, :lname, :address, :salary
6)     from EMPLOYEE where Ssn = :ssn ;
7)   if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)     else printf("Social Security Number does not exist: ", ssn) ;
9)   prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

Figure 13.2
Program segment E1,
a C program segment
with embedded SQL.

In E1 a single record is selected by the embedded SQL query (because Ssn is a key attribute of EMPLOYEE). When a single record is retrieved, the programmer can assign its attribute values directly to C program variables in the INTO clause, as in line 5. In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. The concept of a cursor is used to allow tuple-at-a-time processing of a query result by the host language program.

b. Retrieving Multiple Tuples with Embedded SQL Using Cursors

We can think of a **cursor** as a pointer that points to a single tuple (row) from the result of a query that retrieves multiple tuples. The cursor is declared when the SQL query command is declared in the program. Later in the program, an **OPEN CURSOR** command fetches the query result from the database and sets the cursor to a position before the first row in the result of the query. This becomes the **current row** for the cursor. Subsequently, **FETCH** commands are issued in the program; each **FETCH** moves the cursor to the next row in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the **FETCH** command by an INTO clause. The cursor variable is basically an iterator that iterates (loops) over the tuples in the query result—one tuple at a time.

To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked. If a **FETCH** command is issued that results in moving the cursor past the last tuple in the result of the query, a positive value (SQLCODE > 0) is returned in SQLCODE, indicating that no data (tuple) was found (or the string '02000' is returned in SQLSTATE). The programmer uses this to terminate a loop over the tuples in the query result. In general, numerous cursors can be opened at the same time. A **CLOSE CURSOR** command is issued to indicate that we are done with processing the result of the query associated with that cursor.

An example of using cursors to process a query result with multiple records is shown in Figure 13.3, where a cursor called EMP is declared in line 4.

Figure 13.3

Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

```
//Program Segment E2:
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2)   select Dnumber into :dnumber
3)   from DEPARTMENT where Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5)   select Ssn, Fname, Minit, Lname, Salary
6)   from EMPLOYEE where Dno = :dnumber
7)   FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11)   printf("Employee name is:", Fname, Minit, Lname) ;
12)   prompt("Enter the raise amount: ", raise) ;
13)   EXEC SQL
14)     update EMPLOYEE
15)       set Salary = Salary + :raise
16)       where CURRENT OF EMP ;
17)   EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;
```

The EMP cursor is associated with the SQL query declared in lines 5 through 6, but the query is not executed until the OPEN EMP command (line 8) is processed. The OPEN <cursor name> command executes the query and fetches its result as a table into the program workspace, where the program can loop through the individual rows (tuples) by subsequent FETCH <cursor name> commands (line 9). We assume that appropriate C program variables have been declared as in Figure 13.1. The program segment in E2 reads (inputs) a department name (line 0), retrieves the matching department number from the database (lines 1 to 3), and then retrieves the employees who work in that department via the declared EMP cursor. A loop (lines 10 to 18) iterates over each record in the query result, one at a time, and prints the employee name. The program then reads (inputs) a raise amount for that employee (line 12) and updates the employee's salary in the database by the raise amount that was provided (lines 14 to 16).

This example also illustrates how the programmer can update database records. When a cursor is defined for rows that are to be modified (**updated**), we must add the clause **FOR UPDATE OF** in the cursor declaration and list the names of any attributes that will be updated by the program. This is illustrated in line 7 of code segment E2. If rows are to be **deleted**, the keywords **FOR UPDATE** must be added without specifying any attributes. In the embedded UPDATE (or DELETE) command, the condition **WHERE CURRENT OF <cursor name>** specifies that the current tuple referenced by the cursor is the one to be updated (or deleted), as in line 16 of E2.

Notice that declaring a cursor and associating it with a query (lines 4 through 7 in E2) does not execute the query; the query is executed only when the OPEN <cursor name> command (line 8) is executed. Also notice that there is no need to include the **FOR UPDATE OF** clause in line 7 of E2 if the results of the query are to be used for retrieval purposes only (no update or delete).

General Options for a Cursor Declaration. Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR  
[ WITH HOLD ] FOR <query specification>  
[ ORDER BY <ordering specification> ]  
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

We already briefly discussed the options listed in the last line. The default is that the query is for retrieval purposes (FOR READ ONLY). If some of the tuples in the query result are to be updated, we need to specify FOR UPDATE OF <attribute list> and list the attributes that may be updated. If some tuples are to be deleted, we need to specify FOR UPDATE without any attributes listed.

When the optional keyword SCROLL is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access. A **fetch orientation** can be added to the FETCH command, whose value can be one of NEXT, PRIOR, FIRST, LAST, ABSOLUTE i, and RELATIVE i. In the latter two commands, i must evaluate to an integer value that specifies an absolute tuple position within the query result (for ABSOLUTE i), or a tuple position relative to the current cursor position (for RELATIVE i). The default fetch orientation, which we used in our examples, is NEXT. The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order. When SCROLL is specified on the cursor, the general form of a FETCH command is as follows, with the parts in square brackets being optional:

```
FETCH [ [ <fetch orientation> ] FROM ] <cursor name> INTO <fetch target list> ;
```

The ORDER BY clause orders the tuples so that the FETCH command will fetch them in the specified order. It is specified in a similar manner to the corresponding clause for SQL queries. The last two options when declaring a cursor (INSENSITIVE and WITH HOLD) refer to transaction characteristics of database programs.

DBMS Lecture Notes (18MCA31)

MODULE 4

i. Informal Design Guidelines for Relation schemas

There are four informal guidelines that may be used as measures to determine the quality of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another.

a. Imparting Clear Semantics to Attributes in Relations

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. In general, the easier it is to explain the semantics of the relation, the better the relation schema design will be. To illustrate this, consider Figure 15.1, a simplified version of the COMPANY relational database schema in previous module, and Figure 15.2, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber). The Dnumber attribute is a foreign key that represents an implicit relationship between EMPLOYEE and DEPARTMENT. The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, while Dnum of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's attributes can be explained is an informal measure of how well the relation is designed.

Figure 15.1

A simplified COMPANY relational database schema.

EMPLOYEE F.K.

Ename	<u>Ssn</u>	Bdate	Address	Dnumber
-------	------------	-------	---------	---------

P.K.

DEPARTMENT F.K.

Dname	<u>Dnumber</u>	Dmgr_ssn
-------	----------------	----------

P.K.

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

F.K.

P.K.

PROJECT F.K.

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

P.K.

WORKS_ON

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------

F.K. F.K.

P.K.

Figure 15.2

Sample database state for the relational database schema in Figure 15.1.

EMPLOYEE

<u>Ename</u>	<u>Ssn</u>	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

<u>Dname</u>	<u>Dnumber</u>	<u>Dmgr_ssn</u>
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

<u>Ssn</u>	<u>Pnumber</u>	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

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

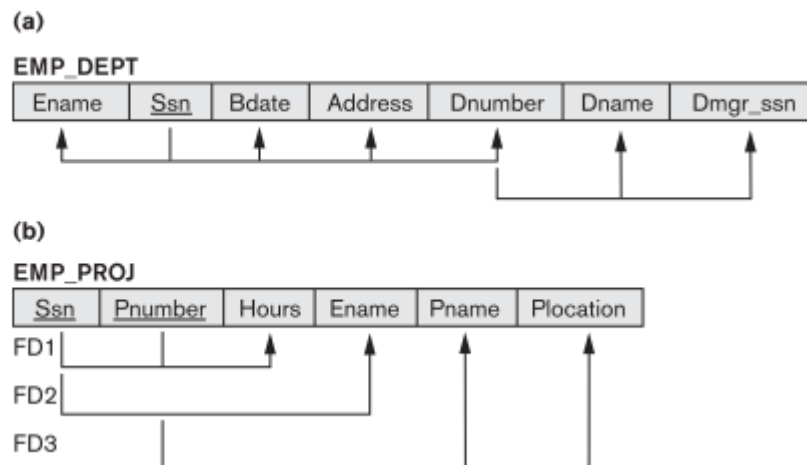
The semantics of the other two relation schemas in Figure 15.1 are slightly more complex. Each tuple in DEPT_LOCATIONS gives a department number (Dnumber) and one of the locations of the department (Dlocation). Each tuple in WORKS_ON gives an employee Social Security number (Ssn), the project number of one of the projects that the employee works on (Pnumber), and the number of hours per week that the employee works on that project (Hours). However, both schemas have a well-defined and unambiguous interpretation. The schema DEPT_LOCATIONS represents a multivalued attribute of DEPARTMENT, whereas WORKS_ON represents an M:N relationship between EMPLOYEE and PROJECT. Hence, all the relation schemas in Figure 15.1 may be considered as easy to explain and therefore good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

Guideline 1

Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Figure 15.3

Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.



Examples of Violating Guideline 1. The relation schemas in Figures 15.3(a) and 15.3(b) also have clear semantics. A tuple in the EMP_DEPT relation schema in Figure 15.3(a) represents a single employee but includes additional information—namely, the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager. For the EMP_PROJ relation in Figure 15.3(b), each tuple relates an employee to a project but also includes the employee name (Ename), project name (Pname), and project location (Plocation). Although there is nothing wrong logically with these two relations, they violate Guideline 1 by mixing attributes from distinct real-world entities: EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship. Hence, they fare poorly against the above measure of design quality. They may be used as views, but they cause problems when used as base relations.

b. Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 15.2 with that for an EMP_DEPT base relation in Figure 15.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department. In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 15.2. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 15.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Redundancy Redundancy

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation
123456789	1	32.5	Smith, John B.	ProductX	Bellaire
123456789	2	7.5	Smith, John B.	ProductY	Sugarland
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston
888665555	20	Null	Borg, James E.	Reorganization	Houston

Figure 15.4

Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 15.2. These may be stored as base relations for performance reasons.

Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.

Insertion Anomalies. Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are consistent with the corresponding values for department 5 in other tuples in EMP_DEPT. In the design of Figure 15.2, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.

- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its primary key. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values any more. This problem does not occur in the design of Figure 15.2 because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the database of Figure 15.2 because DEPARTMENT tuples are stored separately.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence, we can state the next guideline as follows.

Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines. It is important to note that these guidelines may sometimes have to be violated in order to improve the performance of certain queries. If EMP_DEPT is used as a stored relation (known otherwise as a materialized view) in addition to the base relations of EMPLOYEE and DEPARTMENT, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates). This way, whenever the base relation is updated, we do not end up with inconsistencies. In general, it is advisable to use anomaly-free base relations and to specify views that include the joins for placing together the attributes frequently referenced in important queries.

c. NULL Values in Tuples

In some schema designs we may group many attributes together into a “fat” relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level. Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute does not apply to this tuple. For example, Visa_status may not apply to U.S. students.

- The attribute value for this tuple is unknown. For example, the Date_of_birth may be unknown for an employee.
- The value is known but absent; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet. Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

Guideline 3

As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns (with the appropriate key columns). For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

d. Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 15.5(a), which can be used instead of the single EMP_PROJ relation in Figure 15.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on some project whose location is Plocation. A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation. Figure 15.5(b) shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the EMP_PROJ relation in Figure 15.4, which are obtained by applying the appropriate PROJECT (π) operations to EMP_PROJ (ignore the dashed lines in Figure 15.5(b) for now).

(a)

EMP_LOCS

Ename	Plocation
-------	-----------

P.K.

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
-----	---------	-------	-------	-----------

P.K.

(b)

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

Figure 15.5

Particularly poor design for the EMP_PROJ relation in Figure 15.3(b). (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 15.4 onto the relations EMP_LOCS and EMP_PROJ1.

Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ. In Figure 15.6, the result of applying the join to only the tuples above the dashed lines in Figure 15.5(b) is shown (to reduce the size of the resulting relation). Additional tuples that were not in EMP_PROJ are called **spurious tuples** because they represent spurious information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 15.6.

Ssn	Pnumber	Hours	Pname	Plocation	Ename
123456789	1	32.5	ProductX	Bellaire	Smith, John B.
* 123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
123456789	2	7.5	ProductY	Sugarland	Smith, John B.
* 123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
* 123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
* 666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
* 453453453	1	20.0	ProductX	Bellaire	Smith, John B.
453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
* 453453453	2	20.0	ProductY	Sugarland	Smith, John B.
453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
* 453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
* 333445555	2	10.0	ProductY	Sugarland	Smith, John B.
* 333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
* 333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
* 333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

*
*
*

Figure 15.6
Result of applying NATURAL JOIN to the tuples above the dashed lines in EMP_PROJ1 and EMP_LOCS of Figure 15.5. Generated spurious tuples are marked by asterisks.

Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case Plocation is the attribute that relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1. We can now informally state another design guideline.

Guideline 4

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

ii. Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. The single most important concept in relational schema design theory is that of a functional dependency. In this section we formally define the concept, and in later sections we see how it can be used to define normal forms for relation schemas.

a. Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A_1, A_2, \dots, A_n ; let us think of the whole database as being described by a single **universal** relation schema $R = \{A_1, A_2, \dots, A_n\}$. We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.

A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R . The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) determine the values of the Y component. We also say that there is a functional dependency from X to Y , or that Y is **functionally dependent** on X . The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X -value, they must necessarily agree on their Y value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X -value in any relation instance $r(R)$ —that is, X is a **candidate key** of R —this implies that $X \rightarrow Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state $r(R)$ will have the same value of X). If X is a candidate key of R , then $X \rightarrow R$.
- If $X \rightarrow Y$ in R , this does not say whether or not $Y \rightarrow X$ in R .

A functional dependency is a property of the **semantics or meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R —that is, how they relate to one another—to specify the functional dependencies that should hold on all relation states (extensions) r of R . Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R . Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold at all times. Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, $\{\text{State}, \text{Driver_license_number}\} \rightarrow \text{Ssn}$ should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation. It is also possible that certain functional dependencies may cease to exist in the real world if the relationship changes. For example, the FD $\text{Zip_code} \rightarrow \text{Area_code}$ used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

Consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

- a. $\text{Ssn} \rightarrow \text{Ename}$
- b. $\text{Pnumber} \rightarrow \{\text{Pname}, \text{Plocation}\}$
- c. $\{\text{Ssn}, \text{Pnumber}\} \rightarrow \text{Hours}$

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or given a value of Ssn, we know the value of Ename, and so on.

A functional dependency is a property of the relation schema R, not of a particular legal relation state r of R. Therefore, an FD cannot be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R. For example, Figure 15.7 shows a particular state of the TEACH relation schema. Although at first glance we may think that Text → Course, we cannot confirm this unless we know that it is true for all possible legal states of TEACH. It is, however, sufficient to demonstrate a single counterexample to disprove a functional dependency. For example, because 'Smith' teaches both 'Data Structures' and 'Data Management,' we can conclude that Teacher does not functionally determine Course.

TEACH		
Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Figure 15.7

A relation state of TEACH with a possible functional dependency TEXT → COURSE. However, TEACHER → COURSE is ruled out.

Given a populated relation, one cannot determine which FDs hold and which do not unless the meaning of and the relationships among the attributes are known. All one can say is that a certain FD may exist if it holds in that particular extension. One cannot guarantee its existence until the meaning of the corresponding attributes is clearly understood. One can, however, emphatically state that a certain FD does not hold if there are tuples that show the violation of such an FD. See the illustrative example relation in Figure 15.8.

Figure 15.8
A relation R(A, B, C, D) with its extension.

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

Here, the following FDs may hold because the four tuples in the current extension have no violation of these constraints: B → C; C → B; {A, B} → C; {A, B} → D; and {C, D} → B. However, the following do not hold because we already have violations of them in the given extension: A → B (tuples 1 and 2 violate this constraint); B → A (tuples 2 and 3 violate this constraint); D → C (tuples 3 and 4 violate it).

Figure 15.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are semantically obvious; usually,

however, numerous other functional dependencies hold in all legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F. Those other dependencies can be inferred or deduced from the FDs in F.

iii. Normal Forms based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify some aspects of the semantics of relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the normalization process for relational schema design. Most practical relational design projects take one of the following two approaches:

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms, using the normalization theory presented in this chapter and the next. We focus in this section on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically.

a. Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to certify whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as relational design by analysis. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies. It can be considered as a “filtering” or “purification” process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—**the normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
- A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be normalized to any desired degree

The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Normal forms, when considered in isolation from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
- The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and must be achieved at any cost, whereas the dependency preservation property, although desirable, is sometimes sacrificed.

b. Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF, the practical utility of these normal forms becomes questionable when the constraints on which they are based are rare, and hard to understand or to detect by the database designers and users who must discover these constraints. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers need not normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons. Doing so incurs the corresponding penalties of dealing with the anomalies.

Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

c. Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema.

A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A key K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.

The difference between a key and a superkey is that a key has to be minimal; that is, if we have a key $K = \{A_1, A_2, \dots, A_k\}$ of R , then $K - \{A_i\}$ is not a key of R for any $A_i, 1 \leq i \leq k$. In Figure 15.1, $\{Ssn\}$ is a key for EMPLOYEE, whereas $\{Ssn\}$, $\{Ssn, Ename\}$, $\{Ssn, Ename, Bdate\}$, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a candidate key. One of the candidate keys is arbitrarily designated to be the primary key, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 15.1, $\{Ssn\}$ is the only candidate key for EMPLOYEE, so it is also the primary key.

An attribute of relation schema R is called a **prime attribute** of R if it is a member of some candidate key of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

In Figure 15.1, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation already satisfies 2NF.

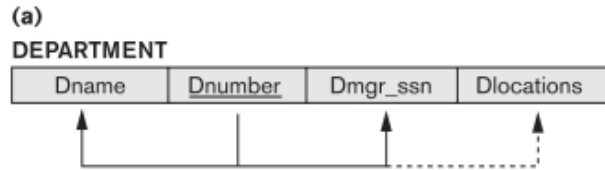
d. First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. In other words, 1NF disallows relations within relations or relations as attribute values within tuples. The only attribute values permitted by 1NF are single **atomic (or indivisible) values**.

Consider the DEPARTMENT relation schema shown in Figure 15.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 15.9(a). We assume that each department can have a number of locations. The DEPARTMENT schema and a sample relation state are shown in Figure 15.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 15.9(b). There are two ways we can look at the Dlocations attribute:

- The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.
- The domain of Dlocations contains sets of values and hence is nonatomic. In this case, $Dnumber \rightarrow Dlocations$ because each set is considered a single member of the attribute domain.

In either case, the DEPARTMENT relation in Figure 15.9 is not in 1NF; in fact, it does not even qualify as a relation.



(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9
 Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation}, as shown in Figure 15.2. A distinct tuple in DEPT_LOCATIONS exists for each location of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 15.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing redundancy in the relation.
3. If a maximum number of values is known for the attribute—for example, if it is known that at most three locations can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing NULL values if most departments have fewer than three locations. It further introduces spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: List the departments that have ‘Bellaire’ as one of their locations in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation within it. Figure 15.10 shows how

the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS (Pnumber, Hours) within each tuple represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ (Ssn, Ename,{PROJS(Pnumber, Hours)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses (). Interestingly, recent trends for supporting complex objects and XML data attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that Ssn is the primary key of the EMP_PROJ relation in Figures 15.10(a) and (b), while Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber. To normalize this into 1NF, we remove the nested relation attributes into a new relation and propagate the primary key into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2, as shown in Figure 15.10(c).

(a)

EMP_PROJ		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP_PROJ			
Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

Figure 15.10

Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

(c)

EMP_PROJ1	
Ssn	Ename

EMP_PROJ2		
Ssn	Pnumber	Hours

This procedure can be applied recursively to a relation with multiple-level nesting to unnest the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (Ss#, {Car_lic#}, {Phone#})

This relation represents the fact that a person has multiple cars and multiple phones. If strategy 2 above is followed, it results in an all-key relation:

PERSON_IN_1NF (Ss#, Car_lic#, Phone#)

To avoid introducing any extraneous relationship between Car_lic# and Phone#, all possible combinations of values are represented for every Ss#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(Ss#, Car_lic#) and P2(Ss#, Phone#).

e. Second Normal Form

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 15.3(b), {Ssn, Pnumber} \rightarrow Hours is a full dependency (neither Ssn \rightarrow Hours nor Pnumber \rightarrow Hours holds). However, the dependency {Ssn, Pnumber} \rightarrow Ename is partial because Ssn \rightarrow Ename holds.

A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 15.3(b) is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure 15.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

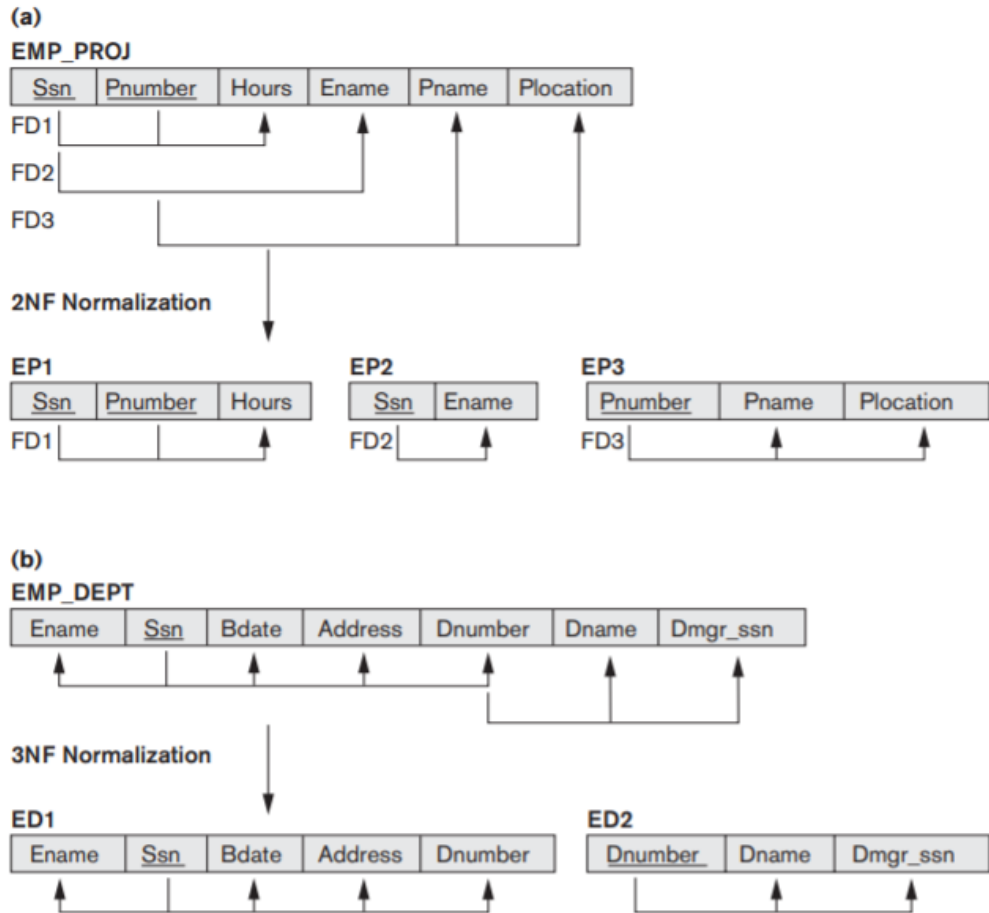


Figure 15.11
 Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

f. Third Normal Form

Third normal form (3NF) is based on the concept of transitive dependency. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R , and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency $Ssn \rightarrow Dmgr_ssn$ is transitive through $Dnumber$ in EMP_DEPT in Figure 15.3(a), because both the dependencies $Ssn \rightarrow Dnumber$ and $Dnumber \rightarrow Dmgr_ssn$ hold and $Dnumber$ is neither a key itself nor a subset of the key of EMP_DEPT . Intuitively, we can see that the dependency of $Dmgr_ssn$ on $Dnumber$ is undesirable in EMP_DEPT since $Dnumber$ is not a key of EMP_DEPT .

According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 15.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of $Dmgr_ssn$ (and also $Dname$) on Ssn via $Dnumber$. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas $ED1$ and $ED2$ shown in Figure 15.11(b). Intuitively, we see that $ED1$ and $ED2$ represent independent entity facts about employees and departments. A NATURAL JOIN operation on $ED1$ and $ED2$ will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a problematic FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 15.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding remedy or normalization performed to achieve the normal form.

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

iv. General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies. The steps for normalization into 3NF relations disallow partial and transitive dependencies on the primary key. The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take all candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of prime attribute, an attribute that is part of any candidate key will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered with respect to all candidate keys of a relation.

a. General Definition of Second Normal Form

A relation schema R is in **second normal form (2NF)** if every nonprime attribute A in R is not partially dependent on any key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 15.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

Based on the two candidate keys Property_id\# and $\{\text{County_name}, \text{Lot\#}\}$, the functional dependencies FD1 and FD2 in Figure 15.12(a) hold. We choose Property_id\# as the primary key, so it is underlined in Figure 15.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: $\text{County_name} \rightarrow \text{Tax_rate}$

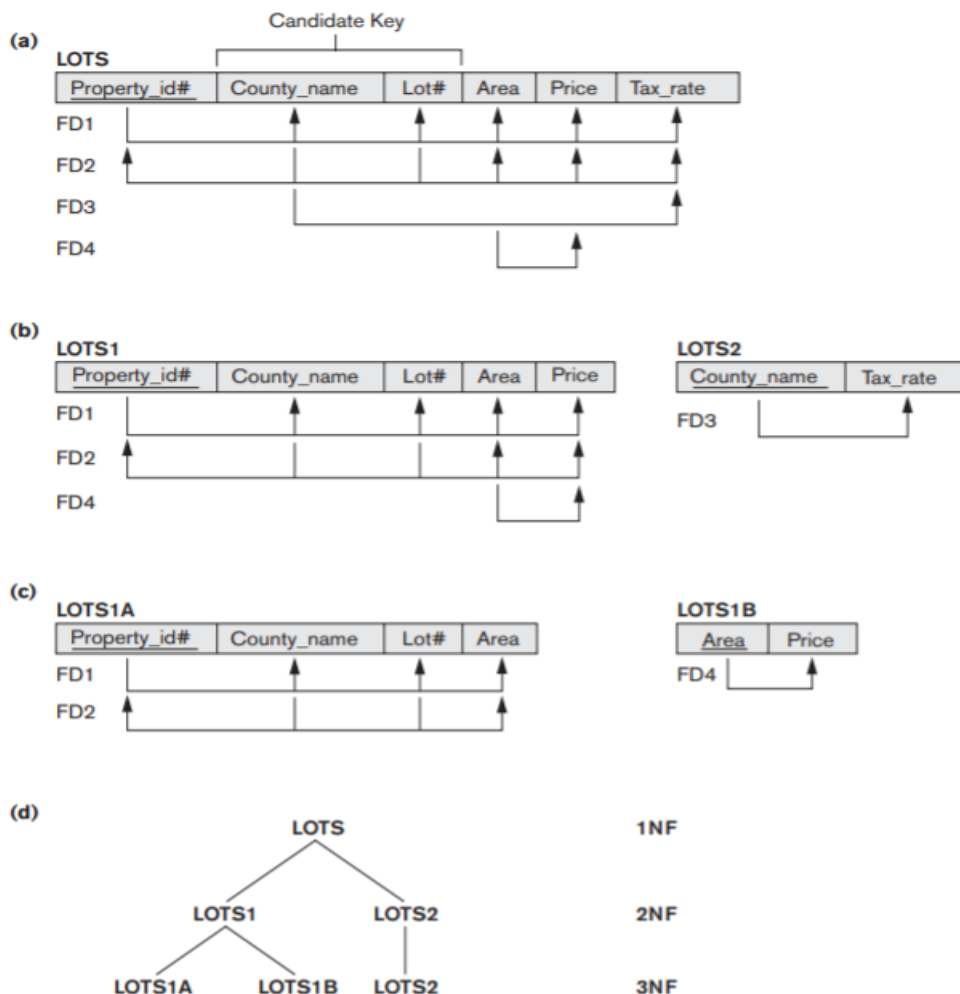
FD4: $\text{Area} \rightarrow \text{Price}$

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key $\{\text{County_name}, \text{Lot\#}\}$, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 15.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

Figure 15.12

Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



b. General Definition of Third Normal Form

A relation schema R is in **third normal form (3NF)** if, whenever a nontrivial functional dependency $X \rightarrow A$ holds in R , either (a) X is a superkey of R , or (b) A is a prime attribute of R .

According to this definition, LOTS2 (Figure 15.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the lefthand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.
- This general definition can be applied directly to test whether a relation schema is in 3NF; it does not have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that both FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed in any order.

c. Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that does not meet either condition—meaning that it violates both conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a general alternative definition of 3NF as follows:

A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- It is fully functionally dependent on every key of R .
- It is nontransitively dependent on every key of R .

v. Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 15.12(a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area \rightarrow County_name. If we add

this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation R(Area, County_name), since there are only 16 possible Area values (see Figure 15.13). This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a stronger normal form that would disallow LOTS1A and suggest the need for decomposing it.

A relation schema R is in **BCNF** if whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

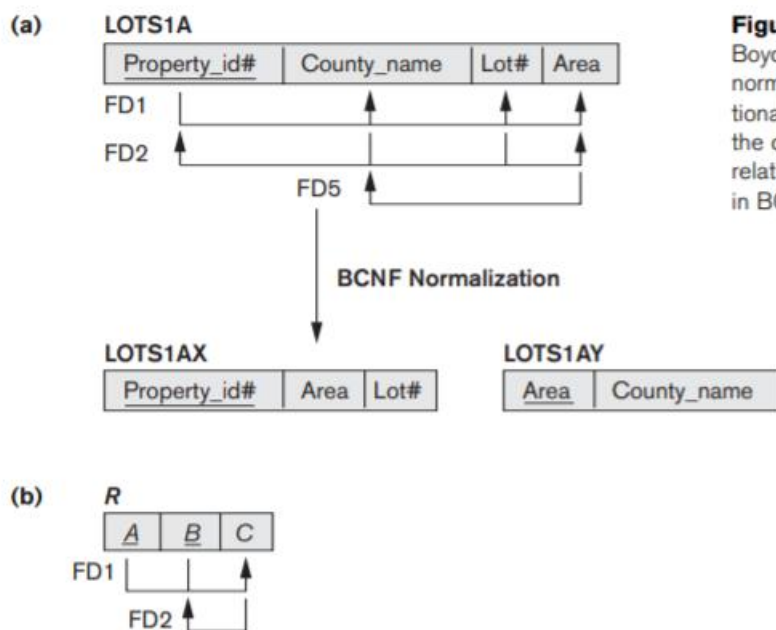


Figure 15.13 Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema R with X not being a superkey and A being a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in Figure 15.13(b) illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 15.14, which shows a relation TEACH with the following dependencies:

- FD1: {Student, Course} → Instructor
- FD2: 12 Instructor → Course

Figure 15.14

A relation TEACH that is in 3NF but not BCNF.

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 15.13(b), with Student as A, Course as B, and Instructor as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. {Student, Instructor} and {Student, Course}.
2. {Course, Instructor} and {Course, Student}.
3. {Instructor, Course} and {Instructor, Student}.

All three decompositions lose the functional dependency FD1. The desirable decomposition of those just shown is 3 because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (or lossless) is discussed under Property NJB. In general, a relation not in BCNF should be decomposed so as to meet this property. We make sure that we meet this property, because nonadditive decomposition is a must during normalization. We may have to possibly forgo the preservation of all functional dependencies in the decomposed relations, as is the case in this example.

vi. Database Stored Procedures and Functions

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client machine, or more likely at the application server computer in the middle-tier of a three-tier client-server architecture. In either case, the machine where the program is executing is different from the machine on which the database server—and the main part of the DBMS software package—is located. Although this is suitable for many applications, it is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database **stored procedures**, although they can be functions or procedures. The term used in the SQL standard for stored procedures is **persistent stored modules** because these programs are stored persistently by the DBMS, similarly to the persistent data stored by the DBMS.

Stored procedures are useful in the following circumstances:

- If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
- Executing a program at the server can reduce data transfer and communication cost between the client and server in certain situations.

- These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users. Additionally, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can be made of simple SQL commands such as retrievals and updates. The general form of declaring stored procedures is as follows:

```
CREATE PROCEDURE <procedure name> (<parameters>)  
<local declarations>  
<procedure body>;
```

The parameters and local declarations are optional, and are specified only if needed. For declaring a function, a return type is necessary, so the declaration form is

```
CREATE FUNCTION <function name> (<parameters>)  
RETURNS <return type>  
<local declarations>  
<function body>;
```

If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language as well as a file name where the program code is stored. For example, the following format can be used:

```
CREATE PROCEDURE <procedure name> (<parameters>)  
LANGUAGE <programming language name>  
EXTERNAL NAME <file path name>;
```

In general, each parameter should have a **parameter type** that is one of the SQL data types. Each parameter should also have a **parameter mode**, which is one of IN, OUT, or INOUT. These correspond to parameters whose values are input only, output (returned) only, or both input and output, respectively.

Because the procedures and functions are stored persistently by the DBMS, it should be possible to call them from the various SQL interfaces and programming techniques. The **CALL statement** in the SQL standard can be used to invoke a stored procedure—either from an interactive interface or from embedded SQL or SQLJ. The format of the statement is as follows:

```
CALL <procedure or function name> (<argument list>);
```

If this statement is called from JDBC, it should be assigned to a statement object of type *CallableStatement*.

vii. Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is

to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific stored procedure or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL. Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database. Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

```
R5: CREATE TRIGGER SALARY_VIOLATION
  BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
  ON EMPLOYEE
  FOR EACH ROW
  WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
    WHERE SSN = NEW.SUPERVISOR_SSN ) )
  INFORM_SUPERVISOR (NEW.Supervisor_ssn, NEW.Ssn );
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.

Let us consider some examples to illustrate these concepts. The examples are based on a much simplified variation of the COMPANY database application and is shown in Figure 26.1, with each employee having a name (Name), Social Security number (Ssn), salary (Salary), department to which they are currently assigned (Dno, a foreign key to DEPARTMENT), and a direct supervisor (Supervisor_ssn, a (recursive) foreign key to EMPLOYEE). For this example, we assume that NULL is allowed for Dno, indicating that an employee may be temporarily unassigned to any department. Each department has a name (Dname), number (Dno), the total salary of all employees assigned to the department (Total_sal), and a manager (Manager_ssn, which is a foreign key to EMPLOYEE).

Figure 26.1
A simplified COMPANY
database used for active
rule examples.

EMPLOYEE				
Name	<u>Ssn</u>	Salary	Dno	Supervisor_ssn

DEPARTMENT			
Dname	<u>Dno</u>	Total_sal	Manager_ssn

Notice that the Total_sal attribute is really a derived attribute, whose value should be the sum of the salaries of all employees who are assigned to the particular department. Maintaining the correct value of such a derived attribute can be done via an active rule. First we have to determine the **events** that may cause a change in the value of Total_sal, which are as follows:

1. Inserting (one or more) new employee tuples
2. Changing the salary of (one or more) existing employees
3. Changing the assignment of existing employees from one department to another
4. Deleting (one or more) employee tuples

In the case of event 1, we only need to recompute Total_sal if the new employee is immediately assigned to a department—that is, if the value of the Dno attribute for the new employee tuple is not NULL (assuming NULL is allowed for Dno). Hence, this would be the **condition** to be checked. A similar condition could be checked for event 2 (and 4) to determine whether the employee whose salary is changed (or who is being deleted) is currently assigned to a department. For event 3, we will always execute an action to maintain the value of Total_sal correctly, so no condition is needed (the action is always executed).

The **action** for events 1, 2, and 4 is to automatically update the value of Total_sal for the employee's department to reflect the newly inserted, updated, or deleted employee's salary. In the case of event 3, a twofold action is needed: one to update the Total_sal of the employee's old department and the other to update the Total_sal of the employee's new department.

The four active rules (or triggers) R1, R2, R3, and R4—corresponding to the above situation—can be specified in the notation of the Oracle DBMS as shown in Figure 26.2(a). Let us consider rule R1 to illustrate the syntax of creating triggers in Oracle.

```

(a) R1: CREATE TRIGGER Total_sal1
      AFTER INSERT ON EMPLOYEE
      FOR EACH ROW
      WHEN ( NEW.Dno IS NOT NULL )
      UPDATE DEPARTMENT
      SET Total_sal = Total_sal + NEW.Salary
      WHERE Dno = NEW.Dno;

R2: CREATE TRIGGER Total_sal2
      AFTER UPDATE OF Salary ON EMPLOYEE
      FOR EACH ROW
      WHEN ( NEW.Dno IS NOT NULL )
      UPDATE DEPARTMENT
      SET Total_sal = Total_sal + NEW.Salary - OLD.Salary
      WHERE Dno = NEW.Dno;

R3: CREATE TRIGGER Total_sal3
      AFTER UPDATE OF Dno ON EMPLOYEE
      FOR EACH ROW
      BEGIN
      UPDATE DEPARTMENT
      SET Total_sal = Total_sal + NEW.Salary
      WHERE Dno = NEW.Dno;
      UPDATE DEPARTMENT
      SET Total_sal = Total_sal - OLD.Salary
      WHERE Dno = OLD.Dno;
      END;

R4: CREATE TRIGGER Total_sal4
      AFTER DELETE ON EMPLOYEE
      FOR EACH ROW
      WHEN ( OLD.Dno IS NOT NULL )
      UPDATE DEPARTMENT
      SET Total_sal = Total_sal - OLD.Salary
      WHERE Dno = OLD.Dno;

(b) R5: CREATE TRIGGER Inform_supervisor1
      BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn
      ON EMPLOYEE
      FOR EACH ROW
      WHEN ( NEW.Salary > ( SELECT Salary FROM EMPLOYEE
                           WHERE Ssn = NEW.Supervisor_ssn ) )
      inform_supervisor(NEW.Supervisor_ssn, NEW.Ssn );

```

Figure 26.2

Specifying active rules as triggers in Oracle notation. (a) Triggers for automatically maintaining the consistency of Total_sal of DEPARTMENT. (b) Trigger for comparing an employee's salary with that of his or her supervisor.

The CREATE TRIGGER statement specifies a trigger (or active rule) name— Total_sal1 for R1. The AFTER clause specifies that the rule will be triggered after the events that trigger the rule occur. The triggering events—an insert of a new employee in this example—are specified following the AFTER keyword.

The ON clause specifies the relation on which the rule is specified—EMPLOYEE for R1. The optional keywords FOR EACH ROW specify that the rule will be triggered once for each row that is affected by the triggering event.

The optional WHEN clause is used to specify any conditions that need to be checked after the rule is triggered, but before the action is executed. Finally, the action(s) to be taken is (are) specified as a PL/SQL block, which typically contains one or more SQL statements or calls to execute external procedures.

The four triggers (active rules) R1, R2, R3, and R4 illustrate a number of features of active rules. First, the basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. They are specified by the keywords INSERT, DELETE, and UPDATE in Oracle notation. In the case of UPDATE, one may specify the attributes to be updated—for example, by writing **UPDATE OF** Salary, Dno. Second, the rule designer needs to have a way to refer to the tuples that have been inserted, deleted, or modified by the triggering event. The keywords **NEW** and **OLD** are used in Oracle notation; NEW is used to refer to a newly inserted or newly updated tuple, whereas OLD is used to refer to a deleted tuple or to a tuple before it was updated.

Thus, rule R1 is triggered after an INSERT operation is applied to the EMPLOYEE relation. In R1, the condition (**NEW.Dno IS NOT NULL**) is checked, and if it evaluates to true, meaning that the newly inserted employee tuple is related to a department, then the action is executed. The action updates the DEPARTMENT tuple(s) related to the newly inserted employee by adding their salary (**NEW.Salary**) to the Total_sal attribute of their related department.

Rule R2 is similar to R1, but it is triggered by an UPDATE operation that updates the SALARY of an employee rather than by an INSERT. Rule R3 is triggered by an update to the Dno attribute of EMPLOYEE, which signifies changing an employee's assignment from one department to another. There is no condition to check in R3, so the action is executed whenever the triggering event occurs. The action updates both the old department and new department of the reassigned employees by adding their salary to Total_sal of their new department and subtracting their salary from Total_sal of their old department. Note that this should work even if the value of Dno is NULL, because in this case no department will be selected for the rule action.

It is important to note the effect of the optional FOR EACH ROW clause, which signifies that the rule is triggered separately for each tuple. This is known as a **row-level trigger**. If this clause was left out, the trigger would be known as a **statement-level trigger** and would be triggered once for each triggering statement. To see the difference, consider the following update operation, which gives a 10 percent raise to all employees assigned to department 5. This operation would be an event that triggers rule R2:

```
UPDATE EMPLOYEE
SET Salary = 1.1 * Salary
WHERE Dno = 5;
```

Because the above statement could update multiple records, a rule using row-level semantics, such as R2 in Figure 26.2, would be triggered once for each row, whereas a rule using statement-level semantics is triggered only once. The Oracle system allows the user to choose which of the above options is to be used for each rule. Including the optional FOR EACH ROW clause creates a row-level trigger, and leaving it out creates a statement-level trigger. Note that the keywords NEW and OLD can only be used with row-level triggers.

As a second example, suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor. Several events can trigger this rule: inserting a new employee, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external procedure inform_supervisor, which will notify the supervisor. The rule could then be written as in R5 (see Figure 26.2(b)).

Figure 26.3 shows the syntax for specifying some of the main options available in Oracle triggers.

Figure 26.3

A syntax summary for specifying triggers in the Oracle system (main options only).

```
<trigger>          ::= CREATE TRIGGER <trigger name>
                    ( AFTER | BEFORE ) <triggering events> ON <table name>
                    [ FOR EACH ROW ]
                    [ WHEN <condition> ]
                    <trigger actions> ;
<triggering events> ::= <trigger event> { OR <trigger event> }
<trigger event>    ::= INSERT | DELETE | UPDATE [ OF <column name> { , <column name> } ]
<trigger action>  ::= <PL/SQL block>
```
