1.0 Introduction

Databases and database technology have a major impact on the growing use of computers. Databases play a critical role in almost all areas where computers are used, including business, electronic commerce, engineering, medicine, law, education, and library science.

Definition: 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 index address book or you may have stored 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 has the following implicit properties:

A database represents some aspect of the real world, sometimes called the miniworld or the universe of discourse (UoD). Changes to the miniworld are reflected in the database. A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.

A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested. In other words, a database has some source from which data is derived, some degree of interaction with events in the real world, an audience that is actively interested in its contents. The end users of a database may perform business transactions (for example, a customer buys a camera) or events may happen (for example, an employee has a baby) that cause the information in the database to change. In order for a database to be accurate and reliable at all times, it must be a true reflection of the miniworld that it represents; changes must be reflected in the database as soon as possible.

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 and descriptive information is also stored in the database in the form of a database catalog 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.

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 and software malfunction (crashes) and security protection against unauthorised 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. To complete our initial definitions, we will call the database and DBMS software together a database system.

1.1 An Example

Let us consider a simple example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.1 shows the database structure and some sample data for such a database. The database is organised as five files, each of which stores data records of the same type. The *STUDENT* file stores data on each student, the COURSE file stores data on each course, the SECTION file stores data on each section of a course, the GRADE_REPORT file stores the grades that students receives in the various sections they have completed, and the PREREQUISITE file stores the prerequisites of each course.

To define this database, we must specify the structure of the records of each file by specifying the different types of data elements to be stored in each record. In Figure 1.1, each STUDENT record includes data to represent the student's Name, Student_number, Class and Major (such as 'MATH' and computer science or 'CS'); each COURSE record includes data to represent the Course_name, Course_number, Credit_hours, and Department (the department that offers the course); and so on. We must also specify a data type for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, Student_number of STUDENT is an integer, and Grade of GRADE_REPORT is a single character format, the set {'A', 'B','C','D', 'E','F','I'}. We may also use a coding scheme to represent the values of a data item. For example, in Figure 1.1, we represent the Class of a

STUDENT as 1 for First Year, 2 for Second Year, 3 for Third Year, 4 for Fourth Year, and 5 for graduate student.

To construct the UNIVERSITY database, we store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file. Notice that records in the various files may be related. For example, the record for *Smith* in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith's grades in two sections. Similarly, each record in the PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite. Most medium-size and large databases include many types of records and have many relationships among the records.

COURSE

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

SECTION

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

GRADE_REPORT

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.1
A database that stores student and course information

1.3 Characteristics of the Database Approach

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

In the database approach, a single repository of data is maintained 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.

1.3.1 Self-Describing Nature of a Database System

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

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of the 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. We call this property program-data independence.

1.3.3 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views.

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.

1.4 A Short Database History

Ancient to modern: The origins go back to libraries, governmental, business, and medical records. There is a very long history of information storage, indexing, and retrieval. Good design principles goes way back and lots is known now about how to make good designs that lead to better reliability and performance.

1960's: Computers become cost effective for private companies along with increasing storage capability of computers. Two main data models were developed: network model (CODASYL) and hierarchical (IMS). Access to database is through low-level pointer operations linking records. Storage details depended on the type of data to be stored. Thus adding an extra field to

your database requires rewriting the underlying access/modification scheme. Emphasis was on records to be processed, not overall structure of the system. A user would need to know the physical structure of the database in order to query for information. One major commercial success was SABRE system from IBM and American Airlines.

1970-72: E.F. Codd proposed relational model for databases in a landmark paper on how to think about databases. He disconnects the schema (logical organization) of a database from the physical storage methods. This system has been standard ever since.

1970's: Several camps of proponents argue about merits of these competing systems while the theory of databases leads to mainstream research projects. Two main prototypes for relational systems were developed during 1974-77. These provide nice example of how theory leads to best practice.
Ingres: Developed at UCB. This ultimately led to Ingres Corp., Sybase, MS SQL Server, Britton-Lee, Wang's PACE. This system used QUEL as query language.
System R: Developed at IBM San Jose and led to IBM's SQL/DS & DB2, Oracle, HP's Allbase, Tandem's Non-Stop SQL. This system used SEQUEL as query language. The term Relational Database Management System (RDBMS) is coined during this period.

1976: P. Chen proposed the Entity-Relationship (ER) model for database design giving yet another important insight into conceptual data models. Such higher level modelling allows the designer to concentrate on the use of data instead of logical table structure.

Early 1980's: Commercialization of relational systems begins as a boom in computer purchasing fuels DB market for business.

Mid-1980's: SQL (Structured Query Language) becomes "intergalactic standard". DB2 becomes IBM's flagship product. Network and hierarchical models fade into the background, with essentially no development of these systems today but some legacy systems are still in use. Development of the IBM PC gives rise to many DB companies and products such as RIM, RBASE 5000, PARADOX, OS/2 Database Manager, Dbase III, IV (later Foxbase, even later Visual FoxPro), Watcom SQL.

Early 1990's: An industry shakeout begins with fewer surviving companies offering increasingly complex products at higher prices. Much development during this period centers

on client tools for application development such as PowerBuilder (Sybase), Oracle Developer, VB (Microsoft), etc. Client-server model for computing becomes the norm for future business decisions. Development of personal productivity tools such as Excel/Access (MS) and ODBC. This also marks the beginning of Object Database Management Systems (ODBMS) prototypes.

Mid-1990's: Kaboom! The usable Internet/WWW appears. A mad scramble ensues to allow remote access to computer systems with legacy data. Client-server frenzy reaches the desktop of average users with little patience for complexity while Web/DB grows exponentially.

Late-1990's: The large investment in Internet companies fuels tools market boom for Web/Internet/DB connectors. Active Server Pages, Front Page, Java Servlets, JDBC, Enterprise Java Beans, ColdFusion, Dream Weaver, Oracle Developer 2000, etc are examples of such offerings. Open source solution come online with widespread use of gcc, cgi, Apache, MySQL, etc. Online Transaction processing (OLTP) and online analytic processing (OLAP) comes of age with many merchants using point-of-sale (POS) technology on a daily basis.

Early 21st century: Decline of the Internet industry as a whole but solid growth of DB applications continues. More interactive applications appear with use of PDAs, POS transactions, consolidation of vendors, etc. Three main (western) companies predominate in the large DB market: IBM (buys Informix), Microsoft, and Oracle.

Future trends: Huge (terabyte) systems are appearing and will require novel means of handling and analyzing data. Large science databases such as genome project, geological, national security, and space exploration data. Clickstream analysis is happening now. Data mining, data warehousing, data marts are a commonly used technique today. More of this in the future without a doubt. Smart/personalized shopping using purchase history, time of day, etc.

Successors to SQL (and perhaps RDBMS) will be emerging in the future. Most attempts to standardize SQL successors has not been successful. SQL92, SQL2, SQL3 are still underpowered and more extensions are hard to agree upon. Most likely this will be overtaken by XML and other emerging techniques. XML with Java for databases is the current poster child of the "next great thing". Check in tomorrow to see what else is news.

Mobile database use is a product now coming to market in various ways. Distributed transaction processing is becoming the norm for business planning in many arenas.

Probably there will be a continuing shakeout in the RDBMS market. Linux with Apache supporting mySQL (or even Oracle) on relatively cheap hardware is a major threat to high cost legacy systems of Oracle and DB2 so these have begun pre-emptive projects to hold onto their customers.

Object Oriented Everything, including databases, seems to be always on the verge to sweeping everything before it. Object Database Management Group (ODMG) standards are proposed and accepted and maybe something comes from that.
Ethical and security issues tend to be diminished at times but always come back. Should you be able to consult a database of the medical records/genetic makeup of a prospective employee? Should you be able to screen a prospective partner/lover for genetic diseases? Should amazon.com keep track of your book purchasing? Should there be a national database of convicted sex offenders/violent criminals/drug traffickers? Who is allowed to do Web tracking? How many times in the last six months did you visit a particular sex chat room/porn site/political satire site? Who should be able to keep or view such data? Who makes these decisions?

2.0 Database System Concepts And Architecture

In this chapter we present the terminology and basic concepts that will be used throughout this course.

2.1 Data Models, Schemas, and Instances

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

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorise 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 in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users. Between these two extremes is a class of **representational (or implementation) data models,** which provide concepts that may be understood by end users but that are not too far removed from the way data is organised within the computer. Representational data models hide some details of data storage but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities, for example, a works-on relationship between an employee and a project.

2.1.2 Schemas, Instances, and Database State

In any data model, it is important to distinguish between the description of the database and the database itself. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently. Most data models have certain conventions for displaying schemas as diagrams. A displayed schema is called a **schema diagram.** Figure 2.1 shows a schema diagram for the database shown in Figure 1.1; 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.**

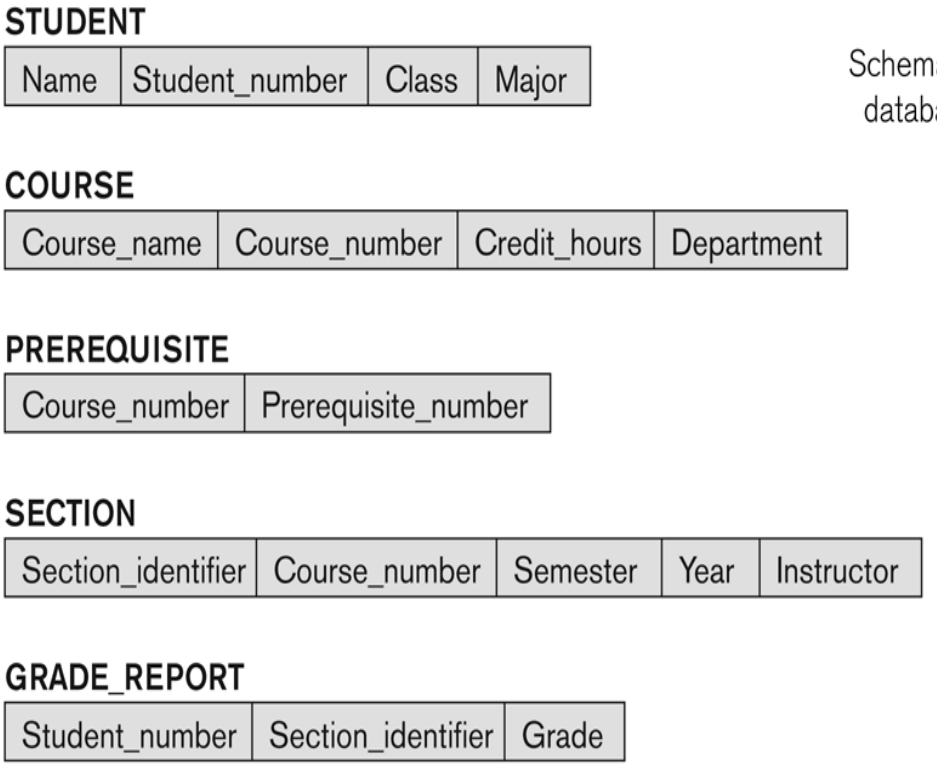


Figure 2.1
Schema diagram for the database in Figure 1.1

A schema diagram displays only some aspects of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example Figure 2.1 shows neither the data type of each data item nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as *students majoring in computer science must take CSM 388 before the end of the third year* is quite difficult to represent.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.1 changes every time we add a student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot.** It is also called the

current set of **occurrences or instances** in the 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. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. Specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints – also called the **meta-data –** in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications and 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 case, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level data model.

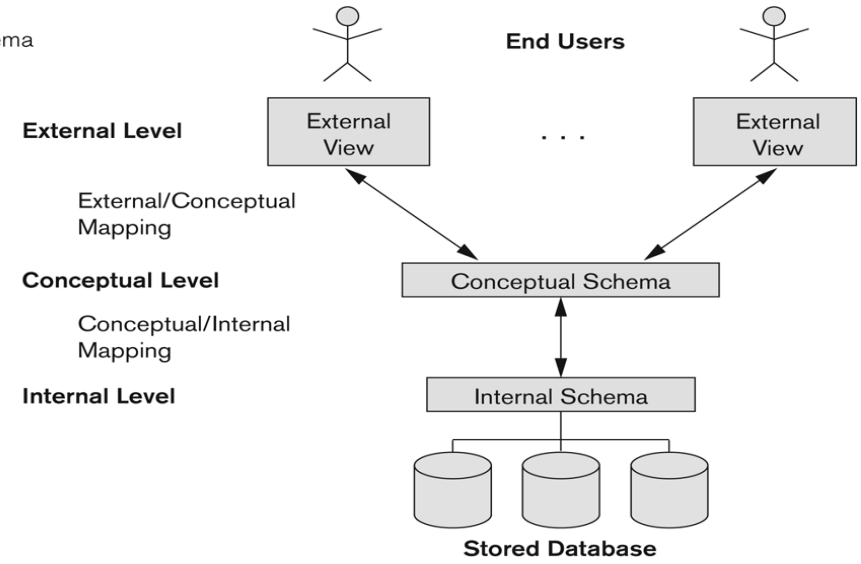


Figure 2.2
The three-schema architecture.

Notice that the three schemas are only descriptions of data; the stored data that actually exists is at the physical level. In a DBMS based on the three-schema architecture, each use group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The process of transforming requests and results between levels are called **mappings.**

2.2 Centralised and Client/Server Architectures for DBMS

2.2.1 Centralised DBMS Architecture

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 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 communication 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 **centralised** DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Figure 2.3 illustrates the physical components in a centralised architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

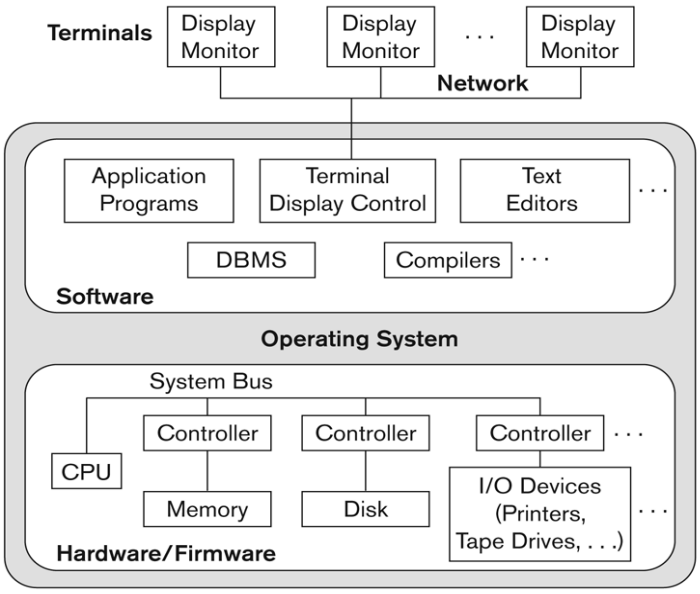


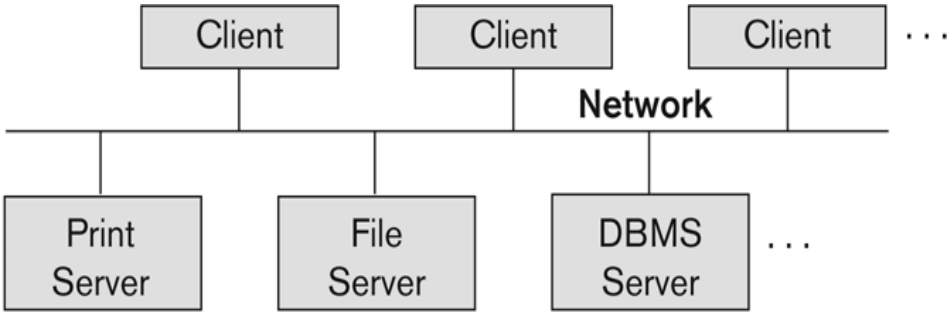
Figure 2.3
A physical centralized architecture.

2.2.2 Basic Client/Server Architectures

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, and other equipment are connected via a network. The idea is to define **specialised 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 **print server** by being connected to various printers; therefore all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into this category of specialised servers. In this way the resources provided by the specialised servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to software, with specialised programs – such as DBMS or a CAD (computer aided design) package – being stored on

specific server machines and being made accessible to multiple clients. Figure 2.4 illustrates client/server architecture at the logical level.

Figure 2..4
Logical two-tier client/server architecture.



2.2.3 Two-Tier Client/Server Architectures for DBMSs

The client/server architecture is increasingly being incorporated into commercial DBMS packages. In a relational database management system (RDBMS), many of which started as centralised systems, the system components that were first moved to the client side were their user interfaces 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 remained on the server.

In such a client/server architecture, 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. 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 or display the results as needed.

The architecture describe here is called a **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 server, leading to the three-tier architecture.

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

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.5

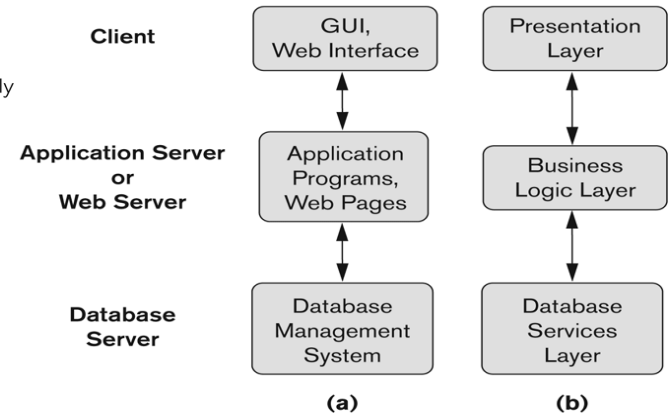


Figure 2.5
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

This intermediate layer or **middle tier** is sometimes called the **application server** and sometimes the **Web server**, depending on the application. This server plays an intermediary role by 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 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.

Figure 2.5(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. If the bottom layer is split into two layers (a Web server and a database server), then this becomes a four-tier architecture. It is customary 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.

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.

3.0 Data Modelling Using the Entity-Relationship (ER) Model

Conceptual modelling is a very important phase in designing a successful database application. Generally, the term **database application** refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement updates corresponding to customer deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilising forms and menus for the end-user application – the bank tellers, in this example.

In this chapter, we present the modelling concepts of the **Entity-Relationship (ER) model**, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present diagrammatic notation associated with the ER model, known as **ER diagrams.**

3.1 Using High-Level Conceptual Data Models for Database Design

Figure 3.1 shows a simplified description of the database 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 consists of user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates.

Once all the requirement have been collected and analysed, 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 do not include implementation details, they are usually easier to understand and

can be used to communicate with non-technical users. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage details. Consequently it is easier for them to create a good conceptual database design.

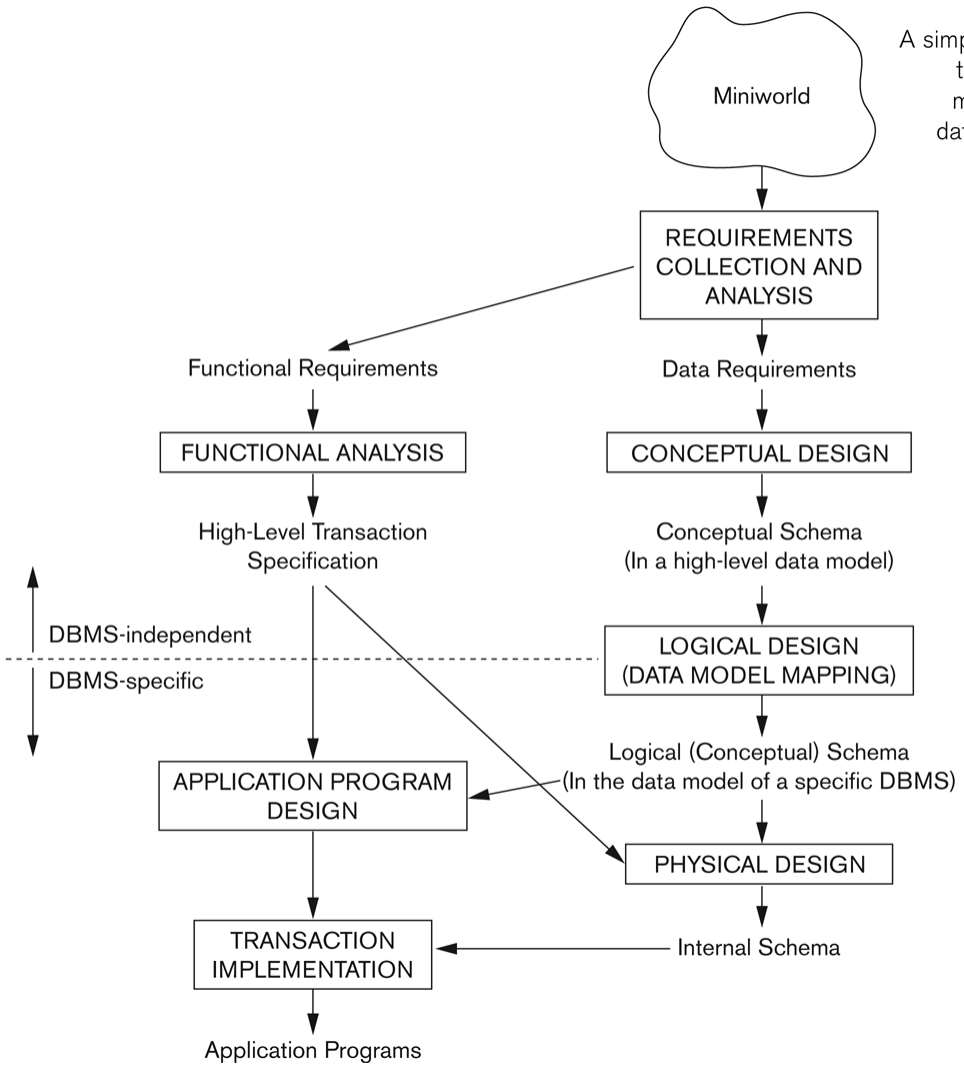


Figure 3.1
A simplified diagram to illustrate the main phases of database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user operations identified during functional analysis. This also serve 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 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.

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

We present only the basic ER model concepts for conceptual schema design in the chapter.

3.2 An Example Database Applicaton

In this section we describe an example 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 modelling 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 to be represented in the database.

- The company is organised 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, and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.

- 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.

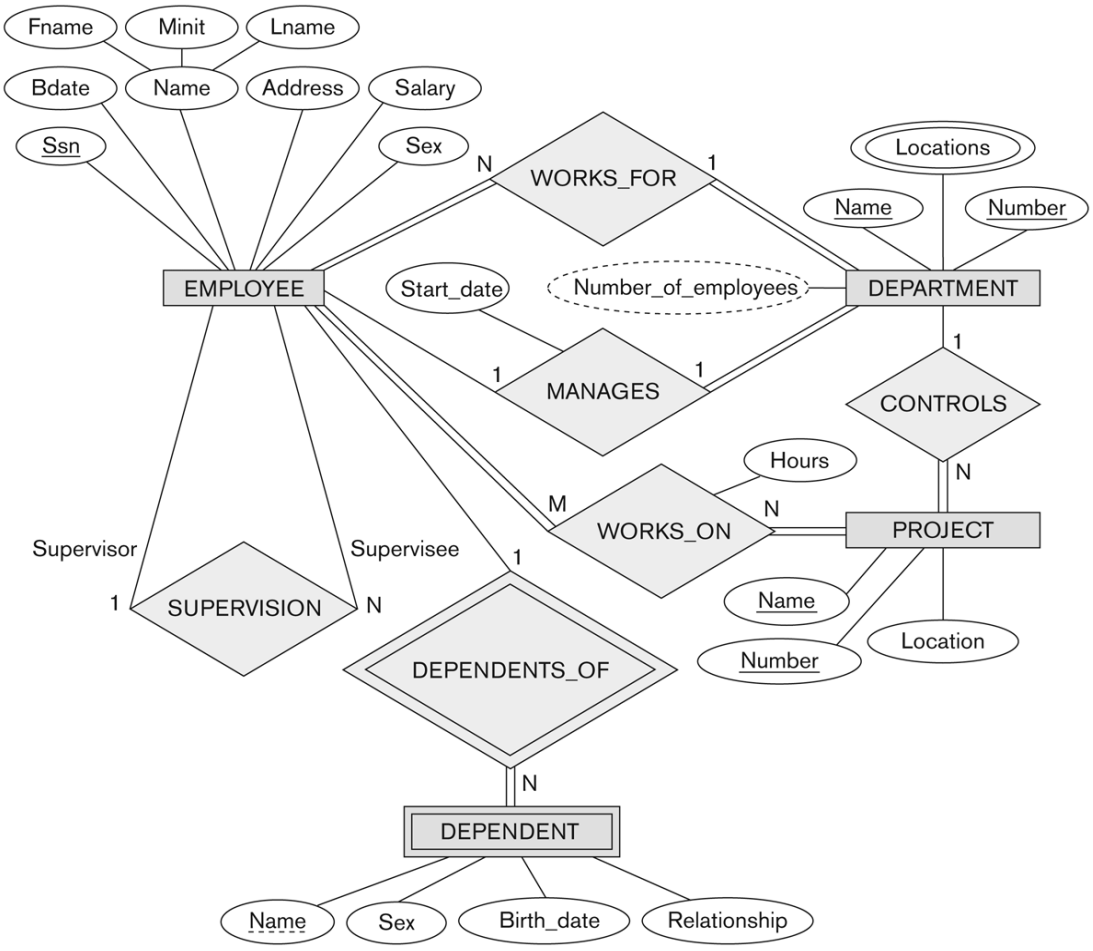


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Figure 3.2 shows how the schema for this application can be displayed by means of the graphical notation known as **ER diagrams.** This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements – and explain the ER diagrammatic notation – as we introduce the ER model concepts.

3.3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as *entities, relationships, and attributes.* In Section 3.3.1 we introduce the concepts of entities and their attributes. We discuss entity types and key attributes in Section 3.3.2. Then, in Section 3.3.3, we specify the initial conceptual design of the entity types for the COMPANY database. Relationships are described in Section 3.4.

3.3.1 Entities and Attributes

Entities and Their Atributes. The basic object that 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 example, 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. For example, The EMPLOYEE entity could have 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 could have 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 introduce 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 with independent meanings. For example, the Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State, and Zip, with the values '2311 Kirby,' 'Houston,' 'Texas,' '77001.' Attributes that are not divisible are called **simple** or **atomic attributes.** Composite attributes can for a hierarchy; for example Street_address can be subdivided into three simple attributes: Number, Street, and Apartment_number as shown in **Figure** 3.3 . The value of a composite attribute is the concatenation of the values of its constituent 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 refer 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.

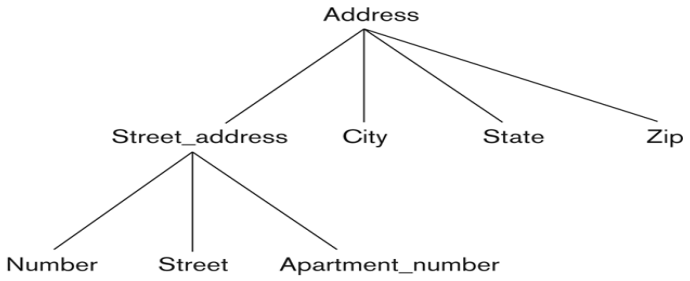


Figure 3.3

Figure 3.4
A hierarchy of composite attributes.

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 single-valued attribute of a person. 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 example, a Colours attribute for a car, or a College_degrees attribute for a person. Cars with one colour have a single value, whereas two-tone cars have two 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 persons 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 Colours attribute of a car may have between one and three values, if we assume that a car can have three colours 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 **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 persons 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'.

3.3.2 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 defines in the database is described by its name and attributes. **Figure 3.4** shows two entity types: EMPLOYEE and COMPANY, and a list of 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 (See Figure 3.2).

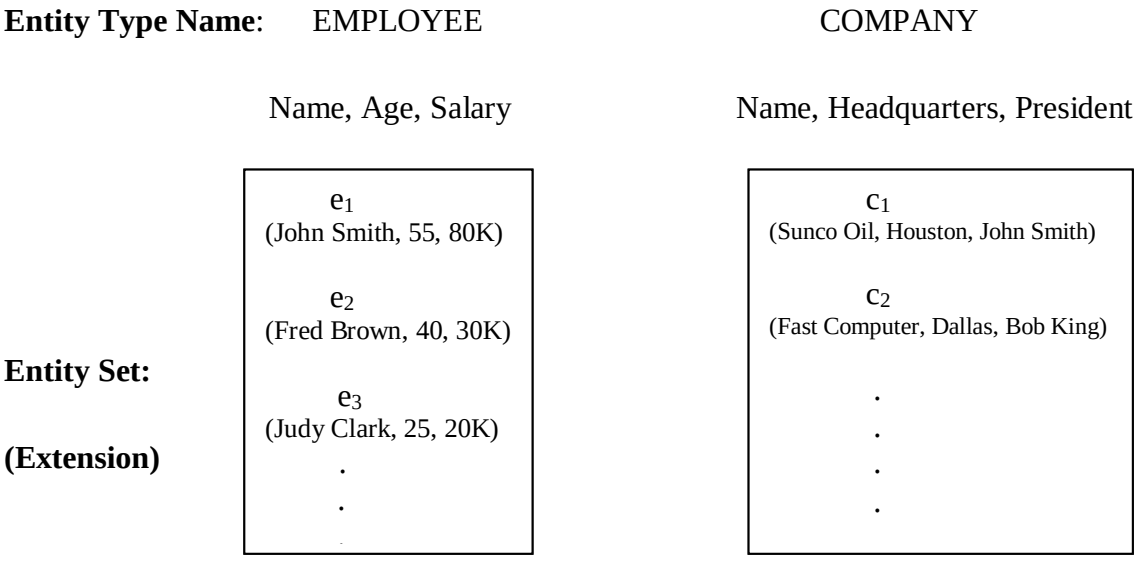


Figure 3.4 : Two entity types, EMPLOYEE and COMPANY, and some member entities of each

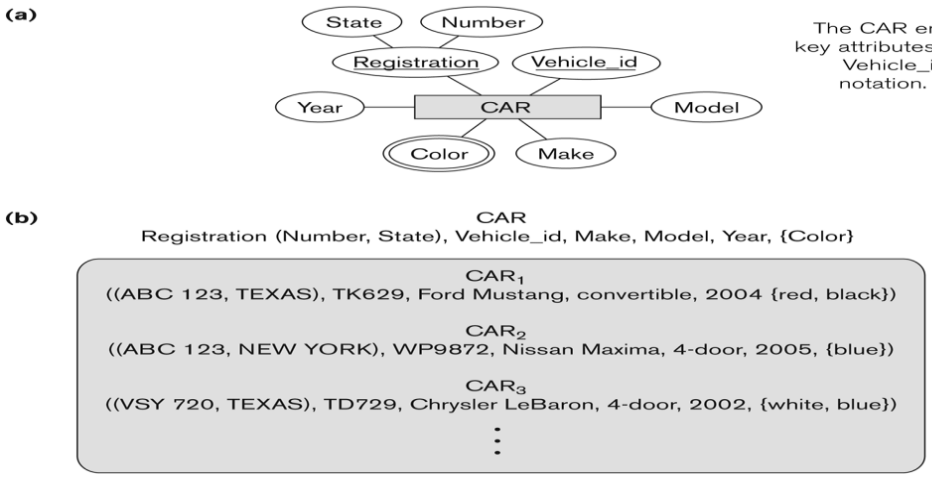
Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**,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

attribute is a key of the COMPANY entity in Figure 3.4 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.

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. Some entities have more than one key attributes. For example, each of the Vehicle_id and Registration attributes of the entity type CAR Figure 3.5(a) 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.

Figure 3.5

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



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. In Figure 3.4, if the range of ages allowed for employees is between 16 and 70, we can specify the value of 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. Value sets are not displayed in ER diagrams. Value sets 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 date, time, and other concepts are also employed.

3.3.3 Initial Conceptual Design of the COMPANY Database

We can define the entity types for the COMPANY database, based on the requirements described in Section 3.2. After defining several entity types and their attributes here, we refine our design in Section 3.4 after we introduce the concept of a relationship. According to the requirements listed in Section 3.2, we can identify four entity types – one corresponding to each of the four items in the specification(see Figure 3.6) :

- An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attributes. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.

- An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.

- An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name – First_name, Middle_initial, Last_name – or of Address.

- An entity type DEPENDENT with attributes Employee, Dependent_name, Sex ,Birth_date, and Relationship (to the employee).

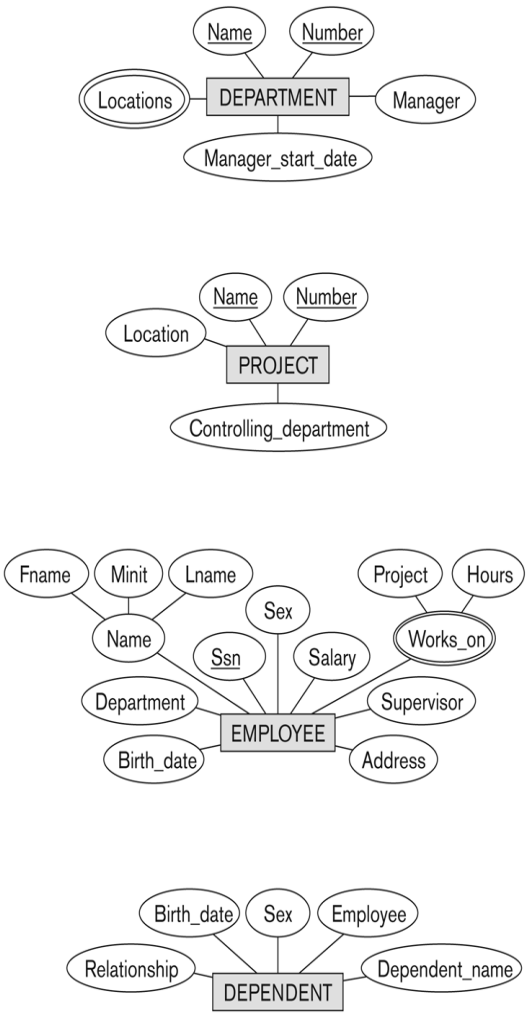


Figure 3.6

Figure 3.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

So far we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic is listed as part of the third requirement in Section 3.2, and it can be represented by a multivalued composite attribute of EMPLOYEE called Works_on with the simple components (Project, Hours). The Name attribute of EMPLOYEE is shown as a composite attribute, presumably after consultation with the users.

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

In Figure 3.6 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manger of DEPARTMENT refers to an employee who manages the department; the attribute Controlling_department of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the

department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships,** which are discussed in this section.

3.4.1 Relationship Types, Sets, and Instances

A **relationship type** R among n entity types E_1, E_2 ,…. ,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 ($e_1, e_2, \ldots, e_n$) , and each entity e_j in r_i is a member of entity type E_j, $1<j<n$. Hence, a relationship type is a mathematical relation on E_1, E_2 ,…. ,E_n; alternatively, it can be defined as a subset of the Cartesian product $E_1 X E_2 X$,…. , $X E_n$. Each of the entity types E_1, E_2 ,…. ,E_n is said to participate in the relationship type R; similarly, each of the individual entities e_1, e_2 ,…. ,e_n is said to participate in the relationship instance $r_i = (e_1, e_2 ,\ldots ,e_n)$.

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 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. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity. Figure 3.7 illustrates this example, where each relationship instance r_i is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in r_i.

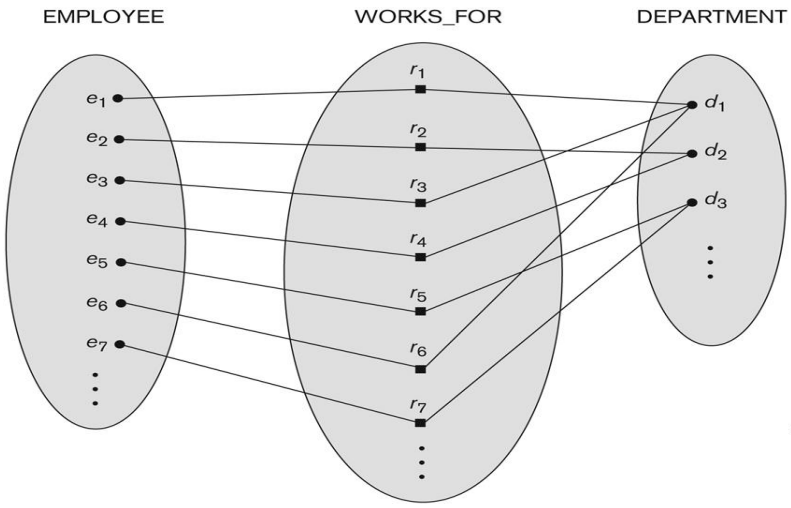


Figure 3.7

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

In the miniworld represented by Figure 3.7, employees e_1, e_3, and e_6 work for department d_i; employees e_2 and e_4 work for department d_2; and employees e_5 and e_5 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.

3.4.2 Relationship Degree, Role Names,

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.**

3.4.3 Constraints on Relationship Types

Relationship types usually have certain constraints that limit the possible combination of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. For example, in Figure 3.7, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of relationship constraints: *cardinality ratio and participation.*

Cardinality Ratios for Binary Relationships. 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 ration 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. 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 (Figure 3.8) is of cardinality ration M:N because the miniworld rule is that an employee can work on several projects and a project can have several employees. Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 3.2.

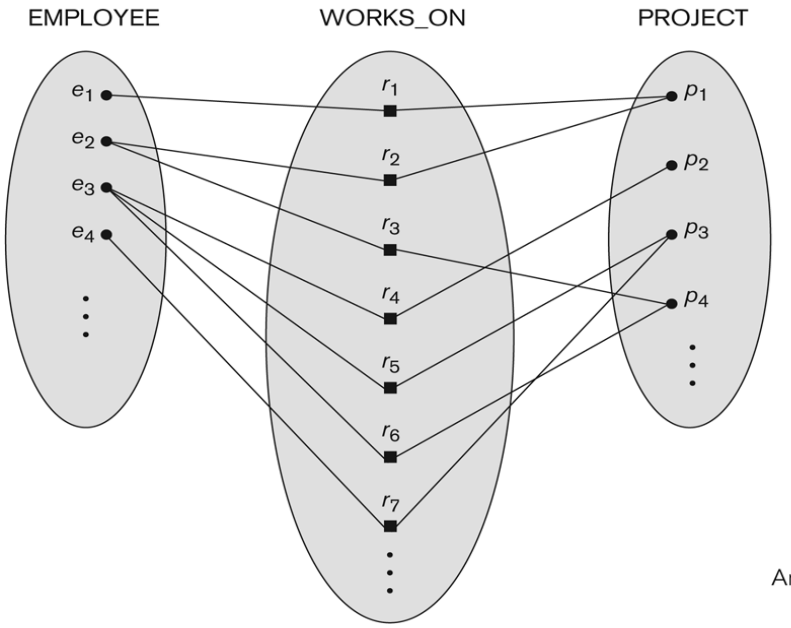


Figure 3.8
An M:N relationship, WORKS_ON.

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 constraints 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 – which we illustrate by example. 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 aWORKS_FOR relationship instance (Figure 3.7). 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.** In the MANAGES relationship 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.

3.5 Refining the ER Design for the COMPANY Database

We can refine the database design of Figure 3.6 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed in Section 3.2. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

- MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.

- WORKS_FOR, a 1: N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.

- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.

- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.

- WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.

- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

4.0 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. When a relation is thought of as a **table** of values, each row in the table represents a collection of related values. We introduced entity types and relationship types as concepts for modelling real-world data in Chapter 3. In the relational model, each row in the table represents a fact that typically corresponds to a real-entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row. For example, the first table of Figure 2.1 is called STUDENT because each row represents facts about a particular student entity. The column name – Name, Student_number, Class, and Major –specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In 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 more precisely.

4.1.1 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 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 domain follow:

- Usa_phone_numbers. The set of ten digit phone number 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.
- Names: The set of character strings that represent names of persons.
- Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.

- Employee_ages. Possible ages of employees of a company; each must be a value between 15 and 80.
- Academic_department_names. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- Academic_department_codes. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

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 Employee_ages is an integer number between 15 and 80. A domain is thus given a name, data type, and format.

A **relation schema** R, denoted by $R(A_1, A_{2...,}A_{n)}$ is made up of a relation name R and a list of attributes $A_1, A_{2...,}A_n$. 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 **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.

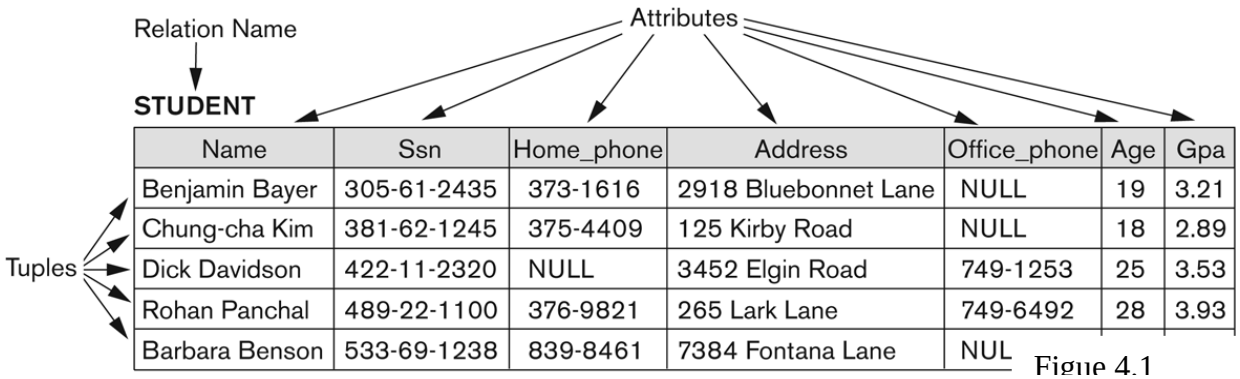
An example of a relation schema for a relation degree seven, which describes university students, is the following:

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)

Figure 4. 1 shows an example of STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity. 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.



Figue 4.1

Figure 5.1
The attributes and tuples of a relation STUDENT.

4.2 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. 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** 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 or explicit constraints.**

3. Constraints that cannot be directly expressed in 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.**

4.2.1 Key Constraints and Constraints on NULL Values

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.

Primary key constraints

A primary key constraint is a unique key with special attributes that make the key the primary access path for the file. Primary key constraints identify a field or set of fields in a database file whose values must be unique across records in the file. A table cannot have more than one set of attributes specified as the primary key, and the columns of a primary key cannot contain null values

For example the attribute {Ssn} is a primary key of STUDENT relation of Figure 4.1. because no two student tuples can have the same value for Ssn.

FOREIGN KEY Constraints

A foreign key (FK) is a column or combination of columns that is used to establish and enforce a link between the data in two tables. You can create a foreign key by defining a FOREIGN KEY constraint when you create or modify a table. Usually a foreign key is a primary key in the one of the tables.

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

4.2.3 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 some 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 $S = \{R_1, R_2, ..., R_m\}$ and a set of **integrity constraints** IC. A **relational database state** DB of S is a set of relation states $DB = \{R_1, R_2, ..., R_m\}$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. Figure 4.2 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPARTMENT}. The underlined attributes represent primary keys. Figure 4.3 shows a relational database state corresponding to the COMPANY schema.

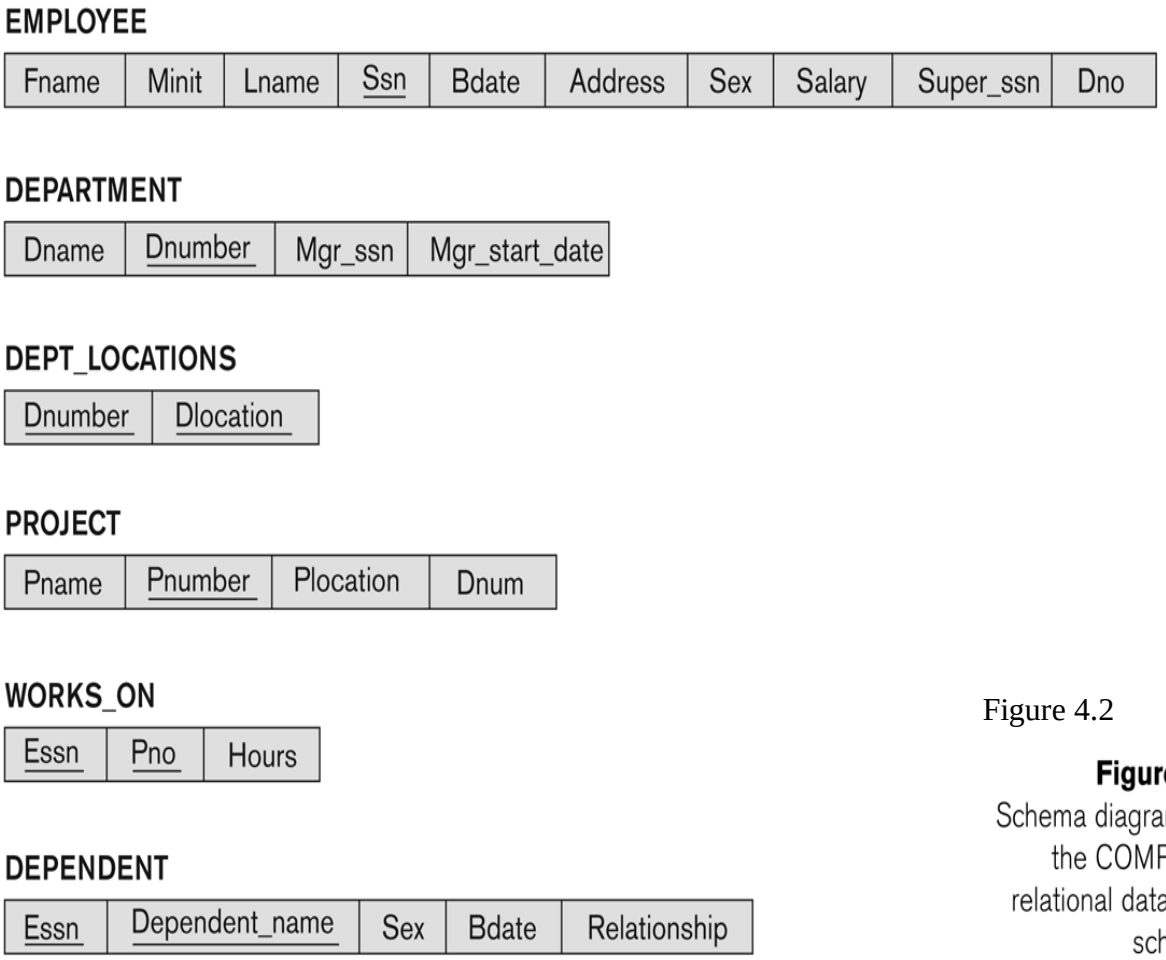


Figure 4.2

Figure 5.5
Schema diagram for the COMPANY relational database schema.

In Figure 4.2 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 could have two attributes that share the same name but represent different real-world concepts – project names and department names.

Figure 4.3

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

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

DEPENDENT

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

4.2.4 Entity 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 might not be able to distinguish them if we tried 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 Figure 4.3, 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.

The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R1 that **references** relation R_2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2 ; attributes FK are said to **reference or refer to** the relation R_2.

2. A value of FK in a tuple t_1 of the current state r_1 (R_1) either occurs as a value of PK for some tuple t_2 in the current state r_2(R_2) or is *NULL*. In the former case, se have t_1[FK] = t_2[PK], and we say that the tuple t_1 **references** or **refer to** the tuple t_2.

In the above, R_1 is called the **referencing relation** and R_2 is the referenced relation. If these two conditions hold, a referential integrity constraint from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

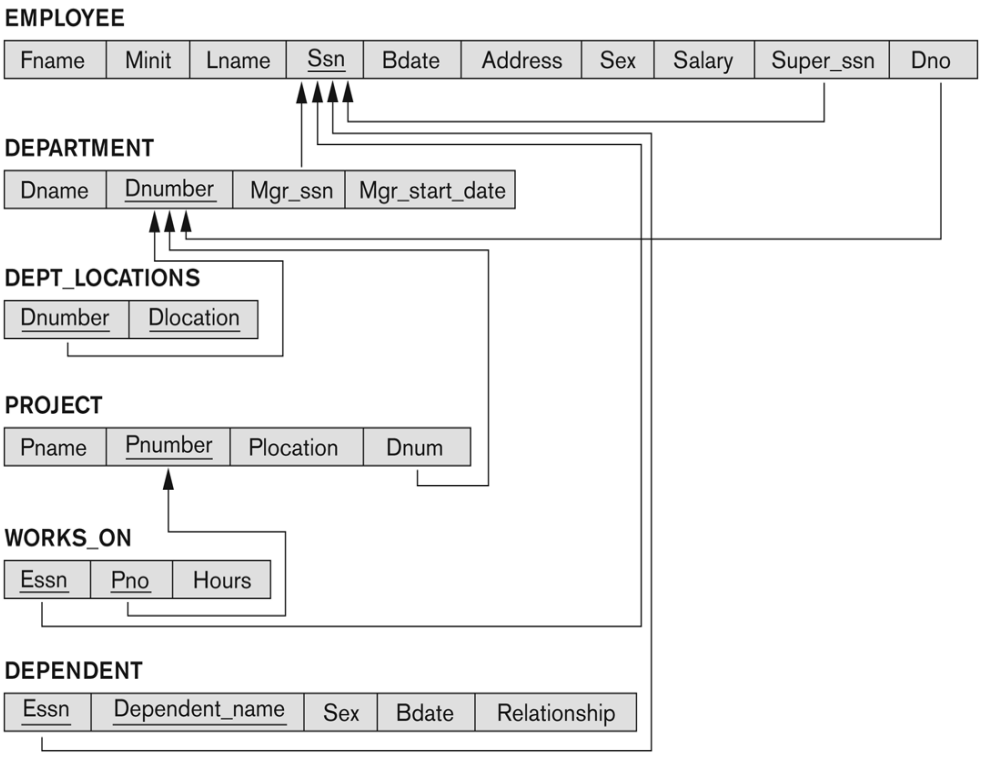
Referential integrity constraints typically arise from the *relationships* among the *entities* represented by the relation schemas. For example, consider the database shown in Figure 4.3. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referring to the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT – the Dnumber attribute – in some tuple t_2 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. In Figure 4.3 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 4.3 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 form each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 4.4 shows the schema in Figure 4.2 with the referential integrity constraints displayed in this manner.

Figure 4.4

Referential integrity constraints displayed on the COMPANY relational database schema.



5.0 The Relational Algebra and Relational Calculus

In this chapter we discuss the two formal languages for the relational model: the relational algebra and the relational calculus. A data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining database structure and constraints. The basic set of operations for the relational model is the **relational algebra.** These operations enable a user to specify basic retrieval requests. 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 relational database management systems (RDBMSs). Third some of its concept are incorporated into SQL standard query language for RDBMSs.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level declarative notation for specifying relational queries. A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus). In a calculus expression, there is no *order of operations* to specify how to retrieve the query result, a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundation in the tuple relational calculus.

5.1 Unary Relational Operations: SELECT and PROJECT

5.1.1 The SELECT Operation

The SELECT operation is used to select 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. 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)

$\sigma_{SALARY > 30,000}$ (EMPLOYEE)

In general, the SELECT operation is denoted by

σ <selection condition>(R)

where the symbol **σ (sigma)** is used to denote the SELECT operator and the selection condition is a Boolean expression specified on the attributes of a relation R. The relation resulting from the SELECT operation has the same attributes as R.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

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

or

<attribute name><comparison op><attribute name>

where <attribute name> is the name of an attribute R, <comparison op> is normally one of the operators {=, <, <, >, >, =}, and <constant value> is a constant value from the attribute domain.

5.1.2 The PROJECT Operation

If we think of a relation a table, the SELECT operation selects 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. For example, to list each employee's first and last name and salary, we use the PROJECT operation as follows:

$\pi_{LNAME, FNAME,SALARY}$(EMPLOYEE)

The general form of the PROJECT operation is

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

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

5.2 Relational Algebra Operations from Set Theory

5.2.1 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:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{DNO=5}}(\text{EMPLOYEE})$$

$$\text{RESULT1} \leftarrow \pi_{\text{SSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT2(SSN)} \leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise and employee who works in department 5. The UNION operation produces the tuples that are I either RESULT1 or RESULT2 or both (see Figure 5.1). Thus, the Ssn value ‘333445555’ appears only once in the result.

Figure 5.1

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

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
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 **MINUS**). 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*. Two relations $R(A_1, A_2, \ldots A_n)$ and $S(B_1, B_2, \ldots, B_n)$ are said to be **union compatible** if they have the same degree n and if dom(Ai) = dom(Bi) for 1<i<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.

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 as both are *associative* operations; that is

$$R \cup (S \cup T) = (R \cup S) \cup T$$

$$(R \cap S) \cap T = R \cap (S \cap T)$$

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 5.2

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

5.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

We discuss the **CARTESIAN PRODUCT** operation which is also known as **CROSS PRODUCT** or **CROSS JOIN** which is denoted by x. This is also a binary set operation, but the relation 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 $R(A_1, A_2, ...,A_m)$ x $S(B_1, B_2, ...,B_m)$, is a relation Q with degree n + m attributes $Q(A_1, A_2, ...,A_m,B_1,$

B_2, ...,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 x S will have $n_R * n_S$ tuples.

5.3 Binary Relational Operations : JOIN and DIVISION

The **JOIN** operation, denoted by ⋈ , is used to combine *related tuples* from two relations into single 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:

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$$

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

The first operation is illustrated in Figure 5.3 Note the MGR_SSN is a foreign key and that the referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

DEPT_MGR

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

Figure 5.3

Result of the JOIN operation

5.3.2 Variations of JOIN : The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.** Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple. For example in Figure 5.3, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR because of the equality join condition specified on

these two attributes. 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.

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:

$$PROJ\_DEPT \leftarrow ROJECT * \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$

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

$$DEPT \leftarrow \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$

$$PROJ\_DEPT \leftarrow ROJECT * DEPT$$

The attribute Dnum is called the **join attribute.** The resulting relation is illustrated in Figure 5.4 (a). In the PROJ_DEPT relation, each tuple combines a PROJECT, but only *one join attribute is* kept.

If the attributes on which the natural join is specified already have the same names in the 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 5.4(b), which combines each department with its locations and has one tuple for each location. In general, NATURAL JOIN is performed by equating all attribute pairs that have the same name in the two relations. 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

Figure 5.4

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

5.3.4 The DIVISION Operation

The DIVISION operation, denoted by ÷ , is useful for a special kind of query that sometimes occurs in database applications. An example is *Retrieve the names of employees* who work on all the projects that 'John Smith' works on. To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$SMITH \leftarrow \sigma_{Fname='John' \text{ AND } Lname='Smith'}(EMPLOYEE)$$

$$SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$$

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

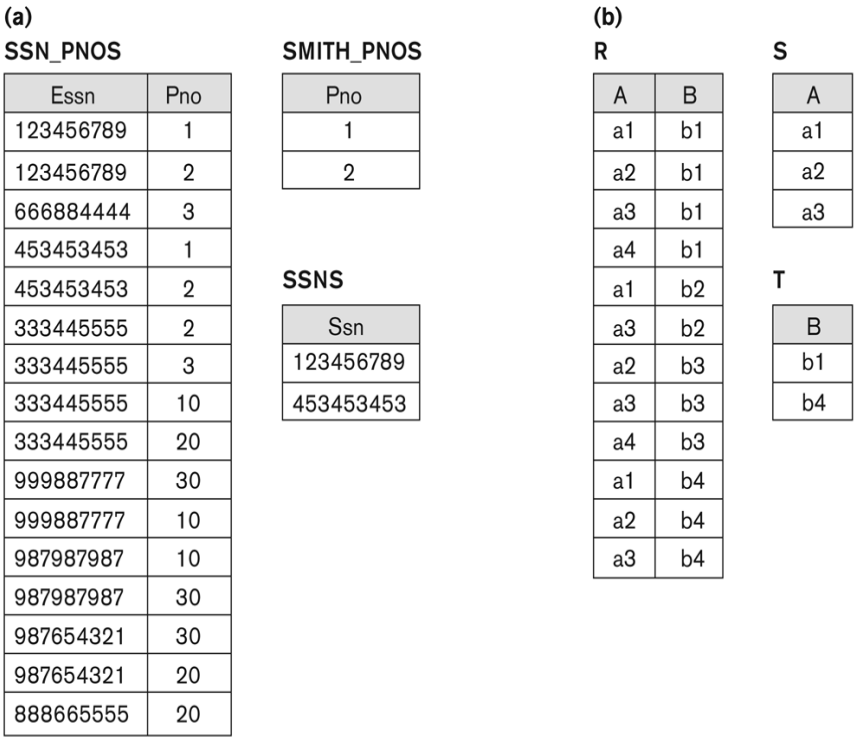
$$SSN\_PNOS \leftarrow \pi_{Essn, Pno}(WORKS\_ON)$$

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

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$

$$RESULT \leftarrow \pi_{Fname, Lname}(SSNS * EMPLOYEE)$$

The previous operations are shown in Figure 5.5.



(a)

SSN_PNOS

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

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

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

S

A
a1
a2
a3

T

B
b1
b4

Figure 5.5

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

5.5 The Tuple Relational Calculus

In relational calculus, we write on **declarative** expression to specify a retrieval request; hence, there is no description of how to evaluate a query. A calculus expression specifies what is to be retrieved rather than how to retrieve it. Therefore, the relational calculus is considered to be a **non-procedural** language.

It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the two languages is *identical.*

5.6.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables.** Each tuple variable usually ranges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

{t | COND(t)}

where t is a tuple variable and COND (t)

The result of such a query is the set of all tuples t that satisfy COND (t). For example, to find the first and last names of all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{t.FNAME, t.LNAME | EMPLOYEE(t) AND t.SALARY>50000}

The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE. The first and last name (PROJECTION πFNAME, LNAME) of each EMPLOYEE tuple t that satisfies the condition t.SALARY>50000 (SELECTION σ SALARY >50000) will be retrieved.

5.6.2 The Existential and Universal Quantifiers

Two special symbols called quantifiers can appear in formulas; these are the universal quantifier ($\forall$) and the existential quantifier ($\exists$).

- Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $\forall$(t) or $\exists$(t) clause; otherwise, it is free.
- If F is a formula, then so are $\exists$(t)(F) and $\forall$(t)(F), where t is a tuple variable.

The formula $\exists$(t)(F) is true if the formula F evaluates to true for some (at least one) tuple assigned to free occurrences of t in F; otherwise $\exists$(t)(F) is false.

The formula $\forall$(t)(F) is true if the formula F evaluates to true for every tuple (in the universe) assigned to free occurrences of t in F; otherwise $\forall$(t)(F) is false.

- $\forall$ is called the universal or “for all” quantifier because every tuple in “the universe of” tuples must make F true to make the quantified formula true.
- $\exists$ is called the existential or “there exists” quantifier because any tuple that exists in “the universe of” tuples may make F true to make the quantified formula true.

Example :Retrieve the name and address of all employees who work for the ‘Research’ department. The query can be expressed as :

{t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) and ∃(d) (DEPARTMENT(d) and d.DNAME='Research' and d.DNUMBER=t.DNO) }

The only *free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|). In above query, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in the query, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple.

The conditions EMPLOYEE (t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.DNAME = 'Research' is a selection condition and corresponds to a SELECT operation in the relational algebra, whereas the condition d.DNUMBER = t.DNO is a JOIN condition.

6.0 SQL-99: Schema Definition, Constraints, Queries, and Views

This chapter presents the main features of the SQL standard for commercial relational DBMSs. The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language).

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL and a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.

6.1 SQL Data Definition and Data Types

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

6.1.1 Data Definition Language

Data definition language, which is usually part of a database management system, is used to define and manage all attributes and properties of a database, including row layouts, column definitions, key columns, file locations, and storage strategy. A DDL statement supports the definition or declaration of database objects such as databases, tables, and views. The Transact-SQL DDL used to manage objects is based on SQL-92 DDL statements (with extensions). For each object class, there are usually CREATE, ALTER, and DROP statements (for example, CREATE TABLE, ALTER TABLE, and DROP TABLE).

Most DDL statements take the following form:

- CREATE *object_name*
- ALTER *object_name*
- DROP *object_name*

The following three examples illustrate how to use the Transact-SQL CREATE keyword to create, alter, and drop tables. CREATE is not limited only to table objects, however.

CREATE TABLE

The CREATE TABLE statement creates a table in an existing database. The following statement will create a table named Importers in the Northwind database. The table will include three columns: CompanyID, CompanyName, and Contact.

```
USE Northwind
CREATE TABLE Importers
(
CompanyID int NOT NULL,
CompanyName varchar(40) NOT NULL,
Contact varchar(40) NOT NULL
)
```

ALTER TABLE

The ALTER TABLE statement enables you to modify a table definition by altering, adding, or dropping columns and constraints or by disabling or enabling constraints and triggers. The following statement will alter the Importers table in the Northwind database by adding a column named ContactTitle to the table.

```
USE Northwind
ALTER TABLE Importers
ADD ContactTitle varchar(20) NULL
```

DROP TABLE

The DROP TABLE statement removes a table definition and all data, indexes, triggers, constraints, and permission specifications for that table. Any view or stored procedure that references the dropped table must be explicitly dropped by using the DROP VIEW or DROP PROCEDURE statement. The following statement drops the Importers table from the Northwind database.

```
USE Northwind
DROP TABLE Importers
```

6.1.2 Data Control Language

Data control language is used to control permission on database objects. Permissions are controlled by using the SQL-92 GRANT and REVOKE statements and the Transact-SQL DENY statement.

GRANT

The GRANT statement creates an entry in the security system that enables a user in the current database to work with data in that database or to execute specific Transact-SQL statements. The following statement grants the Public role SELECT permission on the Customers table in the Northwind database:

```
USE Northwind
GRANT SELECT
ON Customers
TO PUBLIC
```

REVOKE

The REVOKE statement removes a previously granted or denied permission from a user in the current database. The following statement revokes the SELECT permission from the Public role for the Customers table in the Northwind database:

```
USE Northwind
REVOKE SELECT
ON Customers
TO PUBLIC
```

DENY

The DENY statement creates an entry in the security system that denies a permission from a security account in the current database and prevents the security account from inheriting the permission through its group or role memberships.

```
USE Northwind
DENY SELECT
ON Customers
TO PUBLIC
```

6.1.3 Data Manipulation Language

Data manipulation language is used to select, insert, update, and delete in the objects defined with DDL.

SELECT

The SELECT statement retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables. SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form

SELECT <attribute list>
FORM <table list>
WHERE <condition>

Where

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

QUERY 1: Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

SELECT Bdate, Address
FROM EMPLOYEE
WHERE Fname = 'John' **AND** Minit = 'B' AND Lname = 'Smith';

QUERY 2: Retrieve the name and address of all employees who work for the 'Research' department.

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

QUERY 3: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

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

QUERY 4: The following statement retrieves the CustomerID, CompanyName, and ContactName data for companies who have a CustomerID value equal to alfki or anatr. The result is ordered according to the ContactName value:

```
USE Northwind
SELECT CustomerID, CompanyName, ContactName
FROM Customers
WHERE (CustomerID = 'alfki' OR CustomerID = 'anatr')
ORDER BY ContactName
```

6.1.4 Ambiguous Attribute Names, Aliasing, and Tuple Variables

In SQL the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 4.2 and 4.3, the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query 2 would be rephrased as shown in Q2A. We must prefix the attributes Name and Dnumber in Q2A to specify which ones we are referring to because the attribute names are used in both relations:

```
Q2A:  SELECT    Fname, EMPLOYEE.Name, Address
      FROM      EMPLOYEE, DEPARTMENT
      WHERE     DEPARTMENT.NAME='Research' AND
                DEPARTMENT.Dnumber= EMPLOYEE. .Dnumber ;
```

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

QUERY 5: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q5:  SELECT   E.FNAME, E.LNAME, S.FNAME, S.LNAME
     FROM     EMPLOYEE AS E, EMPLOYEE AS S
     WHERE    E.SUPERSSN=S.SSN
```

In this case, we are allowed to declare alternative relation names E and S, called **aliases** or **tuple variables,** for the EMPLOYEE relation. An alias can follow the keyword **AS,** as shown in Q5, or it can directly follow the relation name – for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q5.

We could specify Query 2 Q2A as in Q2B:

```
SELECT   E.Fname, E.Name, E.Address
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    D.Name='Research' AND D.Dnumber= E.Dnumber ;
```

6.1.5 Unspecified WHERE Clause and Use of the Asterisk

We discuss two more features of SQL here. A missing WHERE clause indicates no condition on tuple selection; hence all tuples of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then CROSS PRODUCT – all possible tuple combinations – of these relations is selected. For example, Query 6 selects all EMPLOYEE Ssns and Query 7 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname.

```
Q6: SELECT   Ssn
    FROM     EMPLOYEE
```

```
Q7: SELECT   Ssn, Dname
    FROM     EMPLOYEE, DEPARTMENT
```

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q7 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra. If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q7, we get the CROSS PRODUCT (except for duplicate elimination, if any).

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes. For example, query Q2C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5, query Q2D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and Q7A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q2C:  SELECT   *
      FROM     EMPLOYEE
      WHERE    Dn = 5

Q2C:  SELECT   *
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname = 'Research' AND Dn = Dnumber

Q2C:  SELECT   *
      FROM     EMPLOYEE, DEPARTMENT
```

6.2 Aggregate Functions in SQL

Grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts. A number of built-in functions exist: **COUNT, SUM, MAX, MIN,** and **AVG.** The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set of multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another. We illustrate the use of these functions with example queries.

Query 8. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q8: SELECT   SUM(Salary), MAX(Salary), MIN(Salary),AVG(Salary)
    FROM     EMPLOYEE;
```

If we want to get the preceding function values for employees of a specific department – say the 'Research' department, we can write Query 9, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

Query 9: Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q9:  SELECT   SUM(Salary), MAX(Salary), MIN(Salary),AVG(Salary)
     FROM     (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
     WHERE    Dno=Dnumber AND Dname='Research'
```

Queries 10 and 11. Retrieve the total number of employees in the company (Q10) and the number of employees in the 'Research' department (Q11).

```
Q10: SELECT   COUNT(*)
     FROM     EMPLOYEE
```

```
Q11: SELECT   COUNT(*)
     FROM     EMPLOYEE, DEPARTMENT
     WHERE    Dno=Dnumber AND Dname='Research'
```

Here the asterisk (*) refers to the rows (tuples), so COUNT(*) returns the number of rows in the result of the query.

6.2.1 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees in each department or the number of employees who work on each project. In these cases we need to **partition** the relation into nonoverlapping subsets (or groups) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s).** We can then apply the function to each such group independently. SQL has a **GROUP BY** clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

Query 12. For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q12:  SELECT     Dno,COUNT(*), AVG(Salary)
      FROM       EMPLOYEE
      GROUP BY   Dno
```

In Q12, the EMPLOYEE tuples are partitioned into groups, each group having the same value for the grouping attribute Dno. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples.

If NULLs exist in the grouping attribute, the a **separate group** is created for all tuples with a *NULL* value in the grouping attribute. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Q12.

Query 13. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q13:  SELECT     Pnumber, Pname, COUNT(*)
      FROM       PROJECT, WORKS_ON
      WHERE      Pnumber =Pno
      GROUP BY   Pnumber, Pname;
```

Q13 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied after the joining of the two relations. Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions. For example, suppose that we want to modify Query 13 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the that satisfy the condition are retrieved in the result of the query. This is illustrated in Query 14.

Query 14. For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

Q14: **SELECT** Pnumber, Pname, **COUNT(***)
FROM PROJECT, WORKS_ON
WHERE Pnumber =Pno
GROUP BY Pnumber, Pname
HAVING **COUNT(*)** > 2

6.2.2 The ORDER BY Clause

The ORDER BY clause sorts a query result by one or more columns (up to 8060 bytes). A sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed. If more than one column is named in the ORDER BY clause, sorts are nested.

The following statement sorts the rows in the Titles table, first by publisher (in descending order), then by type (in ascending order within each publisher), and finally by price (also ascending, because DESC is not specified):

```
USE Pubs
SELECT Pub_id, Type, Title_id, Price
FROM Titles
ORDER BY Pub_id DESC, Type, Price
```

6.3 INSERT, DELETE and UPDATE Statements in SQL

INSERT

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.

For example, to add a new tuple to the EMPLOYEE relation shown in Figure 4.2, we can use U1:

```
U1:   INSERT INTO        EMPLOYEE
      VALUES ('Richard','K','Marini', '653298653', '30-DEC-52',
      '98 Oak Forest,Katy,TX', 'M', 37000,'987654321', 4 )
```

A second form of INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification and no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be left out. For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use U1A:

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

The following statement adds a row to the Territories table in the Northwind database. The TerritoryID value for the new row is 98101; the TerritoryDescription value is Seattle; and the RegionID value is 2.

```
USE Northwind
INSERT INTO Territories
VALUES (98101, 'Seatle', 2)
```

UPDATE

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

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

6.4 Views in SQL

A **view** in SQL terminology is a single table that is derived from other tables. These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a **virtual table,** in contrast to base tables, whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitation on querying a view. We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

For example in Figure 4.2 we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS_ON, and PROJECT tables the **defining tables** of the view.
The command to specify a view is **CREATE VIEW.** The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. Below are example views that has been applied to the database schema of Figure 4.2.

```
V1:   CREATE VIEW     WORKS_ON1
      AS SELECT       Fname, Lname, Pname, Hours
          FROM        EMPLOYEE, PROJECT, WORKS_ON
          WHERE       Ssn=Essn AND Pno =Pnumber

V2:   CREATE VIEW     DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT       Dname, COUNT (*), SUM(Salary)
          FROM        DEPARTMENT,EMPLOYEE
          WHERE       Dnumber=Dno
          GROUP BY    Dnumber
```

In V1, we did not specify any attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT and WORKS_ON. View V2 explicitly specifies new attributes names for the view DEPT_INFO, using a one-to-one correspondence between the attributes

specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on 'ProjectX', we can utilise the WORKS_ON1 view and specify the query as in QV1:

```
QV1:  SELECT    Fname, Lname
      FROM      WORKS_ON1
      WHERE     Pname = 'ProjectX'
```

The following statement updates the row in the Territories table (in the Northwind database) whose TerritoryID value is 98101. The TerritoryDescription value will be changed to Downtown Seattle.

```
USE Northwind
UPDATE Territories
SET TerritoryDescription = 'Downtown Seattle'
WHERE TerritoryID = 98101
```

6.5 Using Advanced Query Techniques To Access Data

Once you have grown comfortable with the fundamentals of a SELECT statement and are familiar with the various clauses, you are ready to learn more advanced querying techniques. One of these techniques is to combine the contents of two or more tables to produce a result set that incorporates rows and columns from each table. Another technique is to use subqueries, which are SELECT statements nested inside other SELECT, INSERT, UPDATE, or DELETE statements. Subqueries can also be nested in other subqueries.

6.5.1 Using Joins To Retrieve Data

By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins can be specified in either the FROM or WHERE clauses. The join conditions combine with the WHERE and HAVING search conditions to control the rows that are selected from the base tables referenced in the FROM clause. Specifying the join conditions in the FROM clause, however, helps separate them from any other search conditions that might be specified in a WHERE clause and is the recommended method for specifying joins.

When multiple tables are referenced in a single query, all column references must be unambiguous. The table name must be used to qualify any column name that is duplicated in two or more tables referenced in a single query.

The select list for a join can reference all of the columns in the joined tables or any subset of the columns. The select list is not required to contain columns from every table in the join. For example, in a three-table join, only one table can be used as a bridge from one of the other tables to the third table, and none of the columns from the middle table have to be referenced in the select list. Although join conditions usually use the equals sign (=) comparison operator, other comparison or relational operators can be specified (as can other predicates). Most joins can be categorized as inner joins or other joins.

Inner joins return rows only when there is at least one row from both tables that matches the join condition, eliminating the rows that do not match with a row from the other table. Outer joins, however, return all rows from at least one of the tables or views mentioned in the FROM clause as long as these rows meet any WHERE or HAVING search conditions.

Inner Joins

An inner join is a join in which the values in the columns being joined are compared through the use of a comparison operator.

The following SELECT statement uses an inner join to retrieve data from the Publishers table and the Titles table in the Pubs database:

```
SELECT t.Title, p.Pub_name
FROM Publishers AS p INNER JOIN Titles AS t
ON p.Pub_id = t.Pub_id
ORDER BY Title ASC
```

The SELECT statement retrieves data from the Title column in the Title (t) table and from the Pub_name column in the Publishers (p) table. Because the statement uses an inner join, it returns only those rows for which there is an equal value in the join column (the p.Pub_id column and the t.Pub_id column).

Outer Joins

There are three types of outer joins: left, right, and full. All rows retrieved from the left table are referenced with a left outer join, and all rows from the right table are referenced in a right outer join. All rows from both tables are returned in a full outer join.

Using Left Outer Joins

A result set generated by a SELECT statement that includes a left outer join includes all rows from the table referenced to the left of LEFT OUTER JOIN. The only rows that are retrieved from the table to the right are those that meet join condition.

In the following SELECT statement, a left outer join is used to retrieve the authors' first names, and (when applicable) the names of any publishers that are located in the same cities as the authors:

```
USE Pubs
SELECT a.Au_fname, a.Au_lname, p.Pub_name
FROM    Authors a LEFT OUTER JOIN Publishers p
ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname ASC, a.Au_fname ASC
```

The result set from this query will list the name of every author in the Authors table. The result set will include only those publishers that are located in the same cities as the authors, however. If a publisher is not located in the author's city, a null value is returned for the Pub_name column on the result set.

Using Right Outer Joins

A result set generated by a SELECT statement that includes a right outer join includes all rows from the table referenced to the right of RIGHT OUTER JOIN. The only rows that are retrieved from the table to the left are those that meet the join condition.

In the following SELECT statement, a right outer join is used to retrieve the list of publishers and authors' first names, and last names, if those authors are located in the same cities as the publishers:

```
USE Pubs
SELECT a.Au_fname, a.Au_lname, p.Pub_name
FROM    Authors a RIGHT OUTER JOIN Publishers p
        ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname ASC, a.Au_fname ASC
```

The result set from this query will list the name of every publisher in the Publishers table. The result set will include only those authors that are located in the same cities as the publishers, however. If an author is not located in the publisher's city, a null value is returned for the Au_fname and Au_lname columns of the result set.

Using Full Outer Joins

A result set generated by a SELECT statement that includes a full outer join includes all rows from both tables, regardless of whether the tables have a matching value (as defined in the join condition).

In the following SELECT statement, a full outer join is used to retrieve the list of publishers and authors' first and last names:

```
USE Pubs
SELECT a.Au_fname, a.Au_lname, p.Pub_name
FROM    Authors a FULL OUTER JOIN Publishers p
        ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname ASC, a.Au_fname ASC
```

The result set from this query will list the name of every publisher in the Publishers table and every author in the Authors table. If an author is not located in the publisher's city, a null value is returned for the Au_fname and Au_lname columns of the result set. If a publisher is not located in the author's city, a null value is returned in the Pub_name column of the result set. When the join condition is met, all columns in the result set will contain a value.

6.5.2 Defining Subqueries inside SELECT Statement

A subquery is a SELECT statement that returns a single value and is nested inside a SELECT, INSERT, UPDATE or DELETE statement or inside another subquery. A subquery can be used

anywhere an expression is allowed. A subquery is also called an inner query or inner select, while the statement containing a subquery is called an outer select.

In the following example, a subquery is nested in the WHERE clause of the outer SELECT statement:

```
USE Northwind
SELECT ProductName
FROM Products
WHERE UnitPrice =
        {
         SELECT UnitPrice
         FROM Products
         WHERE ProductName = 'Sir Rodney' 's Scones'
         }
```

The embedded SELECT statement first identifies the UnitPrice value for Sir Rodney's Scones, which is $10. The $10 value is then used in the outer SELECT statement to return the product name of all products whose unit price equals $10.

If a table only appears in a subquery and not in the outer query, then columns from that table cannot be included in the output (the select list of the outer query).

Types of Subqueries

Subqueries can be specified in many places within a SELECT statement. Statements that include a subquery usually take one of the following formats, however:

- WHERE <expression> [NOT] IN (<subquery>)
- WHERE <expression> <comparison_operator> [ANY | ALL] (<subquery>)
- WHERE [NOT] EXISTS (<subquery>)

Subqueries that Are used with IN or NOT IN

The result of a subquery introduced with IN (or with NOT IN) is a list of zero or more values. After the subquery returns the result, the outer query makes use of it. In the following example, a subquery is nested inside the WHERE clause, and the IN keyword is used:

```
USE Pubs
SELECT Pub_name
FROM Publishers
WHERE Pub_id IN
        (
        SELECT Pub_id
        FROM Titles
        WHERE Type = 'business'
        )
```

You can evaluate this statement in two steps. First the inner query returns the identification numbers of the publishers that have published business books (1389 and 0736). Second, these values are substituted into the outer query, which finds the names that match the identification numbers in the Publishers table.

Subqueries introduced with the NOT IN keywords also return a list of zero or more values. The query is exactly the same as the one in subqueries with IN, except that NOT IN is substituted for IN.

Subqueries that Are Used with Comparison Operators

Comparison operators that introduce a subquery can be modified with the keyword ALL or ANY. Subqueries introduced with a modified comparison operator return a list of zero or more values and can include a GROUP BY or HAVING clause. These subqueries can be restated with EXISTS.

The ALL and ANY keywords each compare a scalar value with a single-column set of values. The ALL keyword applies to every value, and the ANY keyword applies to at least one value. In the following example, the greater than (>) comparison operator is used with the ANY keyword:

```
USE Pubs
SELECT Title
FROM Titles
WHERE Advance > ANY
        (
        SELECT Advance
        FROM Publishers INNER JOIN Titles
        ON Titles.Pub_id = Publishers.Pub_id
                AND Pub_name = 'Algodata Infosystems'
        )
```

This statement finds the titles that received an advance larger than the minimum advance amount paid by Algodata Infosystems (which, in this case, is $5,000). The WHERE clause in the outer SELECT statement contains a subquery that uses a join to retrieve advance amounts for Algodata Infosystems. The minimum advance amount is then used to determine which titles to retrieve from the Titles table.

Subqueries that Are Used with EXISTS and NOT EXISTS

When a subquery is introduced with the keyword EXISTS, it functions as an existence test. The WHERE clause of the outer query tests for the existence of rows returned by the subquery. The subquery does not actually produce any data; instead, it returns a value of TRUE or FALSE.

In the following example, the WHERE clause in the outer SELECT statement contains the subquery and uses the EXISTS keyword.

```
USE Pubs
SELECT Pub_name
FROM Publishers
WHERE  EXISTS
        (
        SELECT * FROM Titles
        WHERE Titles.Pub_id = Publishers.Pub_id
                AND Type = 'Business'
        )
```

To determine the result of this query, consider each publisher's name in turn. In this case, the first publisher's name is Algodata Infosystems, which has identification number 1389. Are there any rows in the Titles table in which Pub_id is 1389 and the type is business? If so, Algodata Infosystems should be one of the values selected. The same process is repeated for each of the other publishers' names.

The NOT EXISTS keywords work like EXISTS, except the WHERE clause in which NOT EXISTS is used is satisfied if the subquery returns no rows.

DELETE

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select tuples to be deleted. Depending on the number of selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. The DELETE commands in U4A to U4D, if applied independently to the database of Figure 4.3 will delete zero, one, four and all tuples respectively from the EMPLOYEE relation:

```
U4A:  DELETE FROM    EMPLOYEE
      WHERE          LNAME='Brown'

U4B:  DELETE FROM    EMPLOYEE
      WHERE          SSN='123456789'

U4C:  DELETE FROM    EMPLOYEE
      WHERE          Ssn='123456789'

U4D:  DELETE FROM    EMPLOYEE
```

The following statement removes the Territories table (from the Northwind database) whose TerritoryID value is 98101.

```
USE Northwind
DELETE FROM  Territories
WHERE TerritoryID = 98101
```

7.0 Functional Dependencies and Normalisation for Relational Databases

7.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four informal measures of quality for relational schema design in this chapter.

- Semantics of the attributes
- 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, as we will see.

7.1.1 Imparting Clear Semantics to Attributes in Relations

Whenever we group attributes to form a relational 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 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 7.1, a simplified version of the COMPANY relational database schema of Figure 4.2 and Figure 4.3, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents and employee, with values for the employee's name (Ename), social security number (Ssn), birth date (Bdate), and (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 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 relations is designed.

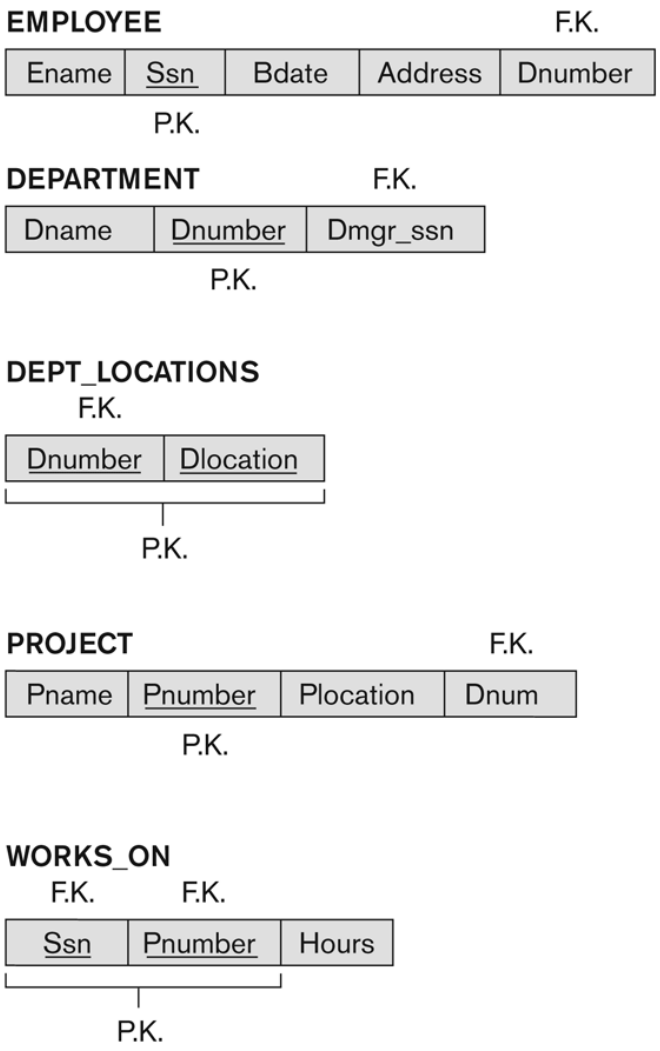


Figure 7.1

A simplified COMPANY relational database schema.

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.

7.1.2 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 4.3 with that for an EMP_DEPT base relation in Figure 7.2, 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.

Figure 7.2

Example states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 10.2. These may be stored as base relations for performance reasons.

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

In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 4.3. 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 7.2), which augments the WORKS_ON relation with additional attributes from EMPLOYEE AND PROJECT.

Another serious problem with using the relations in Figure 7.2 as base relations is the problem of **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 NULL (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 the attribute values of department 5 correctly so that they are consistent with values for department 5 in other tuples in EMP_DEPT. In the design of Figure 4.3, 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 causes a problem because Ssn is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity – not a department entity. 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 4.3 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 discussed earlier. 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 4.3 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.

Based on the preceding three anomalies, we can state the guideline that 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. For example, if an important query retrieves information concerning the department of an employee along with employee attributes, the EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates) so that, 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. This reduces the number of JOIN terms specified in the query, making it simpler to write the query correctly, and in many cases it improves the performance.

7.1.3 NULL Values in Tuples

In some schema designs we may group many attributes together int 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 or JOIN operations involve comparisons. If NULL values are present, the results become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute does not apply to this tuple.
- The attribute value for this tuple is unknown.
- The value is known but absent; that is, it has not been 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 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 10 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.

7.1.4 Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 7.4 (a), which can be used instead of the single EMP_PROJ relation of Figure 7.3(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 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. 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 7.4(b) shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the EMP_PROJ relation of Figure 7.2, which are obtained by applying the appropriate PROJECT (PI) operations to EMP_PROJ (ignore the dashed lines in Figure 7.4(b) for now).

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. 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 7.8

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 (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.

Figure 7.4: Particularly poor design for the EMP_PROJ relation of Figure 7.3(b)
(a) The two relation schemas EMP_LOCS and EMP_PROJ1 (b) The result of projecting the Extension of EMP_PROJ from Figure 7.2 onto the relations EMP_LOCS and EMP_PROJ1

Figure 7.8: Results of applying NATURAL JOIN to the tuples above the dotted lines in EMP_PROJ1 and EMP_LOCS of Figure 7.4. Generated spurious tuples are marked by asterisks.

7.2 Normalisation of Relations

Normalization is a design technique that is widely used as a guide in designing relational databases. Normalization is essentially a two step process that puts data into tabular form by removing repeating groups and then removes duplicated data from the relational tables.

Normalization theory is based on the concepts of **normal forms**. A relational table is said to be a particular normal form if it satisfied a certain set of constraints. There are currently five normal forms that have been defined. In this section, we will cover the first three normal forms that were defined by E. F. Codd.

Basic Concepts

The goal of normalization is to create a set of relational tables that are free of redundant data and that can be consistently and correctly modified. This means that all tables in a relational database should be in the third normal form (3NF). A relational table is in 3NF if and only if all non-key columns are (a) mutually independent and (b) fully dependent upon the primary key. Mutual independence means that no non-key column is dependent upon any combination of the other columns. The first two normal forms are intermediate steps to achieve the goal of having all tables in 3NF. In order to better understand the 2NF and higher forms, it is necessary to understand the concepts of functional dependencies and lossless decomposition.

7.1 Functional Dependencies

The concept of functional dependencies is the basis for the first three normal forms. A column, Y, of the relational table R is said to be **functionally dependent** upon column X of R if and only if each value of X in R is associated with precisely one value of Y at any given time. X and Y may be composite. Saying that column Y is functionally dependent upon X is the same as saying the values of column X identify the values of column Y. If column X is a primary key, then all columns in the relational table R must be functionally dependent upon X.

A short-hand notation for describing a functional dependency is:

R.x —>; R.y

which can be read as in the relational table named R, column x functionally determines (identifies) column y.

Full functional dependence applies to tables with composite keys. Column Y in relational table R is fully functional on X of R if it is functionally dependent on X and not functionally dependent upon any subset of X. Full functional dependence means that when a primary key is composite, made of two or more columns, then the other columns must be identified by the entire key and not just some of the columns that make up the key.

7.1.1 First Normal Form

A relational table, by definition, is in first normal form. All values of the columns are atomic. That is, they contain no repeating values.

Consider the DEPARTMENT relation schema shown in Figure 7.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 7.6. We assume that each department can have a number of locations. The DEPARTMENT schema and an example relation state are shown in Figure 7.6. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 7.6(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 $\longrightarrow$ Dlocations because each set is considered a single member of the attribute domain.

In either case, the DEPARTMENT relation of Figure 7.6 is not in INF. 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 4.3. 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 7.6(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. Querying on this attribute also becomes difficult.

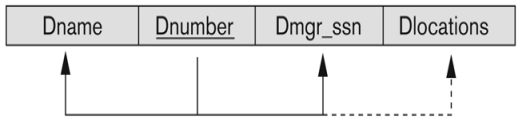
Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

Figure 7.6

Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

(a)

DEPARTMENT



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

7.1.2 Second Normal Form

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X -> 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 ε X, (X – {A})

does not functionally determine Y. A functional dependency X -> Y is a **partial dependency** if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X – {A}) -> Y. In Figure 7.3, {SSN, PNUMBER} -> HOURS is a full functional dependency (neither SSN -> HOURS nor PNUMBER -> HOURS hold). However, the dependency {SSN, PNUMBER} -> ENAME is partial because SSN -> ENAME also holds.

Definition: A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

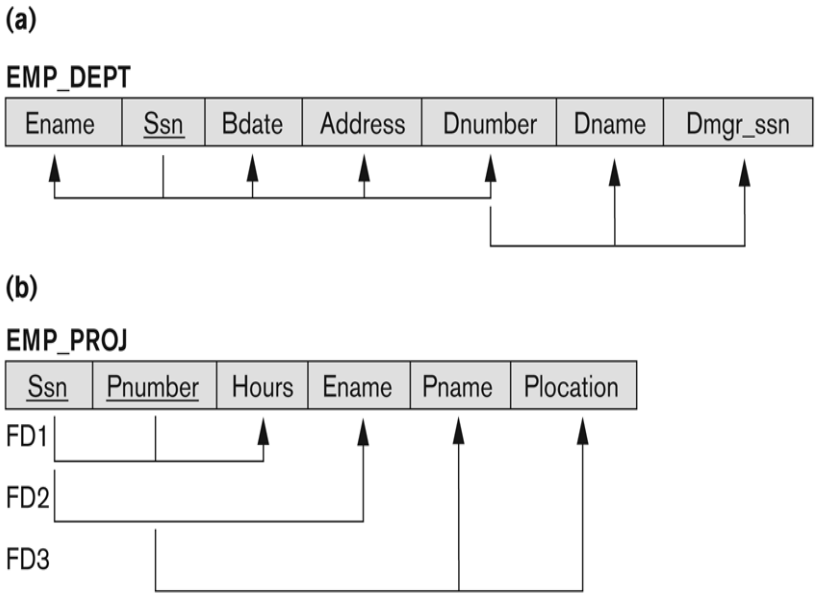
Candidate key: If a relation schema has more than one key, each is called a candidate key. One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called secondary keys. Each relation schema must have a primary key.

Prime Attribute : 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 7.1 both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

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 10.3(b) is in 1NF but is not in 2NF.

Figure 7.3
Two relation schemas suffering from update anomalies.
(a) EMP_DEPT and
(b) EMP_PROJ.



The nonprime attribute Ename violates 2NF because 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 in Figure 10.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 7.7(a), each of which is in 2NF.

7.1.3 Third Normal Form

Third nomal form (3NF) is based on the concept of *transitive dependency*. A functional dependency X -> Y in a relation schema R is a **transitive dependency** if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both X -> Z and Z -> Y hold. The dependency Ssn -> Dmgr_ssn is transitive through Dnumber in EMP_DEPT of Figure 10.3 (a) because both the dependencies Ssn -> Dnumber and Dnumber -> 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 7.3(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 two 3NF relation schemas ED1 and ED2 shown in Figure 7.7(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.

We can see that any functional dependency in which the left-hand side is part (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 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.

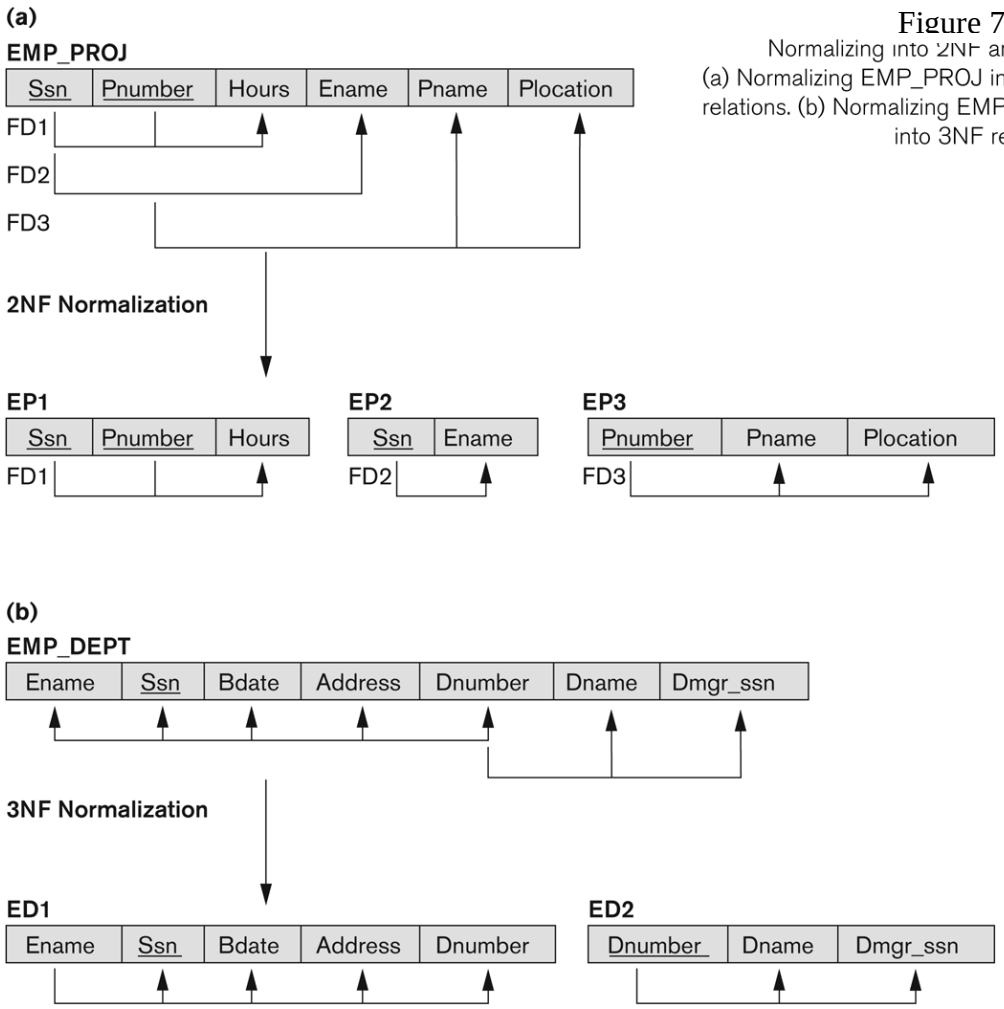


Figure 7.7
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

Table 7.1 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.

Normal Form	Test	Remedy (Normalisation)
First (1NF)	Relation should have no multivalued or nested relations.	Form new relations for each multivalued attribute or nested relationsl
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).

Table 7.1: Summary of Normal Forms Based on Primary Keys and Corresponding Normalisation