

Module 1

Chapter 1: Introduction to Databases

1.1 Introduction

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

Database

A **database** is a collection of related data.¹ By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know.

A database has the following implicit properties:

- It is a logically coherent collection of data, to which some meaning can be attached.
- It 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.

To summarize: a database has some source (i.e., the miniworld) from which data are derived, some degree of interaction with events in the represented miniworld and an audience that is interested in using it.

Size/Complexity: A database can be of any size and complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. An example of a large commercial database is Amazon.com. It contains data for over 20 million books, CDs, videos, DVDs, games, electronics, apparel, and other items.

Computerized vs. manual: A database may be generated and maintained manually or it may be computerized. For example, simple database like telephone directory may be created and maintained manually. Huge and complex database may be created and maintained either by a

group of application programs written specifically for that task or by a database management system.

Database Management System (DBMS)

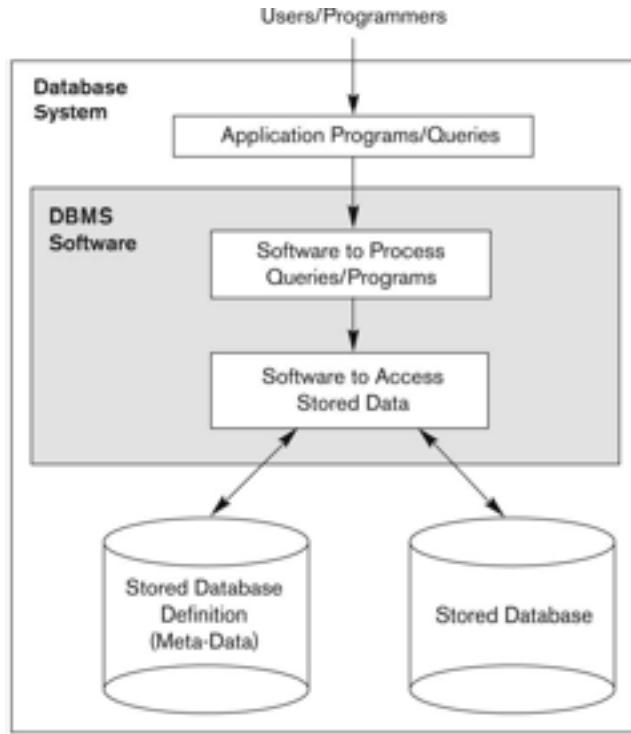
A **database management system** (DBMS) is a collection of programs enabling users to create and maintain a database. More specifically, The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications.

- **Defining** a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is stored by the DBMS 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 or software malfunction (or crashes) and *security protection* against unauthorized or malicious access.
- A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

A database together with the DBMS software is referred to as a **database system**.



An Example

Consider a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. The database is organized as five files, each of which stores **data records** of the same type.

1. STUDENT file: stores data on each student.
2. COURSE file: stores data on each course.
3. SECTION file: stores data on each section of a course.
4. GRADE_REPORT file: stores the grades that students receive in the various sections they have completed.
5. PREREQUISITE file :stores the prerequisites of each course.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

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	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	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

Fig 1.1(b): A database that stores student and course information

Defining a UNIVERSITY database

- Specify the structure of the records of each file - **data elements** to be stored in each record. For example: each STUDENT record includes data to represent the student's Name, Student_number, Class Major. Similarly each COURSE record includes data to represent the Course_name, Course_number, Credit_hours, and Department.
- Specify a data type for each data element within a record. For example: student's Name is a string of alphabetic characters, Student_number is an integer.

Constructing the UNIVERSITY database

- 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.
- 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.

Manipulating a UNIVERSITY database

Database manipulation involves querying and updating.

Examples of queries are as follows:

- Retrieve the transcript—a list of all courses and grades—of 'Smith'
- List the names of students who took the section of the 'Database' course offered in fall 2008 and their grades in that section
- List the prerequisites of the 'Database' course

Examples of updates include the following:

- Change the class of 'Smith' to sophomore
- Create a new section for the 'Database' course for this semester
- Enter a grade of 'A' for 'Smith' in the 'Database' section of last semester

These informal queries and updates must be specified precisely in the query language of the DBMS before they can be processed.

As with software in general, design of a new application for an existing database or design of a brand new database starts off with a phase called **requirements specification and analysis**. These requirements are documented in detail and transformed into a conceptual design that can be represented and manipulated using some computerized tools so that it can be easily maintained, modified, and transformed into a database implementation.

The design is then translated to **logical design** that can be expressed in a data model implemented in a commercial DBMS. The final stage is **physical design**, during which further specifications are provided for storing and accessing the database. The database design is implemented, populated with actual data, and continuously maintained to reflect the state of the miniworld.

1.2 Characteristics of the Database Approach

Database approach vs. File Processing approach

Consider an organization that is organized as a collection of departments/offices. Each department has certain data processing "needs", many of which are unique to it.

In the file processing approach, each department would control a collection of relevant data files and software applications to manipulate that data. For example, one user, the grade reporting office, may keep files on students and their grades. Programs to print a student's transcript and to enter new grades 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 maintains data that is defined once and then accessed by various users. In file systems, each application is free to name data elements independently. In contrast, in a database, the names or labels of data are defined once, and used repeatedly by queries, transactions, and applications.

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

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

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 **meta-data** (i.e., data about data) is stored in the so-called **system catalog**, which contains a description of the structure of each file, the type and storage format of each field, and the various constraints on the data (i.e., conditions that the data must satisfy).

The system catalog is used not only by users but also by the DBMS software, which certainly needs to "know" how the data is structured/organized in order to interpret it in a manner consistent with that structure.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....
....
....
Prerequisite_number	XXXXNNNN	PREREQUISITE

Figure
database catalog for the database

1.2(a): An example of a

2. Insulation between Programs and Data, and Data Abstraction

Program-Data Independence: In traditional file processing, the structure of the data files accessed by an application is "hard-coded" in its source code. If, for some reason, we decide to change the structure of the data ,everyapplication in which a description of that file's structure is hard-coded must be changed!

In contrast, DBMS access programs, in most cases, do not require such changes, because the structure of the data is described separately from the programs that access it and those programs consult the catalog in order to ascertain the structure of the data so that they interpret that data properly.

In other words, the DBMS provides a conceptual or logical view of the data to application programs, so that the underlying implementation may be changed without the programs being modified. (This is referred to as *program-data independence*.)

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

Data abstraction

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

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. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views. For example, one user of the database of Figure 1.2 may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 1.2(b)

TRANSCRIPT					
Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

Fig1.2(
b):
view
derived
from
the
univers
ity
databas
e

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 agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called **online transaction processing(OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

The concept of a **transaction** has become central to many database applications. A transaction is an *executing program or process* that includes one or more database accesses, such as reading or updating of database records. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are.

Database Users

Users may be divided into

- Those who actually use and control the database content, and those who design, develop and maintain database applications called “Actors on the Scene”
- Those who design and develop the DBMS software and related tools, and the computer systems operators called “Workers Behind the Scene”

Actors on the Scene

1. **Database Administrator** (DBA): chief administrator, who oversees and manages the database system (including the data and software). Duties include authorizing users to access the database, coordinating/monitoring its use, acquiring hardware/software for upgrades, etc. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA might have a support staff.
2. **Database Designers:** responsible for identifying the data to be stored and for choosing an appropriate way to organize it. Database designers typically interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups. The final database design must be capable of supporting the requirements of all user groups.
3. **End Users:** These are persons who access the database for querying, updating, and report generation. There are several categories of end users:
 - **Casual end users:** use database occasionally, needing different information each time; use query language to specify their requests; typically middle- or high-level managers.
 - **Naive/Parametric end users:** biggest group of users; frequently query/update the database using standard **canned transactions** that have been carefully programmed and tested in advance. Examples:
 - bank tellers check account balances, post withdrawals/deposits
 - reservation clerks for airlines, hotels, etc., check availability of seats/rooms and make reservations.
 - **Sophisticated end users:** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.

- **Stand-alone users:** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces.

Ex: user of a tax package that stores a variety of personal financial data for tax purposes.

4. System Analysts and Application Programmers (Software Engineers)

- **System Analysts:** determine needs of end users, especially naive and parametric users, and develop specifications for canned transactions that meet these needs.
- **Application Programmers:** Implement, test, document, and maintain programs that satisfy the specifications mentioned above.

Workers behind the Scene

1. **DBMS system designers and implementers:** design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or **modules**, including modules for implementing the catalog, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security.
2. **Tool developers:** design and implement **tools** that facilitate database modeling and design, database system design, and improved performance.
3. **Operators and maintenance personnel** (system administration personnel) : responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.3 Advantages of Using the DBMS Approach

1. Controlling Redundancy

Data redundancy such as tends to occur in the "file processing" approach leads to **wasted storage space, duplication of effort** and a higher likelihood of the introduction of **inconsistency**.

In the database approach, the views of different user groups are integrated during database design. This is known as **data normalization**, and it ensures consistency and saves storage

Space. However, it is sometimes necessary to use **controlled redundancy** to improve the performance of queries. For example, we may store Student_name and Course_number redundantly in a GRADE_REPORT file because whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier.

A DBMS should provide the capability to automatically enforce the rule that no inconsistencies are introduced when data is updated.

2. Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. For example, financial data is often considered confidential and only authorized persons are allowed to access such data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. Hence, the type of access operation—retrieval or update—must also be controlled. A DBMS should provide a **security and authorization subsystem**, which the DBA uses to create accounts, to specify account restrictions and enforce these restrictions automatically.

3. Providing Persistent Storage for Program Objects

The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. Object-oriented database systems make it easier for complex runtime objects to be saved in secondary storage so as to survive beyond program termination and to be retrievable at a later time.

Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions.

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

DBMS maintains indexes that are utilized to improve the execution time of queries and updates. DBMS has a buffering or caching module that maintains parts of the database in main memory buffers. The query processing and optimization module is responsible for choosing an efficient query execution plan for each query submitted to the system.

5. Providing Backup and Recovery

The backup and recovery subsystem of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update transaction, the recovery

subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Disk backup is also necessary in case of a catastrophic disk failure.

6. Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include

- Query languages for casual users
- Programming language interfaces for application programmers
- Forms and command codes for parametric users
- Menu-driven interfaces and natural language interfaces for standalone users.

7. Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways.

For example each section record is related to one course record and to a number of GRADE_REPORT records—one for each student who completed that section. A DBMS must have the capability to represent a variety of complex relationships among the data, to define new relationships as they arise, and to retrieve and update related data easily and efficiently.

8. Enforcing Integrity Constraints

Most database applications are such that the semantics of the data require that it satisfy certain restrictions in order to make sense.

The simplest type of integrity constraint involves specifying a data type for each data item.

For example, in student table we specified that the value of Name must be a string of no more than 30 alphabetic characters.

More complex type of constraint is **referential integrity** involves specifying that a record in one file must be related to records in other files. For example, in university database, we can specify that every section record must be related to a course record.

Another type of constraint specifies uniqueness on data item values, such as every course record must have a unique value for Course_number. This is known as a key or **uniqueness constraint**.

It is the responsibility of the database designers to identify integrity constraints during database design.

9. Permitting Inferencing and Actions Using Rules

In a **deductive** database system, one may specify *declarative* rules that allow the database to infer new data. For example, figure out which students are on academic probation. Such capabilities would take the place of application programs that would be used to ascertain such information otherwise.

Active database systems go one step further by allowing "active rules" that can be used to initiate actions automatically. In today's relational database systems, it is possible to associate triggers with tables.

10. Additional Implications of Using the Database Approach

- **Potential for Enforcing Standards :** database approach permits the DBA to define and enforce standards among database users in a large organization which facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures and so on.
- **Reduced Application Development Time:** once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system.
- **Flexibility:** It may be necessary to change the structure of a database as requirements change. DBMSs allow changes to the structure of the database without affecting the stored data and the existing application programs.

- **Availability of Up-to-Date Information:** DBMS makes the database available to all users. Availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases
- **Economies of Scale:** DBMS approach permits consolidation of data and applications, to overlap between activities of data-processing in different projects or departments. This enables the whole organization to invest in more powerful processors, storage devices, or communication gear, rather than having each department purchase its equipment thus reducing overall costs of operation and management.

1.4 History of Database Applications

▪ Early Database Applications Using Hierarchical and Network Systems

Early database applications maintained records in large organizations such as corporations, universities, hospitals, and banks. In many of these applications, there were large numbers of records of similar structure. There were also many types of records and many interrelationships among them.

Problems with the early database systems

- lack of data abstraction and program-data independence capabilities
- provided only programming language interfaces. This made it time-consuming and expensive to implement new queries and transactions, since new programs had to be written, tested, and debugged.

▪ Providing Data Abstraction and Application Flexibility with Relational Databases

Relational databases were originally proposed to separate the physical storage of data from its conceptual representation and to provide a mathematical foundation for data representation and querying. The relational data model also introduced high-level query languages that provided an alternative to programming language interfaces, making it much faster to write new queries. Hence, data abstraction and program-data independence were much improved when compared to earlier systems.

▪ Object-Oriented Applications and the Need for More Complex Databases

Object-oriented databases (OODBs) mainly used in specialized applications, such as engineering design, multimedia publishing, and manufacturing systems. In addition, many object-oriented concepts were incorporated into the newer versions of relational DBMSs, leading to object-relational database management systems, known as ORDBMSs.

▪ Interchanging Data on the Web for E-Commerce Using XML

The World Wide Web provides a large network of interconnected computers. Users can create documents using a Web publishing language, such as HyperText Markup Language (HTML), and store these documents on Web servers where other users (clients) can access them. Documents can be linked through **hyperlinks**, which are pointers to other documents.

Currently, eXtended Markup Language (XML) is considered to be the primary standard for interchanging data among various types of databases and Web pages. XML combines concepts from the models used in document systems with database modeling concepts.

Extending Database Capabilities for New Applications

The success of database systems in traditional applications encouraged developers of other types of applications to attempt to use them. The following are some examples of these applications:

- **Scientific** applications that store large amounts of data resulting from scientific experiments in areas such as high-energy physics, the mapping of the human genome, and the discovery of protein structures.
- Storage and retrieval of **images**, including scanned news or personal photographs, satellite photographic images, and images from medical procedures such as x-rays and MRIs (magnetic resonance imaging).
- Storage and retrieval of **videos**, such as movies, and **video clips** from news or personal digital cameras.
- **Data mining** applications that analyze large amounts of data searching for the occurrences of specific patterns or relationships, and for identifying unusual patterns in areas such as credit card usage.
- **Spatial** applications that store spatial locations of data, such as weather information, maps used in geographical information systems, and in automobile navigational systems.
- **Time series** applications that store information such as economic data at regular points in time, such as daily sales and monthly gross national product figures.

Databases versus Information Retrieval

Database technology is heavily used in manufacturing, retail, banking, insurance, finance, and health care industries, where structured data is collected through forms, such as invoices or patient registration documents. An area related to database technology is **Information Retrieval (IR)**, which deals with books, manuscripts, and various forms of library-based articles. Data is indexed, cataloged, and annotated using keywords. IR is concerned with searching for material based on these keywords, and with the many problems dealing with document processing and free-form text processing.

When Not to Use a DBMS

- DBMS may involve unnecessary overhead costs that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:
 - High initial investment in hardware, software, and training
 - The generality that a DBMS provides for defining and processing data
 - Overhead for providing security, concurrency control, recovery, and integrity functions
- Therefore, it may be more desirable to use regular files under the following circumstances:
 - Simple, well-defined database applications that are not expected to change at all
 - Stringent, real-time requirements for some application programs that may not be met because of DBMS overhead
 - Embedded systems with limited storage capacity, where a general-purpose DBMS would not fit
 - No multiple-user access to data



Chapter 2: Overview of Database Languages and Architectures

Introduction

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system. Modern DBMS packages are modular in design, with a client/server system architecture. In a basic client/server DBMS architecture, the system functionality is distributed between two types of module. A **client module** is designed to run on a user workstation or personal computer. The client module handles user interaction and provides the user-friendly interfaces such as forms- or menu-based GUIs. The other kind of module, called a **server module** handles data storage, access, search, and other functions.

2.1 Data Models, Schemas, and Instances

Data Model

. By structure of a database we mean the data types, relationships and constraints that apply to the data. Most data models also include a set of basic operations for specifying retrievals and updates on the database. Data model provides the necessary means to achieve abstraction.

Categories of Data Models

Data models can be categorized according to the types of concepts they use to describe the database structure.

1. **High-level** or **conceptual data models**: provide concepts that are close to the way many users perceive data. Conceptual data models use concepts such as entities, attributes, and relationships.
2. **Representational** or **implementation data models**: provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly. Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical**

models. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

3. **Low-level or physical data models:** provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks. Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths.

Database schema

The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.

Schema diagram

A displayed schema is called a **schema diagram**. A schema diagram displays only some aspects of a schema, such as the names of record types and data items, and some types of constraints.

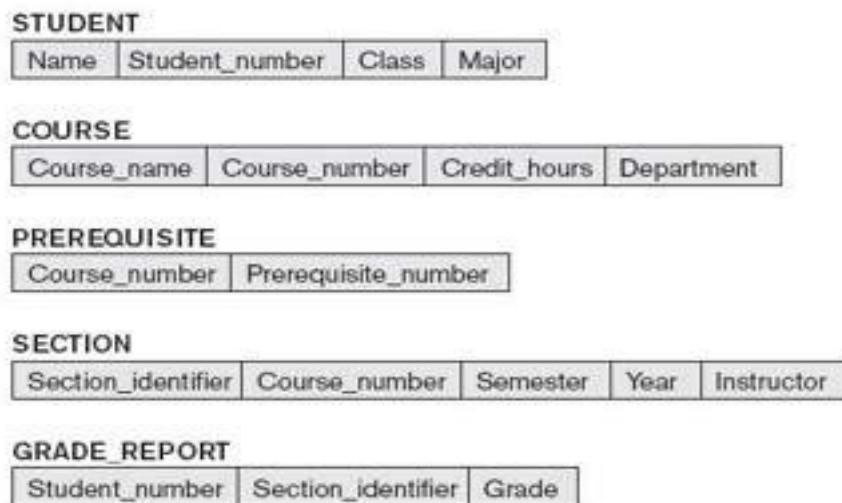


Figure 2.1: Schema diagram for the database

Schema construct

Each object in the schema is called schema construct. For example student or course.

Database state or snapshot

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.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.

The DBMS is partly responsible for ensuring that every state of the database is a **valid state** —that is, a state that satisfies the structure and constraints specified in the schema. The DBMS stores the descriptions of the schema constructs and constraints —also called the **meta-data**— in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

2.2 Three-Schema Architecture and Data Independence

The Three-Schema Architecture

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

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented.
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. Each external schema is typically

implemented using a representational data model, possibly based on an external schema design in a high-level data model.

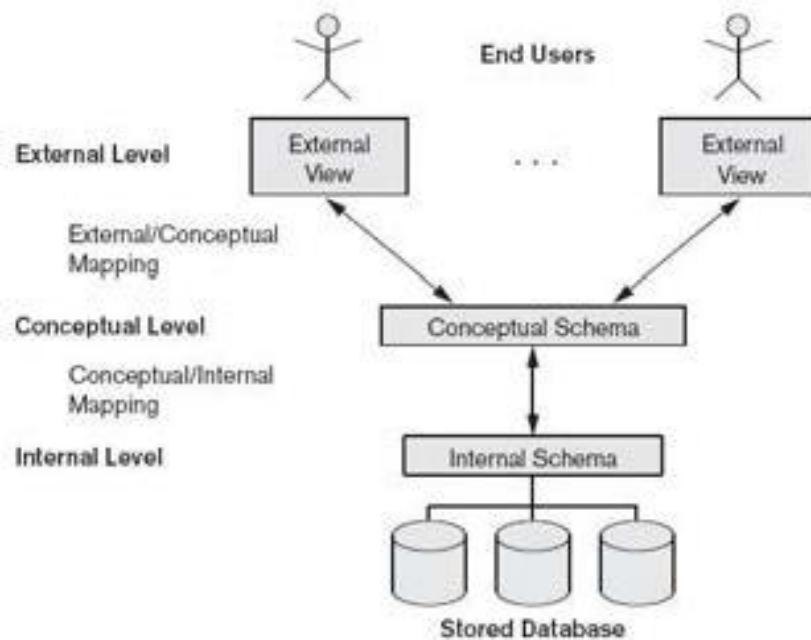


Figure 2.2: The three-schema architecture.

In a DBMS based on the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view.

The processes of transforming requests and results between levels are called **mappings**.

Data Independence

Data independence can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database, to change constraints, or to reduce the database. Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence.

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

Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed.

2.3 Database Languages and Interfaces

The DBMS must provide appropriate languages and interfaces for each category of users.

DBMS Languages

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

Data Definition Language (DDL)

The **data definition language (DDL)** is used by the DBA and by database designers to define both schemas when no strict separation of levels is maintained. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

Storage Definition Language (SDL)

Storage definition language is used when clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only.

The **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

View Definition Language (VDL),

View definition language is used to specify user views and their mappings to the conceptual schema. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries.

Data Manipulation Language (DML)

Data manipulation languages (DML) are used to perform manipulation operation such as retrieval, insertion, deletion, and modification of the data. There are two main types of DMLs :

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

Host language

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.

A high-level DML used in a standalone interactive manner is called a **query language**.

DBMS Interfaces

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

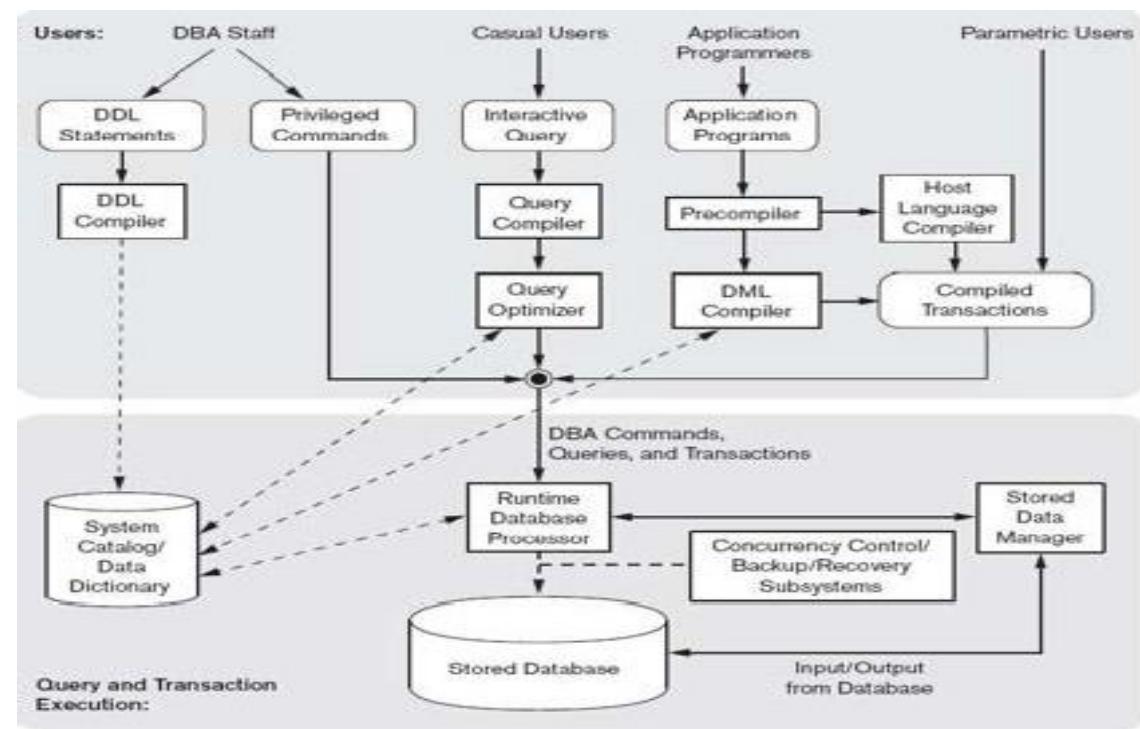
1. **Menu-Based Interfaces for Web Clients or Browsing:** These interfaces present the user with lists of options (called menus) that lead the user through the formulation of a request. There is no need for the user to memorize the specific commands and syntax of a query language. Pull-down menus are a very popular technique in Web-based user interfaces.
2. **Forms-Based Interfaces:** A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions.
3. **Graphical User Interfaces:** A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a pointing device, such as a mouse, to select certain parts of the displayed schema diagram.
4. **Natural Language Interfaces:** These interfaces accept requests written in English or some other language and attempt to understand them. A natural language interface usually has its own schema, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.
5. **Speech Input and Output:** Applications with limited vocabularies such as inquiries for telephone directory, flight arrival/departure, and credit card account information are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.
6. **Interfaces for Parametric Users:** Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Usually a small set of abbreviated commands is included, with the goal of minimizing the number of

keystrokes required for each request.

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

2.4 The Database System Environment

2.4.1 DBMS Component Modules



The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions.

DDL compiler-processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog.

Interactive query interface: interface for Casual users and persons with occasional need for information from the database.

Query compiler- validates for correctness of the query syntax, the names of files and data elements & compiles them into an internal form.

Query optimizer –concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Precompiler - extracts DML commands from an application program and sends to the DML compiler for compilation into object code for database access.

Host language compiler - rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

Runtime database processor executes:

- (1) the privileged commands
- (2) the executable query plans, and
- (3) the canned transactions with runtime parameters.

It works with the system catalog and may update it with statistics. It also works with the stored data manager, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory.

stored data manager uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory.

concurrency control and **backup and recovery systems** integrated into the working of the runtime database processor for purposes of transaction management.

2.4.2 Database System Utilities

Database utilities help the DBA to manage the database system. Common utilities have the following types of functions:

- **Loading:** used to load existing data files—such as text files or sequential files—into the database.
- **Backup:** creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex, but saves storage space.

Database storage reorganization: used to reorganize a set of database files into different file organizations, and create new access paths to improve performance.

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

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

2.4.3 Tools, Application Environments, and Communications Facilities

➤ Tools

- **CASE :** used in the design phase of database systems
- **Data dictionary :** In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**. This information can be accessed directly by users or the DBA when needed.

➤ Application development environments

- **PowerBuilder (Sybase) or JBuilder (Borland):** provide an environment for developing database applications including database design, GUI development, querying and updating, and application program development.
- **Communications software:** allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers. Integrated DBMS and data communications system is called a DB/DC system

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

All DBMS functionality, application program execution, and user interface processing carried out on one machine

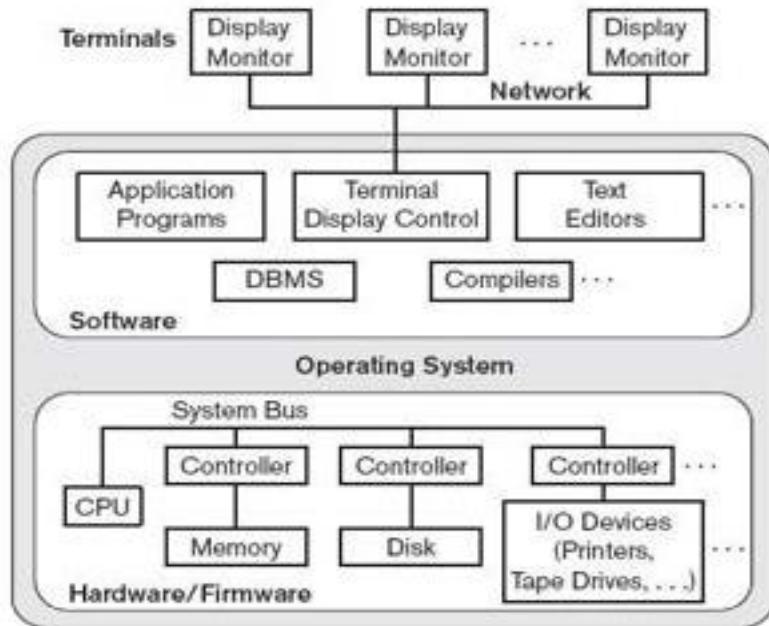


Figure 2.5.1: A physical centralized architecture

▪ **Disadvantages:**

- - When the central site computer or database system goes down, then everyone is blocked from using the system
 - Communication costs from the terminals to the central site can be expensive

2.5.2 Basic Client/Server Architectures

The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network.

▪ **idea**

- define specialized servers with specific functionalities.
- for example file server that maintains the files of the client machines
- The resources provided by specialized servers can be accessed by many client machines.

- The client machines provide the user with the appropriate interfaces to utilize these servers and local processing power to run local applications

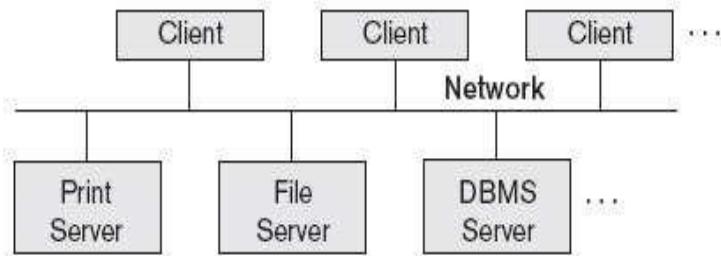


Figure 2.5.2(a) : Logical two-tier client/server architecture

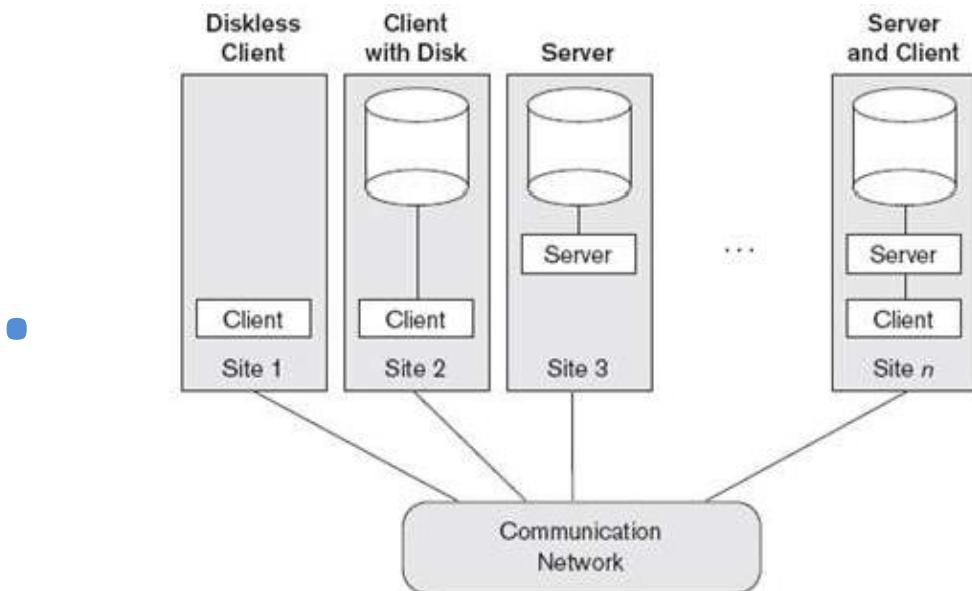


Figure 2.5.2(b) : Physical two-tier client/server architecture.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** is a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality such as database access that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

The software components are distributed over two systems: client and server

- **Server handles**
 - Query and transaction functionality related to SQL processing
- **Client handles**
 - User interface programs and application programs

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 client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined to allow Java client programs to access one or more DBMSs through a standard interface

Object-oriented DBMSs

The different approach to two-tier client/server architecture was taken by some object-oriented DBMSs, where the software modules of the DBMS were divided between client and server in a more integrated way.

- **server level**
 - may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages.
- **client level**
 - may handle the user interface, data dictionary functions, DBMS interactions with programming language compilers, global query optimization, concurrency control, and recovery across multiple servers, structuring of complex objects from the data in the buffers.

In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users/programmers.

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

Many Web applications use an architecture called the three-tier architecture, which adds an intermediate layer between the client and the database server

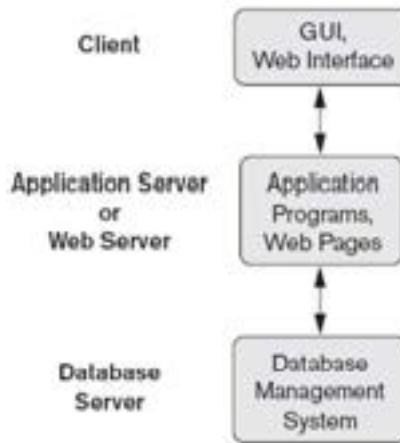


Figure 2.5.4(a): Logical three-tier client/server architecture

- **Client**
 - Contain GUI interfaces and some additional application-specific business rules
- **Application server or the Web server**
 - accepts requests from the client, processes the request and sends database queries and commands to the database server, and then passes 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.
 - It can also improve database security by checking a client's credentials before forwarding a request to the database server.

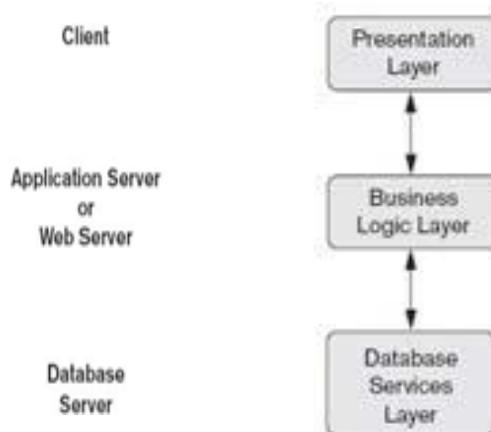


Figure 2.5.4(b): Logical three-tier client/server architecture

Figure 2.5.4(b) shows another architecture used by database and other application package vendors.

- **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
 - can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side
- **The bottom layer**
 - includes all data management services

2.5.5 N-tier Architecture

It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to n-tier architectures; where n may be four or five tiers. The business logic layer is divided into multiple layers

- **Advantage**
 - any one tier can run on an appropriate processor or operating system platform and can be handled independently.

Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a middleware layer, which accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

2.6 Classification of Database Management Systems

Criteria used to classify DBMSs are

1. Data model on which the DBMS is based
 - **Relational:** represents a database as a collection of tables, where each table can be stored as a separate file.
 - **Object:** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class**, and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of each class are specified in terms of predefined procedures called **methods**.
 - **Hierarchical and network (legacy):** The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type**. The

hierarchical model represents data as hierarchical tree structures. Each hierarchy represents a number of related records.

- Native XML DBMS: uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex hierarchical structures.

2. Number of users supported by the system

- Single-user: support only one user at a time and are mostly used with PCs.
- Multiuser: support concurrent multiple users.

3. Number of sites over which the database is distributed

- Centralized: data is stored at a single computer site
- Distributed: can have the actual database and DBMS software distributed over many sites, connected by a computer network
 - **Homogeneous** DDBMSs use the same DBMS software at all the sites
 - **Heterogeneous** DDBMSs can use different DBMS software at each site

4. Cost

- Open source: products like MySQL and PostgreSQL that are supported by third-party vendors with additional services.
- Different types of licensing: Standalone single user versions of some systems like Microsoft Access are sold per copy or included in the overall configuration of a desktop or laptop. In addition, data warehousing and mining features, as well as support for additional data types, are made available at extra cost.

5. On the basis of the **types of access path options for** storing files

- One well-known family of DBMSs is based on inverted file structures.

.6. General purpose or Special purpose

- When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs.

Introduction

Conceptual modeling is a very important phase in designing a successful database application. Entity-Relationship (ER) model 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.

3.1 Using High-Level Conceptual Data Models for Database Design

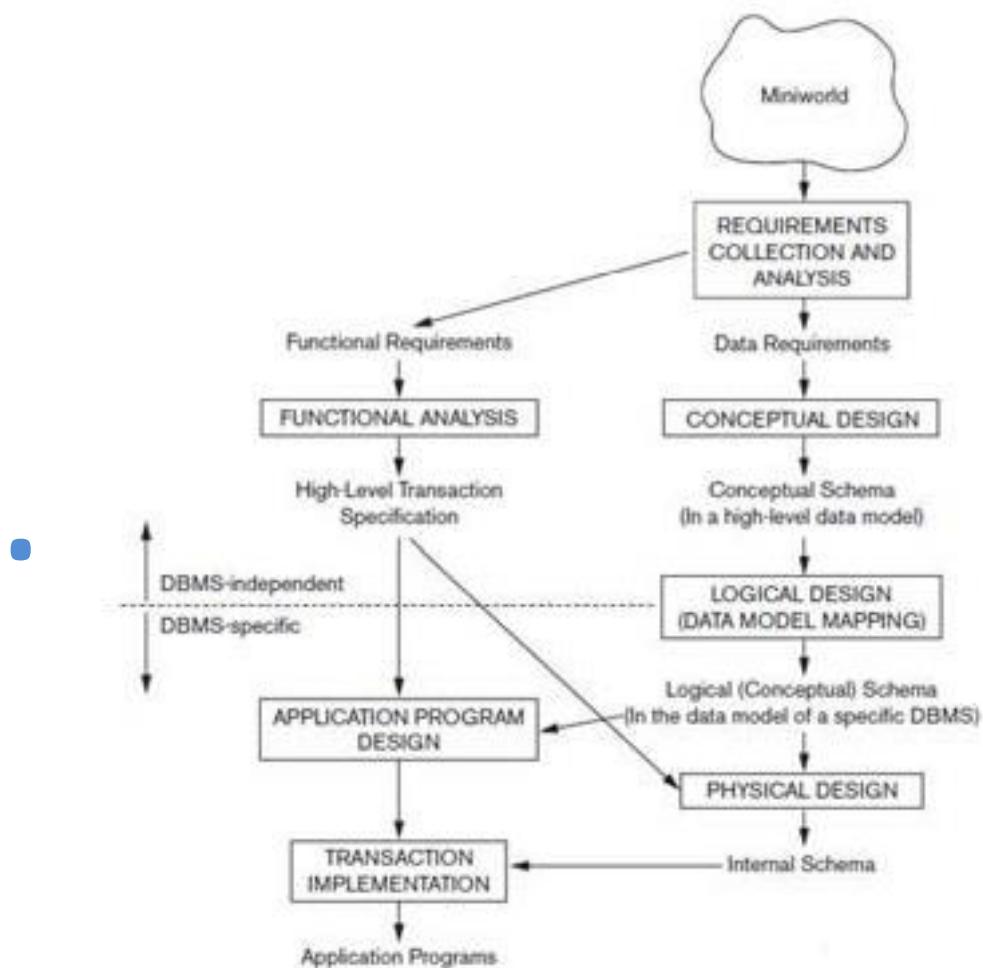


Figure 3.1: A simplified diagram to illustrate the main phases of database design.

The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements.

These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements**

of the application. These consist of the userdefined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates.

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model.

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

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

3.2 Entity Types, Entity Sets, Attributes, and Keys

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

3.2.1 Entities and Attributes

Entity: a thing in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).

Attributes: Particular properties that describe entity. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.

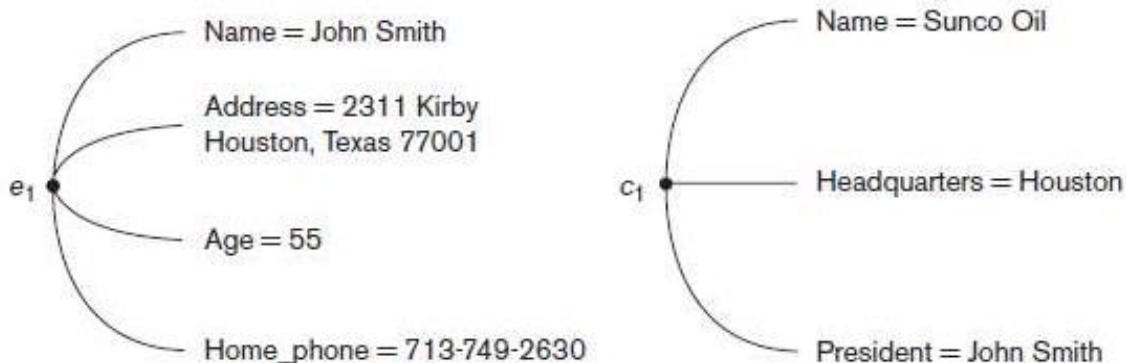


Figure 3.2.1(a): Two entities, EMPLOYEE e1, and COMPANY c1, and their attributes

Types of attributes:

1. Composite versus Simple (Atomic) Attributes
2. Single-valued versus multivalued
3. Stored versus derived
4. NULL values
5. Complex attributes

1. Composite versus Simple (Atomic) Attributes

Composite Attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State, and Zip.

Composite attributes can form a hierarchy. For example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number. The value of a composite attribute is the concatenation of the values of its component simple attributes.



Figure 3.2.1(b): A hierarchy of composite attributes.

Attributes that are not divisible are called **simple** or **atomic attributes**. Example SSN of an employee.

2. Single-Valued versus Multivalued Attributes

Attributes that have a single value for a particular entity are called **single-valued**. For example, Age is a single-valued attribute of a person.

Attributes that can have a set of values for a particular entity are called **Multivalued Attributes**. For example Colors attribute for a car, or a College_degrees attribute for a person. A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

3. Stored versus Derived Attributes

An attribute, which cannot be derived from other attribute are called **stored attribute**. For example, Birth_Date of an employee

Attributes derived from other stored attribute are called **derived attribute**. For example age of an employee can be determined from the current (today's) date and Date of Birth

4. Null Value Attribute(Optional Attribute)

In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called **NULL** is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attribute for a particular entity

5. Complex Attributes

If an attribute for an entity, is built using composite and multivalued attributes, then these attributes are called **complex attributes**. For example, a person can have more than one residence and each residence can have multiple phones, an addressphone for a person entity can be specified as :

{ Addressphone (phone {(Area Code, Phone Number)})},

Address(Sector Address (Sector Number,House Number),
 City, State, Pin))
 }

Here {} are used to enclose multivalued attributes and () are used to enclose composite attributes with comma separating individual attributes

3.2.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types

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

Entity Sets

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

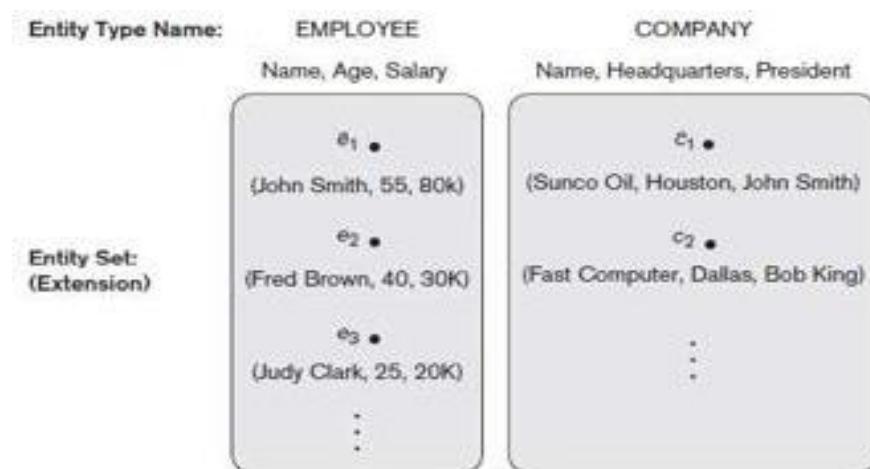


Figure 3.2.2(a): Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

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

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

Key Attributes of an Entity Type

An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a key attribute, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity because no two companies are allowed to have the same name.

In ER diagrammatic notation, each key attribute has its name underlined inside the oval. Some entity types have more than one key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR is a key in its own right

Example: The CAR entity type with two key attributes, Registration and Vehicle_id.

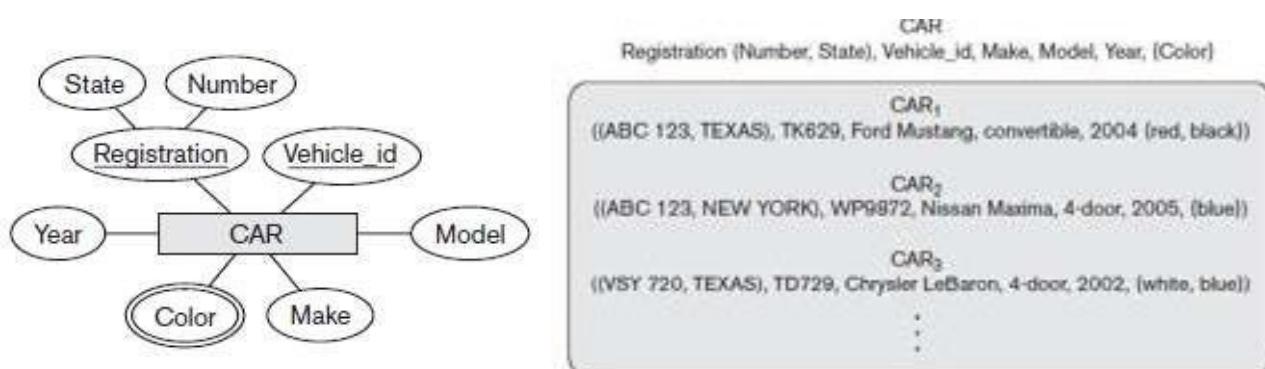


Figure 3.2.2(b) : ER diagram notation

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

Value sets are not displayed in ER diagrams, and are specified using the basic data types available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on.

Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function from E to the power set P(V) of V**: $A : E \rightarrow P(V)$. We refer to the value of attribute A for entity e as $A(e)$. A NULL value is represented by the empty set.

3.3 A Sample Database Application

Mineworld : COMPANY database keeps track of a company's employees, departments, and projects.

- After the requirements collection and analysis phase, the database designers provide the following description of the *mineworld*:
 - The company is organized into departments.
 - Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
 - A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
 - We store each employee's name, Social Security number, address, salary, gender, and birth date.
 - An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department.
 - We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
 - We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, gender, birth date, and relationship to the employee.

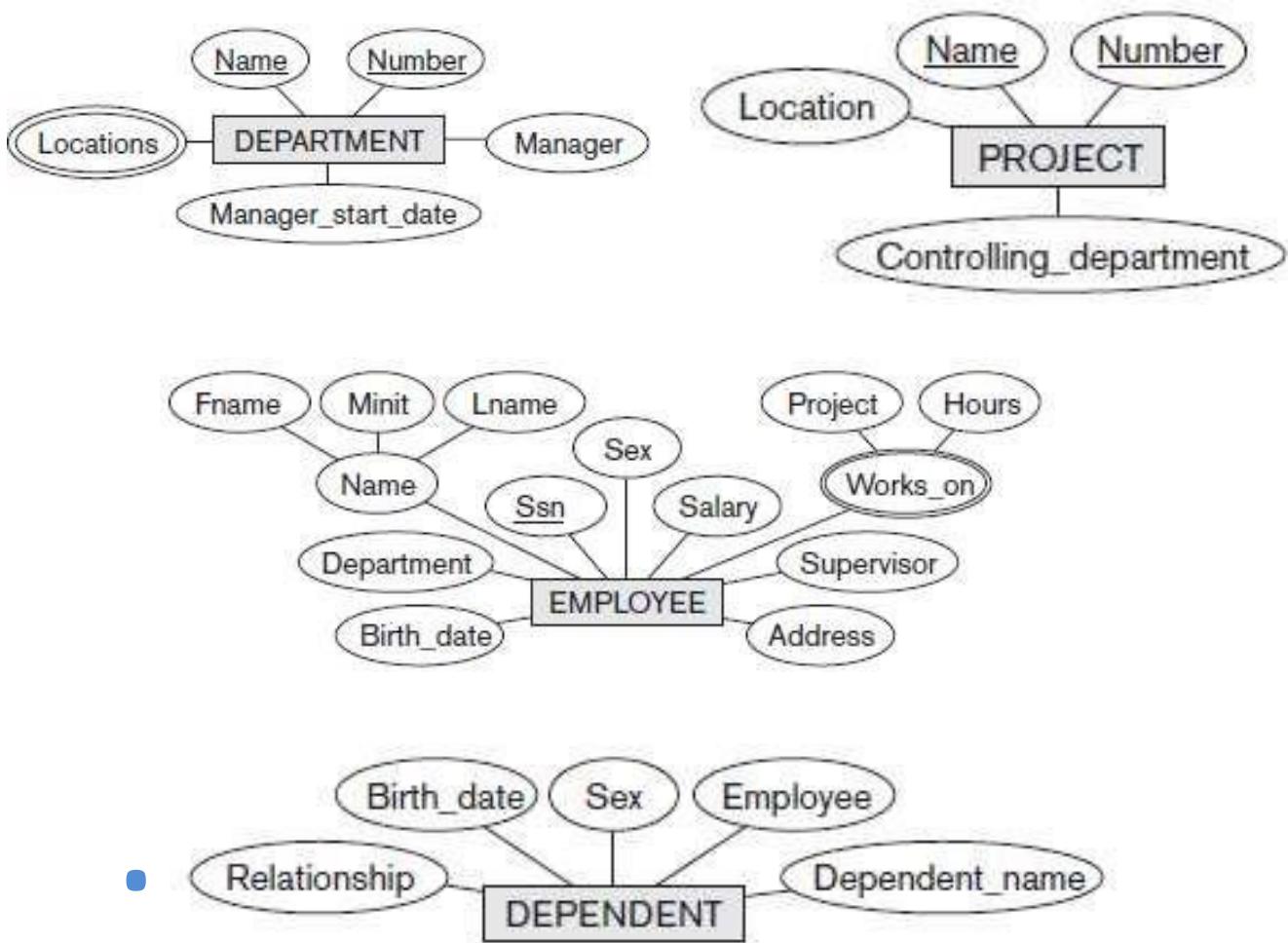


Figure 3.3(a): Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

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

There are several implicit relationships among the various entity types. Whenever an attribute of one entity type refers to another entity type, some relationship exists. For example

- The attribute Manager of DEPARTMENT refers to an employee who manages the department
- 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

In the ER model, these references should not be represented as attributes but as relationships

3.4.1 Relationship Types, Sets, and Instances

A **relationship type R** among n entity types E_1, E_2, \dots, E_n defines a set of associations —or a **relationship set**—among entities from these entity types. Entity types and Entity sets, a Relationship type and its corresponding Relationship set are usually referred to by the same name, R.

Mathematically, the relationship set R is a set of relationship instances r_i , where each r_i associates n individual entities (e_1, e_2, \dots, e_n), and each entity e_i in r_i is a member of entity set E_j , $1 \leq j \leq n$. Each of the entity types E_1, E_2, \dots, E_n is said to participate in the relationship type R. similarly, each of the individual entities e_1, e_2, \dots, e_n is said to participate in the relationship instance $r_i = (e_1, e_2, \dots, e_n)$

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works in the corresponding entity set. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity.

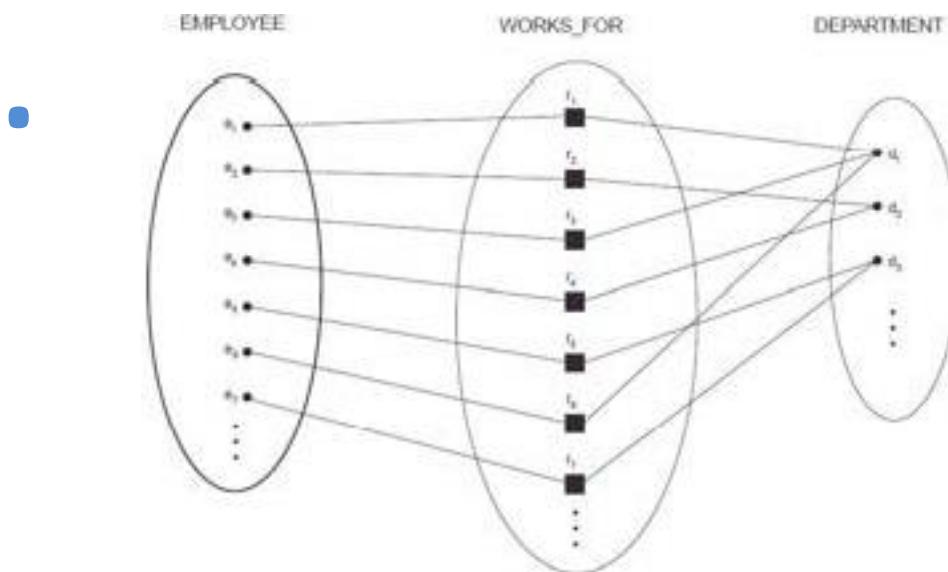


Figure 3.4.1: Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

employees e_1, e_3 , and e_6 work for department d_1 . employees e_2 and e_4 work for department d_2 and employees e_5 and e_7 work for department d_3 .

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

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type

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

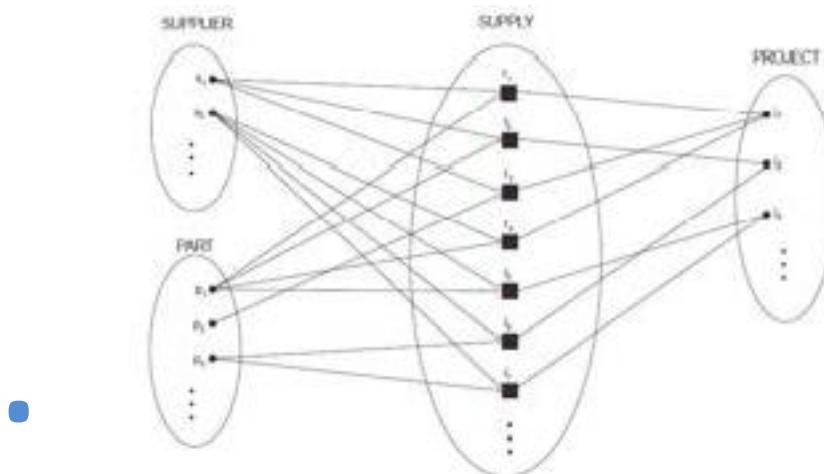


Figure 3.4.2(a): Some relationship instances in the SUPPLY ternary relationship set.

Each relationship instance r_i associates three entities—a supplier s , a part p and a project j —whenever s supplies part p to project j .

Relationships as Attributes

It is sometimes convenient to think of a binary relationship type in terms of attributes. Consider the WORKS_FOR relationship type. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is a reference to the DEPARTMENT entity for which that employee works. This concept of representing relationship types as attributes is used in a class of data models called **functional data models**.

In relational databases, foreign keys are a type of reference attribute used to represent relationships.

Role Names and Recursive Relationships

Each entity type that participates in a relationship type plays a particular role in the relationship.

The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of employee or worker and DEPARTMENT plays the role of department or employer.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the same entity type participates more than once in a relationship type in different roles.

In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays. Such relationship types are called **recursive relationships**. Example of recursive relationships : SUPERVISION relationship type

The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type participates twice in SUPERVISION: once in the role of supervisor (or boss), and once in the role of supervisee (or subordinate). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k , one of which plays the role of supervisor and the other the role of supervisee.

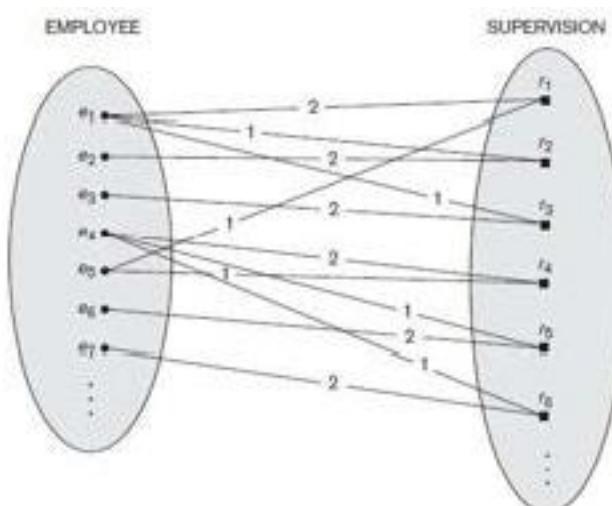


Figure 3.4.2(b):

A recursive relationship

SUPERVISION between EMPLOYEE in the supervisor role (1) and EMPLOYEE in the subordinate role

(2).

3.4.3 Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the

miniworld situation that the relationships represent. For example, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. Two main types of binary relationship constraints:

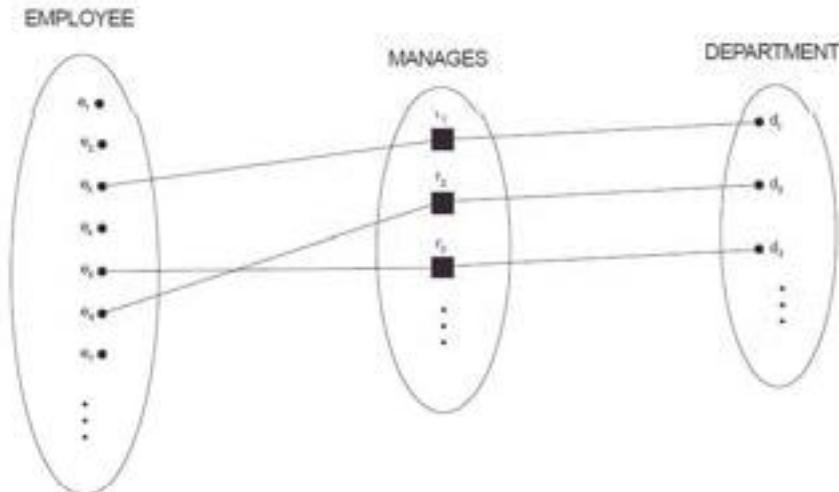
1. cardinality ratio
2. participation.

Cardinality Ratios for Binary Relationships

The cardinality ratio for a binary relationship specifies the maximum number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to 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.

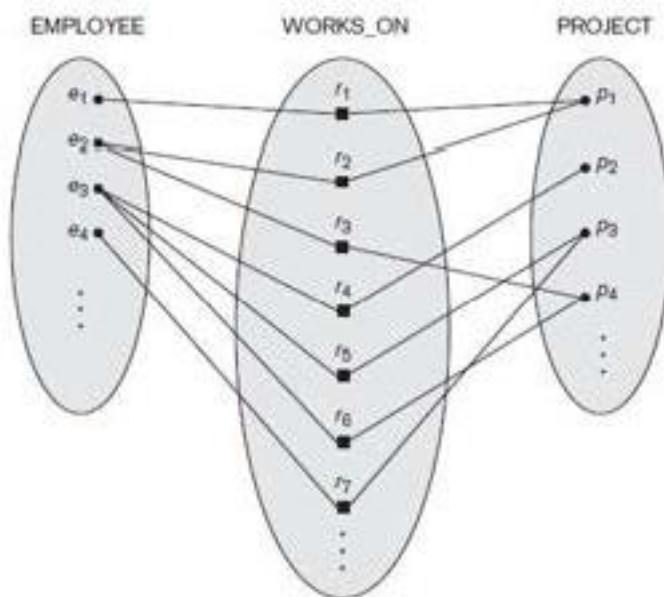
Example of a 1:1 binary relationship

- 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



Example of a M:N binary relationship

- The relationship type WORKS_ON is of cardinality ratio M:N, because the mini-world rule is that an employee can work on several projects and a project can have several employees.
- Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds



Participation Constraints and Existence Dependencies

The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the minimum number of relationship instances that each entity can participate in, and is sometimes called the minimum cardinality constraint. There are two types of participation constraints:

- Total
- Partial

Total participation

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

Partial participation

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.

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

cardinality ratio + participation constraints = structural constraints of a relationship type.

3.4.4 Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type. Another example is to include the date on which a manager started managing a department via an attribute Start_date for the MANAGES relationship type.

Attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For a 1:N relationship type, a relationship attribute can be migrated only to the entity type on the N-side of the relationship. For M:N relationship types, some attributes may be determined by the combination of participating entities in a relationship instance, not by any single entity. Such attributes must be specified as relationship attributes.

3.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity** types. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying or owner entity type**. We call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship. In our example, the attributes of DEPENDENT are Name,Birth_date, gender, and Relationship (to the employee). Two dependents of two distinct employees may, by chance, have the same values for Name, Birth_date, gender, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the particular employee

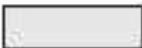
entity to which each dependent is related. Each employee entity is said to own the dependent entities that are related to it.

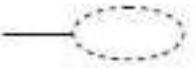
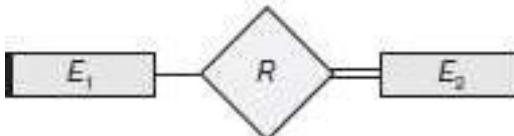
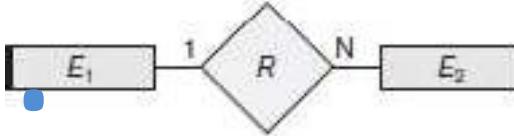
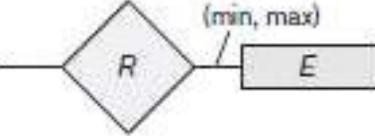
A weak entity type always has a total participation constraint (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity. A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are related to the same owner entity. In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key.

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

3.6 ER Diagrams, Naming Conventions, and Design Issues

3.6.1 Summary of Notation for ER Diagrams

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute

Symbol	Meaning	3.6.2 Prope r Nami ng of Sche ma Const ructs ■ C hoose names that convey, as much as possibl
	Multivalued Attribute	
	Composite Attribute	
	Derived Attribute	
	Total Participation of E2 in R	
	Cardinality Ratio 1:N for E1-E2 in R	
	Structural Constraint (min, max) on Participation of E in R	

e, the meanings attached to the different constructs in the schema.

- Use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type
- In ER diagrams, entity type and relationship type names are uppercase letters, attribute names have their initial letter capitalized, and role names are lowercase letters.
- As a general practice, given a narrative description of the database requirements, the nouns appearing in the narrative tend to give rise to entity type names, and the verbs tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.

- Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom.

3.6.3 Design Choices for ER Conceptual Design

In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached. Some of the refinements that are often used include the following:

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

3.6.4 Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. One alternative ER notation for specifying structural constraints on relationships, which replaces the cardinality ratio (1:1, 1:N, M:N) and single/double line notation for participation constraints. This notation involves associating a pair of integer numbers (min, max) with each *participation* of an entity type *E* in a relationship type *R*, where $0 \leq \text{min} \leq \text{max}$ and $\text{max} \geq 1$.

The numbers mean that for each entity *e* in *E*, *e* must participate in at least min and at most max relationship instances in *R* at any point in time. In this method, $\text{min} = 0$ implies partial participation, whereas $\text{min} > 0$ implies total participation.

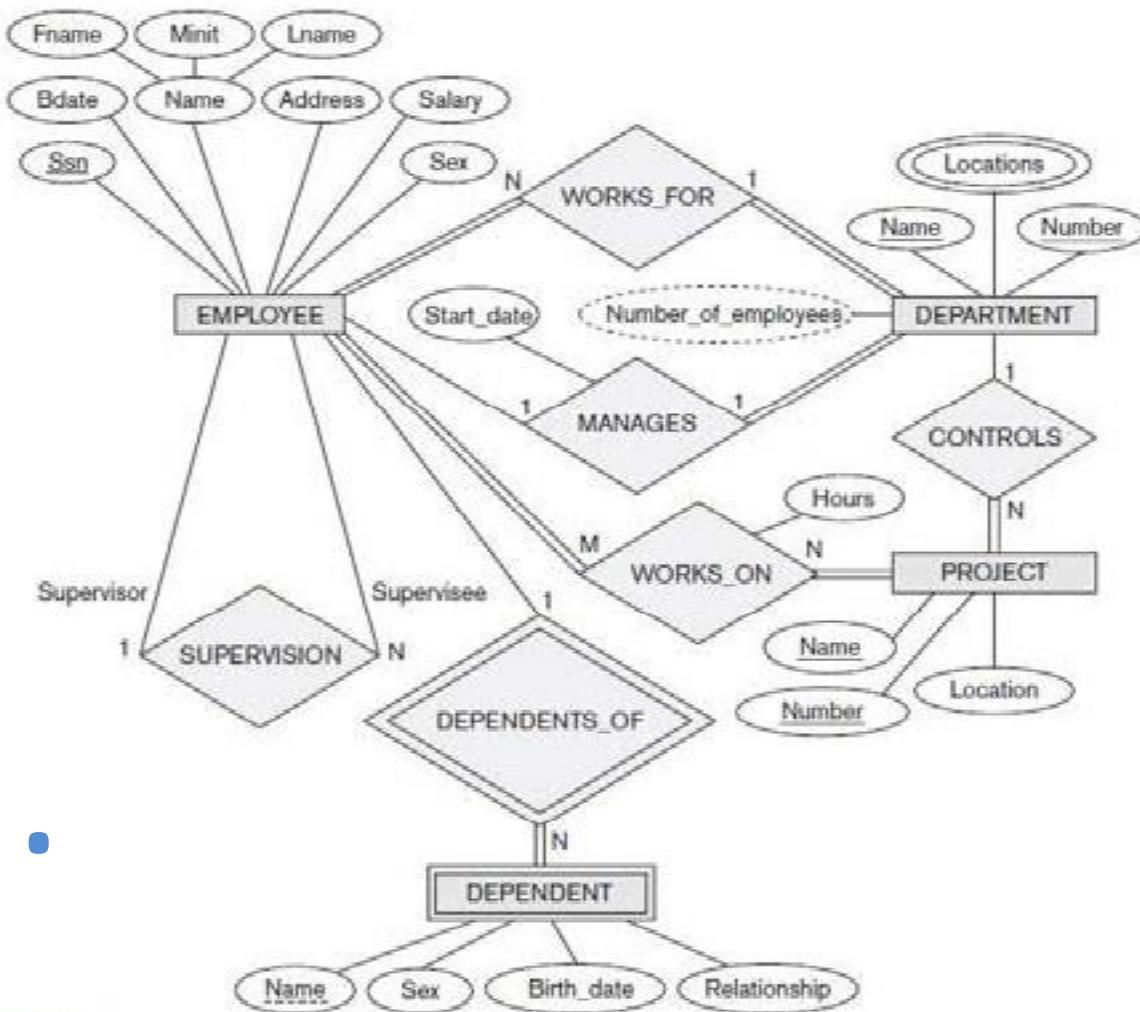


Figure 3.6.4 (a) : ER diagram for Company Database

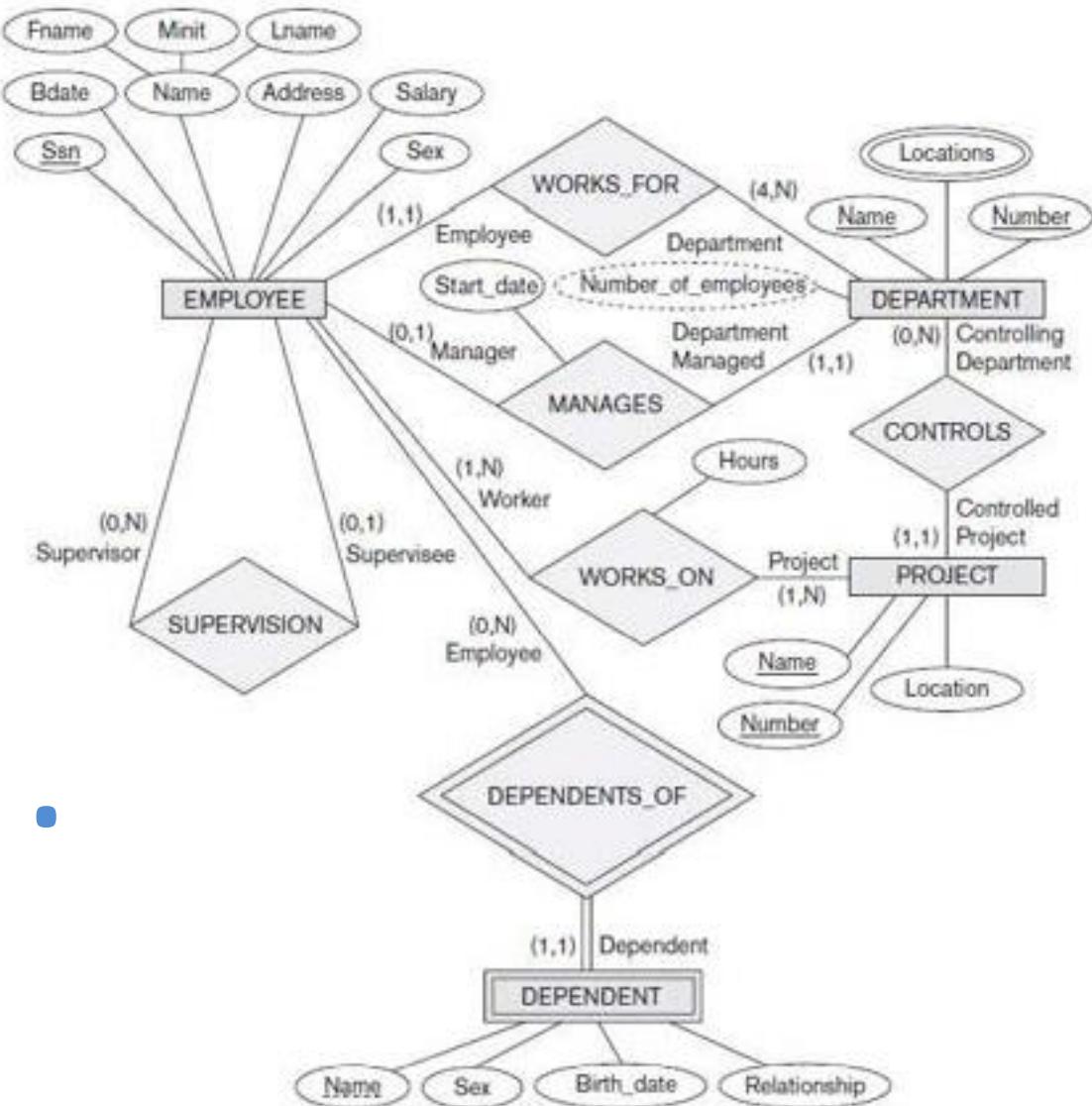


Figure 3.6.4(b): ER diagram for Company Database (using alternative notation)

3.7 Relationship Types of Degree Higher than Two

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

A relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type

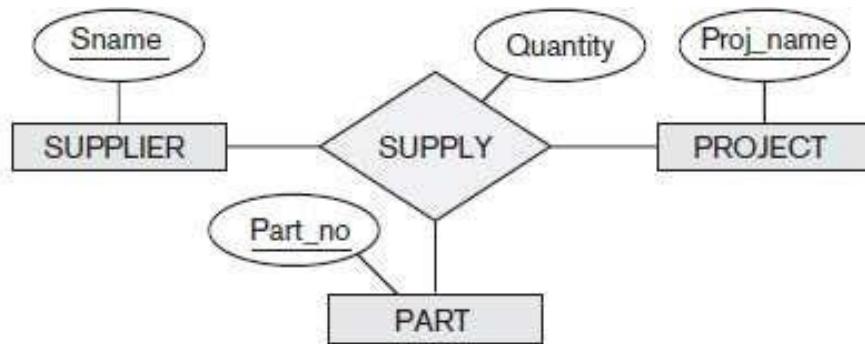


Fig 3.7.1(a): The SUPPLY relationship

Figure 3.7.1(a) shows the ER diagram notation for a ternary relationship. SUPPLY is a set of relationship instances (s, j, p) , where s is a SUPPLIER who is currently supplying a PART p to a PROJECT j .

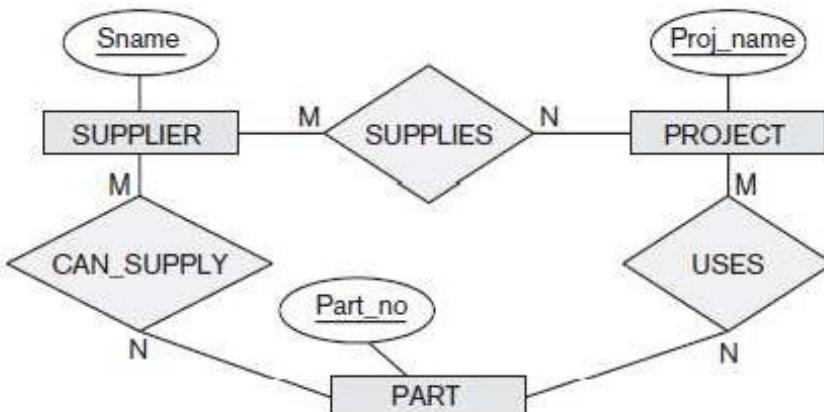


Fig 3.7.1 (b) ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES

Figure 3.7.1(b) shows an ER diagram for three binary relationship types **CAN_SUPPLY**, **USES**, and **SUPPLIES**. **CAN_SUPPLY** between **SUPPLIER** and **PART**, includes an instance (s, p)

whenever supplier s can supply part p (to any project). USES between PROJECT and PART, includes an instance (j, p) whenever project j uses part p. SUPPLIES between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s supplies some part to project j.

Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships.

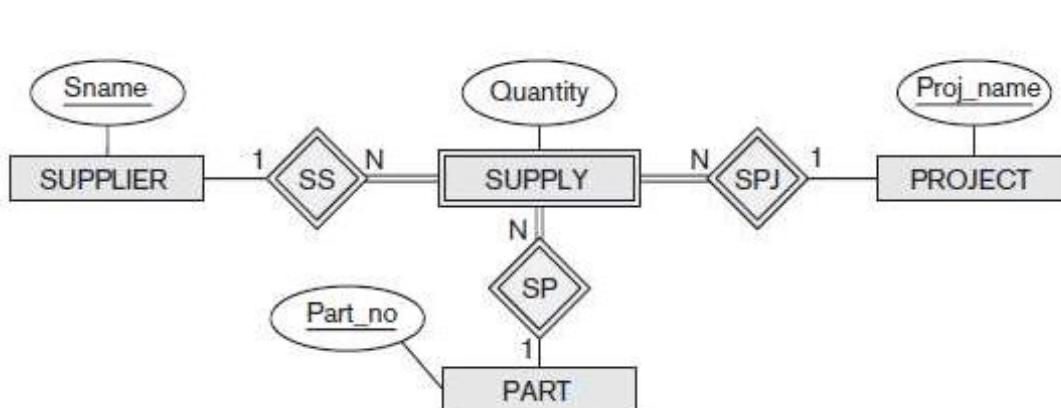


Fig
3.7.
1(c
):
SU
PP
LY
rep
res
ent
ed
as a
we

The three participating entity types SUPPLIER, PART, and PROJECT are called the owner entity types. Hence, an entity in the weak entity type SUPPLY is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

3.7.2

Constraints on Ternary (or Higher-Degree) Relationships

There are two notations for specifying structural constraints on n-ary relationships

1. based on the cardinality ratio notation of binary relationships displayed
 - 1, M, or N is specified on each participation arc (both M and N symbols stand for many or any number)
2. based on the (min, max) notation
 - specifies that each entity is related to at least min and at most max relationship instances in the relationship set

3.8 Specialization and Generalization

3.8.1 Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass.

For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the job type of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the method of pay.

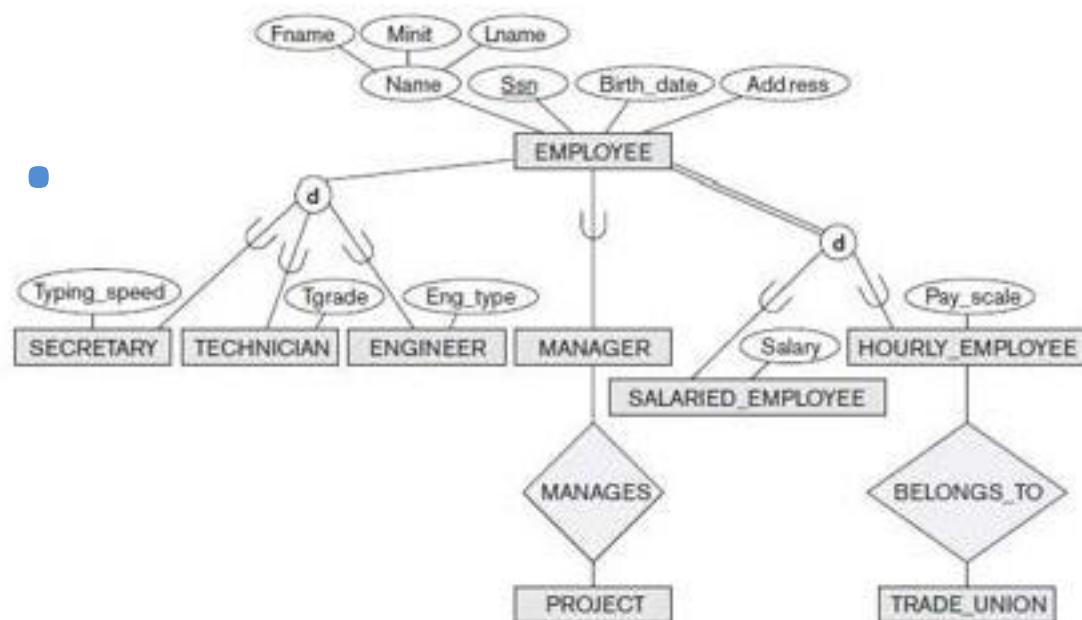


Figure 3.8.1(a): EER diagram notation to represent subclasses and specialization.

Figure 3.8.1(a) shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.⁵ Attributes that apply only to entities of a particular subclass such as TypingSpeed of

SECRETARY are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass.

Similarly, a subclass can participate in **specific relationship types**, such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Figure Figure 3.8.1(b).

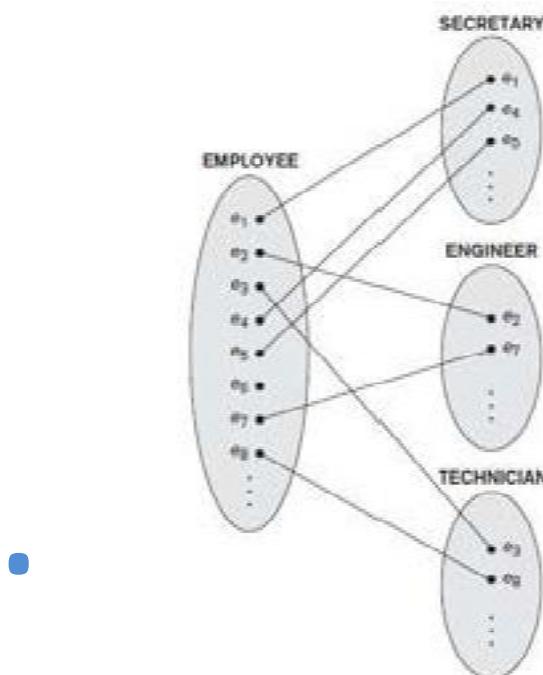


Figure 3.8.1(b): Instances of a specialization

Figure 3.8.1 (b) shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. An entity that belongs to a subclass represents the same real-world entity as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e1 is shown in both EMPLOYEE and SECRETARY. There are two main reasons for including class/subclass relationships and specializations in a data model.

- The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass.
- The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by

creating the subclass `HOURLY_EMPLOYEE` of `EMPLOYEE` and relating the subclass to an entity type `TRADE_UNION` via the `BELONGS_TO` relationship type

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

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

3.8.2 Generalization

Generalization process can be viewed as being functionally the inverse of the specialization process. It is a process of defining a generalized entity type from the given entity types. Generalization is the reverse process of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass**

For example, consider the entity types CAR and TRUCK shown in Figure 3.8.2(a).

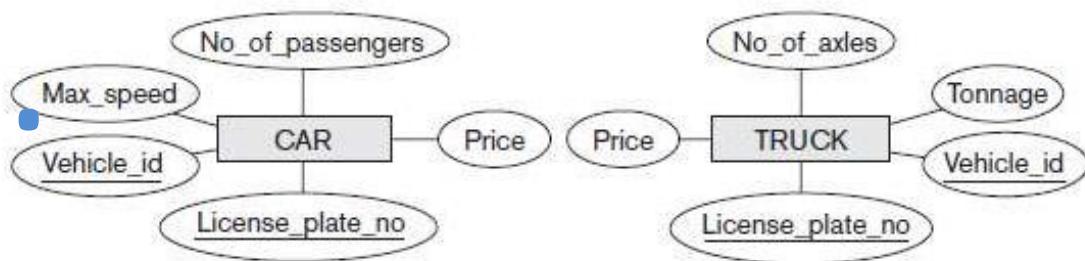


Figure 3.8.2(a): Two entity types, CAR and TRUCK

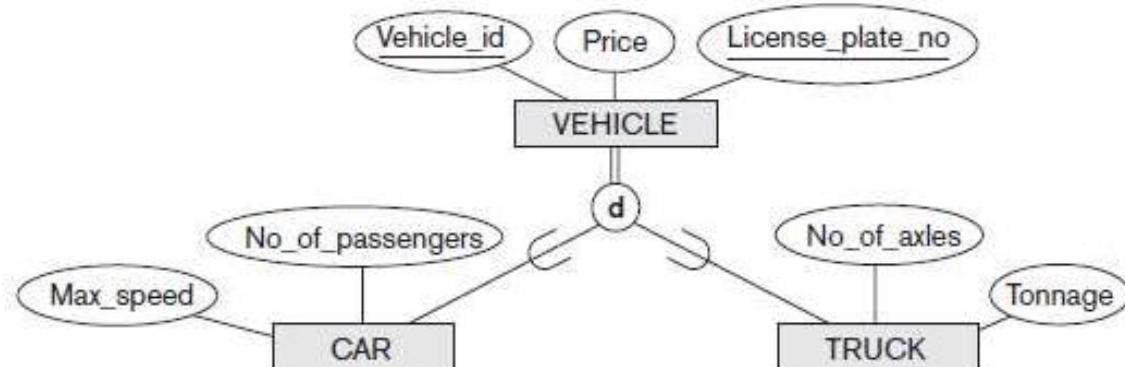


Figure 3.8.2(b): Generalizing CAR and TRUCK into the superclass VEHICLE.

Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 3.8.2(b). Both CAR and TRUCK are now subclasses of the generalized superclass VEHICLE.

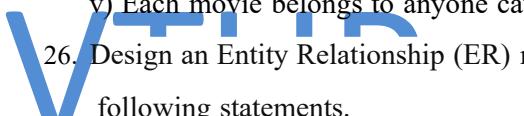
A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization.



Question Bank

1. Define the following terms:
 - i) data ii) database iii) DBMS iv) program-data independence v) Canned transaction
2. Define the database and briefly explain the implicit properties of the database.
3. Discuss the main Characteristics of the database approach and how does it differ from Traditional file systems?
4. What are the different types of database end users? Discuss the main activities of each.
5. Briefly discuss the advantages of using the DBMS.
6. Define the following terms:
 - i) data mode ii) database schema iii) database state iv) schema diagram
7. Describe the three-schema architecture. Why do we need mappings between schema levels?
8. What is the difference between logical data independence and physical data independence?
9. What is the difference between procedural and nonprocedural DMLs?
10. Discuss the various database languages.
11. Discuss the different types of user-friendly interfaces and the types of users who typically use each.
12. Explain the component modules of DBMS and their interaction with the help of a diagram.
13. Discuss some types of database utilities and tools and their functions.
14. Explain two-tier and three-tier architecture.
15. Discuss the classification of database management systems.
16. Explain with a neat diagram, the phases of database design.
17. Define the following terms:
 - i) Entity ii) attribute iii) entity type iv) entity set v) key attribute vi) value set v) degree of a relationship type vi) role names vii) cardinality ratio viii) participation constraints
18. Explain the different types of attributes that occur in an ER model with an example.
19. What is meant by a recursive relationship type? Give some examples of recursive relationship types.
20. What is a weak entity type? Explain the role of partial key in the design of weak entity type.
21. List symbols used in ER diagram and their meaning.

22. Discuss the naming conventions used for ER schema diagrams.
23. Explain with an example specialization and generalization.
24. Design an ER diagram for an insurance company. Assume suitable entity types like CUSTOMER, AGENT, BRANCH, POLICY, PAYEMENT and the relationship between them.
25. Design an ER - diagram for the Movie - database considering the following requirements:
- Each Movie is identified by its title and year of release, it has length in minutes and can have zero or more quotes, language.
 - Production companies are identified by Name, they have address, and each production company can produce one or more movies.
 - Actors are identified by Name and Date of Birth, they can act in one or more movies and each actor has a role in a movie.
 - Directors are identified by Name and Date of Birth, so each Director can direct one or more movie and each movie can be directed by one or more Directors.
 - Each movie belongs to anyone category like Horror, action, Drama, etc.

 26. Design an Entity Relationship (ER) model for a college database . Say we have the following statements.

- A college contains many departments
 - Each department can offer any number of courses
 - Many instructors can work in a department
 - An instructor can work only in one department
 - For each department there is a Head
 - An instructor can be head of only one department
 - Each instructor can take any number of courses
 - A course can be taken by only one instructor
 - A student can enroll for any number of courses
 - Each course can have any number of students
27. Consider the following set of requirements for a UNIVERSITY database that is used to keep track of students' transcripts
- The university keeps track of each student's name, student number, Social Security number, current address and phone number, permanent address and phone number, birth date, sex, class (freshman, sophomore, ..., graduate), major department, minor department

(if any), and degree program (B.A., B.S., ..., Ph.D.). Some user applications need to refer to the city, state, and ZIP Code of the student's permanent address and to the student's last name. Both Social Security number and student number have unique values for each student.

- b. Each department is described by a name, department code, office number, office phone number, and college. Both name and code have unique values for each department.
- c. Each course has a course name, description, course number, number of semester hours, level, and offering department. The value of the course number is unique for each course.
- d. Each section has an instructor, semester, year, course, and section number. The section number distinguishes sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the number of sections taught during each semester.
- e. A grade report has a student, section, letter grade, and numeric grade (0, 1, 2, 3, or 4).

Design an ER schema for this application, and draw an ER diagram for the schema. Specify key attributes of each entity type, and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

- 28. Write ER diagram for Airline reservation and Ban database

Module 2

Chapter 1: The Relational Data Model

Introduction

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a mathematical relation—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems. Current popular relational DBMSs (RDBMSs) include DB2 and Informix Dynamic Server (from IBM), Oracle and Rdb (from Oracle), Sybase DBMS (from Sybase) and SQLServer and Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

1.1 Relational Model Concepts

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

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

For example, in STUDENT relation because each row represents facts about a particular student entity. The column names—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 the formal relational model terminology, a row is called a tuple, a column header is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is represented by a domain of possible values.

1.1.1 Domains, Attributes, Tuples, and Relations

Domain

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

Some examples of domains:

- Usa_phone_numbers: The set of ten-digit phone numbers valid 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.
- Employee_ages: Possible ages of employees in a company; each must be an integer value between 15 and 80.

The preceding are called logical definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (ddd)ddddddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80.

Attribute

An attribute A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by $\text{dom}(A_i)$.

Tuple

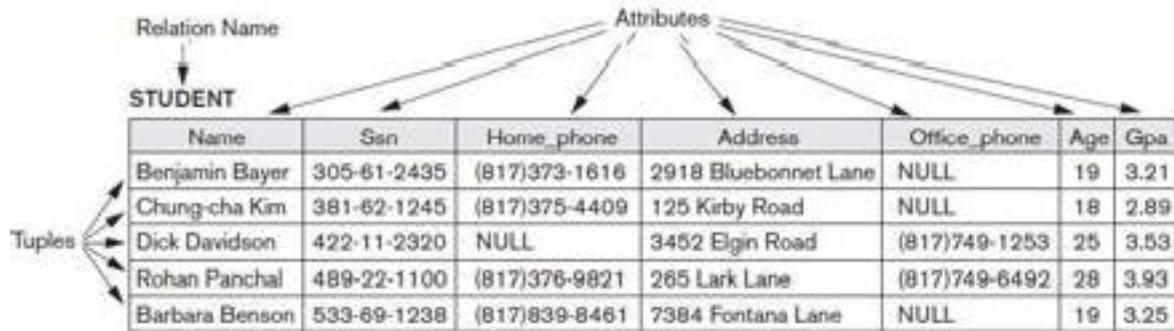
Mapping from attributes to values drawn from the respective domains of those attributes. Tuples are intended to describe some entity (or relationship between entities) in the miniworld.

Example: a tuple for a PERSON entity might be

{ Name --> "smith", Gender --> Male, Age --> 25 }

Relation

A named set of tuples all of the same form i.e., having the same set of attributes.



Relation schema

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

The **degree (or arity)** of a relation is the number of attributes n of its relation schema. A relation of degree seven, which stores information about university students, would contain seven attributes describing each student, as follows:

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

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

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

Domains for some of the attributes of the STUDENT relation:

$\text{dom}(\text{Name}) = \text{Names}$; $\text{dom}(\text{Ssn}) = \text{Social_security_numbers}$;

$\text{dom}(\text{HomePhone}) = \text{USA_phone_numbers}$, $\text{dom}(\text{Office_phone}) = \text{USA_phone_numbers}$,

Relation (or relation state)

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

The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

1.1.2 Characteristics of Relations

1. Ordering of Tuples in a Relation

A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation.

2. Ordering of Values within a Tuple and an Alternative Definition of a Relation

The order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained. An alternative definition of a relation can be given, making the ordering of values in a tuple unnecessary. In this definition A **relation schema** $R(A_1, A_2, \dots, A_n)$, set of attributes and a **relation state** $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \dots, t_m\}$, where each tuple t_i is a **mapping** from R to D .

According to this definition of tuple as a mapping, a **tuple** can be considered as a set of ($<\text{attribute}>$, $<\text{value}>$) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is not important, because the attribute name appears with its value.

3. Values and NULLs in the Tuples

Each value in a tuple is atomic. NULL values are used to represent the values of attributes that may be unknown or may not apply to a tuple. For example some STUDENT tuples have NULL for their office phones because they do not have an office. Another student has a NULL for home phone. In general, we can have several meanings for NULL values, such as **value unknown**, **value exists but is not available**, or **attribute does not apply** to this tuple (also known as **value undefined**).

4. Interpretation (Meaning) of a Relation

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

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate.

1.1.3 Relational Model Notation

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

1.2 Relational Model Constraints and Relational Database Schemas

Constraints are restrictions 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. Inherent model-based constraints or implicit constraints
 - Constraints that are inherent in the data model.
 - The characteristics of relations are the inherent constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint.
2. Schema-based constraints or explicit constraints
 - Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL.
 - The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.
3. Application-based or semantic constraints or business rules
 - Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs.

- Examples of such constraints are the salary of an employee should not exceed the salary of the employee's supervisor and the maximum number of hours an employee can work on all projects per week is 56.

1.2.1 Domain Constraints

Domain Constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and doubleprecision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types.

1.2.2 Key Constraints and Constraints on NULL Values

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. There are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.

- Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R , we have the constraint that: $t_1[\text{SK}] \neq t_2[\text{SK}]$. such set of attributes SK is called a **superkey** of the relation schema R

superkey

A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes.

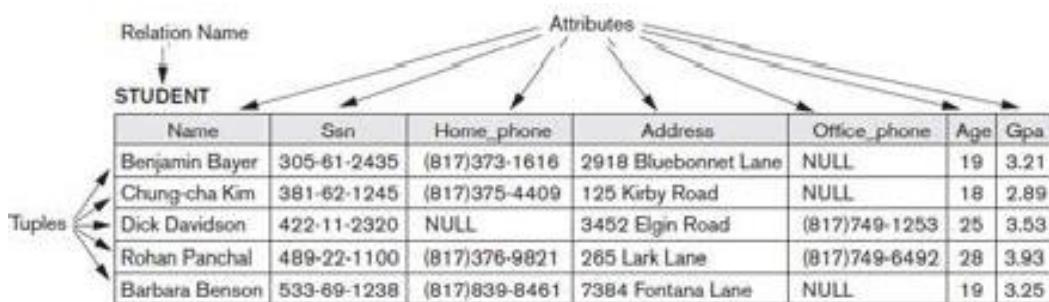
Key

A **key** K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R anymore. Hence, a key satisfies two properties:

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

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

Example: Consider the STUDENT relation



- The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn
- Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey
- The superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey

In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

Candidate key

A relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation has two candidate keys: License_number and Engine_serial_number

CAR				
License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Primary key

It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined. Other candidate keys are designated as **unique keys** and are not underlined.

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

1.2.3 Relational Databases and Relational Database Schemas

A Relational database schema S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of integrity constraints IC.

Example of relational database schema:

COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT,
WORKS_ON, DEPENDENT}



Figure1.2.3 (a): Schema diagram for the COMPANY relational database schema.

The underlined attributes represent primary keys

A Relational database state is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$. Each r_i is a state of R and such that the r_i relation states satisfy integrity constraints specified in IC.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	gender	Salary	Super_ssn	Dno
John	B	Smith	123456789	1985-01-09	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	1989-03-29	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

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

DEPT_LOCATIONS

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

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

Figure 1.2.3(b) :One possible database state for the COMPANY relational database schema.

A database state that does not obey all the integrity constraints is called **Invalid state** and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **Valid state**

Attributes that represent the same real-world concept may or may not have identical names in different relations. For example, 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.

Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different realworld concepts project names and department names.

1.2.4 Integrity, Referential Integrity, and Foreign Keys

Entity integrity constraint

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

Key constraints and entity integrity constraints are specified on individual relations.

Referential integrity constraint

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 COMPANY database, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2 .

A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:

1. Attributes in FK have the same domain(s) as the primary key attributes PK of R_2 ; the 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, we have $t_1[\text{FK}] = t_2[\text{PK}]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2 .

In this definition, 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.

1.2.5 Other Types of Constraints

Semantic integrity constraints

Semantic integrity constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose constraint specification language. Examples of such constraints are the salary of an employee should not exceed the salary of the employee's supervisor and the maximum number of hours an employee can work on all projects per week is 56. Mechanisms called **triggers** and **assertions** can be used. In SQL, CREATE ASSERTION and CREATE TRIGGER statements can be used for this purpose.

Functional dependency constraint

Functional dependency constraint establishes a functional relationship among two sets of attributes X and Y. This constraint specifies that the value of X determines a unique value of Y in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$. We use functional dependencies and other types of dependencies as tools to analyze the quality of relational designs and to “normalize” relations to improve their quality.

State constraints(static constraints)

Define the constraints that a valid state of the database must satisfy

Transition constraints(dynamic constraints)

Define to deal with state changes in the database

1.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into **retrievals** and **updates**

There are three basic operations that can change the states of relations in the database:

1. Insert - used to insert one or more new tuples in a relation
2. Delete- used to delete tuples
3. Update (or Modify)- used to change the values of some attributes in existing tuples

Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.

1.3.1 The Insert Operation

The Insert operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R . Insert can violate any of the four types of constraints

1. **Domain constraints** : if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type
2. **Key constraints** : if a key value in the new tuple t already exists in another tuple in the relation $r(R)$
3. **Entity integrity**: if any part of the primary key of the new tuple t is NULL
4. **Referential integrity** : if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation

Examples:

1. Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, NULL, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4>

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

2. Operation:

Insert <‘Alicia’, ‘J’, ‘Zelaya’, ‘999887777’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, ‘987654321’, 4>

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

3. Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windswept, Katy, TX’, F, 28000, ‘987654321’, 7>

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

4. Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windy Lane,Katy, TX’, F, 28000, NULL, 4>

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

If an insertion violates one or more constraints, the default option is to reject the insertion. It would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to correct the reason for rejecting the insertion

1.3.2 The Delete Operation

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

Examples:

1. Operation:

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

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

2. Operation:

Delete the EMPLOYEE tuple with Ssn = ‘999887777’.

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

3. Operation:

Delete the EMPLOYEE tuple with Ssn = ‘333445555’

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

Several options are available if a deletion operation causes a violation

1. restrict - is to reject the deletion
2. cascade, is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted

3. Set null or set default - is to modify the referencing attribute values that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple.

1.3.3 The Update Operation

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

Examples:

1. Operation:

Update the salary of the EMPLOYEE tuple with Ssn = ‘999887777’ to 28000.

Result: Acceptable.

2. Operation:

Update the Dno of the EMPLOYEE tuple with Ssn = ‘999887777’ to 7.

Result: Unacceptable, because it violates referential integrity.

3. Operation:

Update the Ssn of the EMPLOYEE tuple with Ssn = ‘999887777’ to ‘987654321’.

Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn

Updating an attribute that is neither part of a primary key nor of a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain.

1.3.4 The Transaction Concept

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

Chapter 2: Relational Algebra

2.1 Introduction

Relational algebra is the basic set of operations for the relational model. These operations enable a user to specify basic retrieval requests as relational algebra expressions. The result of an operation is a new relation, which may have been formed from one or more input relations.

The relational algebra is very important for several reasons

- First, it provides a formal foundation for relational model operations.
- Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs)
- Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs

2.2 Unary Relational Operations: SELECT and PROJECT

2.2.1 The SELECT Operation

The SELECT operation denoted by σ (sigma) is used to select a subset of the tuples from a relation based on a selection condition. The selection condition acts as a filter that keeps only those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition.

The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples— those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

In general, the select operation is denoted by

$$\sigma \text{ <selection condition>} (R)$$

where,

- the symbol σ is used to denote the select operator
- the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
- tuples that make the condition true are selected
 - appear in the result of the operation
- tuples that make the condition false are filtered out
 - discarded from the result of the operation

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 of R ,

<comparison op> is one of the operators $\{=, <, \leq, >, \geq, \neq\}$, and

<constant value> is a constant value from the attribute domain

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

Examples:

1. Select the EMPLOYEE tuples whose department number is 4.

$\sigma_{DNO = 4} (\text{EMPLOYEE})$

2. Select the employee tuples whose salary is greater than \$30,000.

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

3. Select the tuples for all employees who either work in department 4 and make over \$25,000 per year, or work in department 5 and make over \$30,000

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

The result of a SELECT operation can be determined as follows:

- The <selection condition> is applied independently to each individual tuple t in R

If the condition evaluates to TRUE, then tuple t is selected. All the selected tuples appear in the result of the SELECT operation

The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 AND cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is

FALSE.

- (cond1 OR cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is

FALSE.

- (\neg cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is unary; that is, it is applied to a single relation. The degree of the relation resulting from a SELECT operation is the same as the degree of R. The number of tuples in the resulting relation is always less than or equal to the number of tuples in R. That is,

$$|\sigma_c(R)| \leq |R| \text{ for any condition } C$$

The fraction of tuples selected by a selection condition is referred to as the selectivity of the condition.

The SELECT operation is commutative; that is,

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

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

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

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

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

would be to the following SQL query:

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



2.2.2 The PROJECT Operation

The PROJECT operation denoted by $\pi(\rho)$ selects certain columns from the table and discards the other columns. Used when we are interested in only certain attributes of a relation. The result of the PROJECT operation can be visualized as a vertical partition of the relation into two relations:

- one has the needed columns (attributes) and contains the result of the operation
- the other contains the discarded columns

The general form of the PROJECT operation is

$$\pi_{\langle \text{attributelist} \rangle}(R)$$

where

$\pi(\rho)$ - symbol used to represent the PROJECT operation,

$\langle \text{attributelist} \rangle$ - desired sublist of attributes from the attributes of relation R.

The result of the PROJECT operation has only the attributes specified in $\langle \text{attribute list} \rangle$ in the same order as they appear in the list. Hence, its degree is equal to the number of attributes in $\langle \text{attribute list} \rangle$.

Example :

1. To list each employee's first and last name and salary we can use the PROJECT operation as follows:

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

If the attribute list includes only nonkey attributes of R , duplicate tuples are likely to occur. The result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**. For example, consider the following PROJECT operation:

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

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

The tuple $\langle 'F', 25000 \rangle$ appears only once in resulting relation even though this combination of values appears twice in the EMPLOYEE relation.

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R . Commutativity does not hold on PROJECT

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

as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$; otherwise, the left-hand side is an incorrect expression.

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

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

would correspond to the following SQL query:

SELECT DISTINCT gender, Salary FROM EMPLOYEE

2.2.3 Sequences of Operations and the RENAME Operation

For most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single relational algebra expression by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

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

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

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$\text{DEP5_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\text{DEP5_EMPS})$$

We can also use this technique to **rename** the attributes in the intermediate and result relations. To rename the attributes in a relation, we simply list the new attribute names in parentheses

$$\text{TEMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$

$$R(\text{First_name}, \text{Last_name}, \text{Salary}) \leftarrow \pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\text{TEMP})$$

TEMP

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

R

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

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

We can also define a formal RENAME operation—which can rename either the relation name or the attribute names, or both—as a unary operator.

The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

1. $\rho_{S(B_1, B_2, \dots, B_n)}(R)$ ρ (rho) – RENAME operator
2. $\rho_S(R)$ S – new relation name
3. $\rho_{(B_1, B_2, \dots, B_n)}(R)$ B_1, B_2, \dots, B_n - new attribute names

The first expression renames both the relation and its attributes. Second renames the relation only and the third renames the attributes only. If the attributes of R are (A_1, A_2, \dots, A_n) in that order, then each A_i is renamed as B_i .

Renaming in SQL is accomplished by aliasing using AS, as in the following example:

```
SELECT E.Fname AS First_name,
```

```
    E.Lname AS Last_name,
```

```
    E.Salary AS Salary
```

```
FROM EMPLOYEE AS E
```

```
WHERE E.Dno=5,
```

2.3 Relational Algebra Operations from Set Theory

2.3.1 The UNION, INTERSECTION, and MINUS Operations

- **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 .

Example: Consider the the following two relations: STUDENT & INSTRUCTOR

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

STUDENT \cup INSTRUCTOR

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

STUDENT \cap INSTRUCTOR

Fn	Ln
Susan	Yao
Ramesh	Shah

STUDENT – INSTRUCTOR

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

INSTRUCTOR – STUDENT

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

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

DEP5_EMPS $\leftarrow \sigma_{Dno=5}(\text{EMPLOYEE})$

RESULT1 $\leftarrow \pi_{Ssn}(\text{DEP5_EMPS})$

RESULT2(Ssn) $\leftarrow \pi_{\text{Super_ssn}}(\text{DEP5_EMPS})$

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

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	gender	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	Y	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4

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

Single relational algebra expression:

$$\text{Result} \leftarrow \pi_{\text{Ssn}} (\sigma_{\text{Dno}=5} (\text{EMPLOYEE})) \cup \pi_{\text{Super_ssn}} (\sigma_{\text{Dno}=5} (\text{EMPLOYEE}))$$

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

Two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be union compatible (or type compatible) if they have the same degree n and if $\text{dom}(A_i) = \text{dom}(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

Both UNION and INTERSECTION are *commutative operations*; that is,

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

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

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

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

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

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

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

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations

2.3.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

The CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN denoted by \times —is a binary set operation, but the relations on which it is applied do *not* have to be union compatible. This set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).

In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S . Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

Example: suppose that we want to retrieve a list of names of each female employee's dependents.

```

FEMALE_EMPS ←  $\sigma_{\text{gender}=\text{'F'}}$ (EMPLOYEE)
EMPNAMESS ←  $\pi_{\text{Fname}, \text{Lname}, \text{Ssn}}$ (FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMESS  $\times$  DEPENDENT
ACTUAL_DEPENDENTS ←  $\sigma_{\text{Ssn}=\text{Essn}}$ (EMP_DEPENDENTS)
RESULT ←  $\pi_{\text{Fname}, \text{Lname}, \text{Dependent_name}}$ (ACTUAL_DEPENDENTS)

```

FEMALE_EMPS

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

EMP_DEPENDENTS

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

EMPNAMESS

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

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

ACTUAL_DEPENDENTS

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

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT related tuples only from the two relations by specifying an appropriate selection condition after the Cartesian product.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables

2.4 Binary Relational Operations: JOIN and DIVISION

2.4.1 The JOIN Operation

The JOIN operation, denoted by \bowtie , is used to combine related tuples from two relations into single “longer” tuples. It allows us to process relationships among relations. The general form of a JOIN operation on two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is

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

Example: 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

$\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn} = \text{Ssn}} \text{EMPLOYEE}$
 $\text{RESULT} \leftarrow \pi_{\text{Dname}, \text{Lname}, \text{Fname}}(\text{DEPT_MGR})$

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	...

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ in that order. Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition. This is the main difference between CARTESIAN PRODUCT and JOIN. In JOIN, only combinations of tuples satisfying the join condition appear in the result, whereas in the CARTESIAN PRODUCT all combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples.

Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q as a single combined tuple. A general join condition is of the form

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

where each <condition> is of the form $A_i \theta B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$.

A_i

JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE do not appear in the result.

2.4.2 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**. In the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple.

For example the values of the attributes `Mgr_ssn` and `Ssn` are identical in every tuple of `DEPT_MGR` (the EQUIJOIN result) because the equality join condition specified on these two attributes requires the values to be identical in every tuple in the result.

The standard definition of **NATURAL JOIN** requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first. Suppose we want to combine each `PROJECT` tuple with the `DEPARTMENT` tuple that controls the project. first we rename the `Dnumber` attribute of `DEPARTMENT` to `Dnum` —so that it has the same name as the `Dnum` attribute in `PROJECT` and then we apply NATURAL JOIN:

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

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

DEPT $\leftarrow \rho_{(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})}(\text{DEPARTMENT})$

PROJ_DEPT \leftarrow PROJECT * DEPT

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

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

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

DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS**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

In general, the join condition for NATURAL JOIN is constructed by equating each pair of join attributes that have the same name in the two relations and combining these conditions with **AND**. If no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples.

A more general, but nonstandard definition for NATURAL JOIN is

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

where,

$\langle \text{list1} \rangle$: list of i attributes from R ,

$\langle \text{list2} \rangle$: list of i attributes from S

The lists are used to form equality comparison conditions between pairs of corresponding attributes and then the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R $\langle \text{list1} \rangle$ is kept in the result Q .

In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{<\text{join condition}>} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called join selectivity, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**. Informally, an inner join is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an n-way join. For example, consider the following three-way join:

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

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

In SQL, JOIN can be realized in several different ways

- The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions.
- The second way is to use a nested relation
- Another way is to use the concept of joined tables

2.4.3 A Complete Set of Relational Algebra Operations

The set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ is a complete set; that is, any of the other original relational algebra operations can be expressed as a sequence of operations from this set.

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

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

As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation,

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

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also not strictly necessary for the expressive power of the relational algebra.

2.4.4 The DIVISION Operation

The DIVISION operation, denoted by \div , 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:

$$\text{SSN_PNOS} \leftarrow \pi_{\text{Essn}, \text{Pno}}(\text{WORKS_ON})$$

- Next, create a relation that includes a tuple $\langle \text{Pno}, \text{Essn} \rangle$ whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$\begin{aligned} \text{SMITH} &\leftarrow \sigma_{\text{Fname}=\text{'John'} \text{ AND } \text{Lname}=\text{'Smith'}}(\text{EMPLOYEE}) \\ \text{SMITH_PNOS} &\leftarrow \pi_{\text{Pno}}(\text{WORKS_ON} \bowtie_{\text{Essn}=\text{Ssn}} \text{SMITH}) \end{aligned}$$

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

$$\begin{aligned} \text{SSNS}(\text{Ssn}) &\leftarrow \text{SSN_PNOS} \div \text{SMITH_PNOS} \\ \text{RESULT} &\leftarrow \pi_{\text{Fname}, \text{Lname}}(\text{SSNS} * \text{EMPLOYEE}) \end{aligned}$$

(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
666665555	20

SSNS

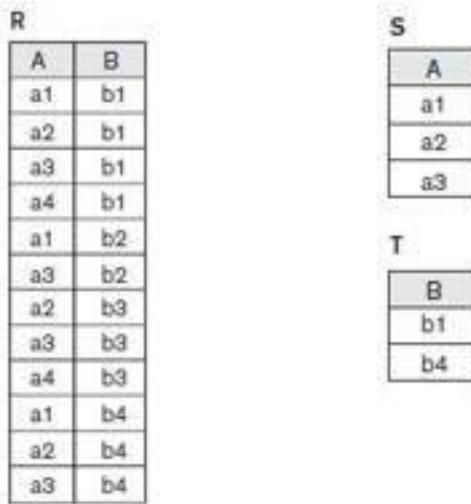
Ssn
123456789
453453453

SMITH_PNOS

Pno
1
2

In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of R are a subset of the attributes of S ; that is, $X \subseteq Z$. Let Y be the set of attributes of R that are not attributes of S ; that is, $Y = Z - X$ (and hence $Z = X \cup Y$). The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for every tuple t_S in S . This means that, for a tuple t to appear in the result T of t

Figure below illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$.



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	

The tuples (values) $b1$ and $b4$ appear in R in combination with all three tuples in S ; that is why they appear in the resulting relation T . All other values of B in R do not appear with all the tuples in S and are not selected: $b2$ does not appear with $a2$, and $b3$ does not appear with $a1$.

The DIVISION operation can be expressed as a sequence of π , \times , and $-$ operations as follows:

$$\begin{aligned} T1 &\leftarrow \pi_Y(R) \\ T2 &\leftarrow \pi_Y((S \times T1) - R) \\ T &\leftarrow T1 - T2 \end{aligned}$$

Table 6.1 Operations of Relational Algebra

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

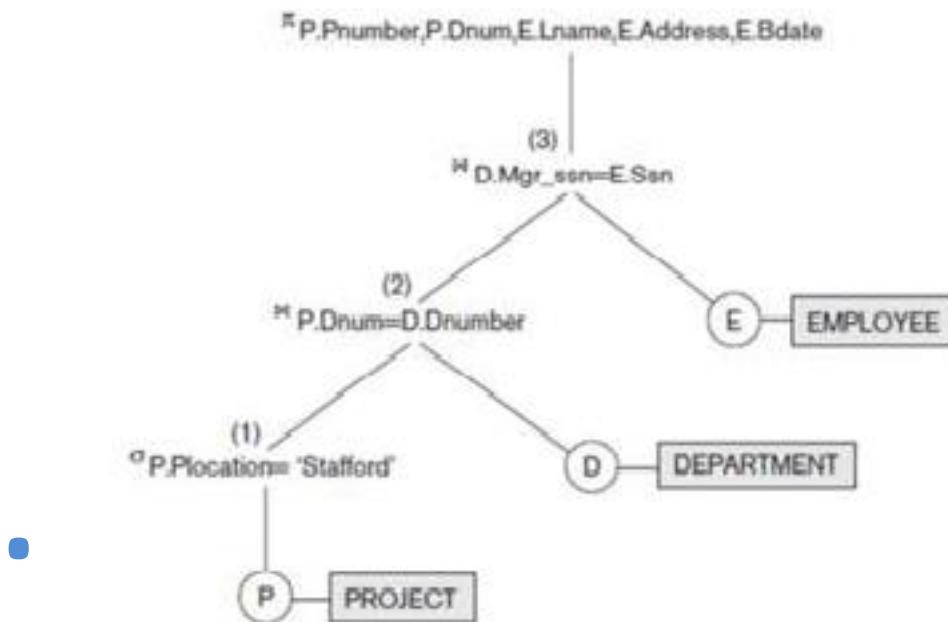
2.4.5 Notation for Query Trees

Query tree (query evaluation tree or query execution tree) is used in relational systems to represent queries internally. A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes.

An execution of the query tree consists of executing an internal node operation whenever its operands represented by its child nodes are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Example: For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

$$\begin{array}{l} \pi_{Pnumber, Dnum, Lname, Address, Bdate}((\sigma_{Plocation='Stafford'}(PROJECT)) \\ \bowtie_{Dnum=Dnumber}(DEPARTMENT) \bowtie_{Mgr_ssn=Ssn}(EMPLOYEE)) \end{array}$$



Leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense. The node marked (1) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on.

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

2.5 Additional Relational Operations

2.5.1 Generalized Projection

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

$$\pi_{F_1, F_2, \dots, F_n}(R)$$

where F_1, F_2, \dots, F_n are functions over the attributes in relation R and may involve arithmetic operations and constant values.

The generalized projection helpful when developing reports where computed values have to be produced in the columns of a query result. For example, consider the relation EMPLOYEE (Ssn, Salary, Deduction, Years_service). A report may be required to show

$$\text{Net Salary} = \text{Salary} - \text{Deduction},$$

$$\text{Bonus} = 2000 * \text{Years_service}, \text{ and}$$

$$\text{Tax} = 0.25 * \text{Salary}.$$

generalized projection combined with renaming :

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



2.5.2 Aggregate Functions and Grouping

Aggregate functions are used in simple statistical queries that summarize information from the database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values. For example, retrieving the average or total salary of all employees or the total number of employee tuples.

Grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. For example, group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

Aggregate function operation can be defined by using the symbol Σ (script F) :

$\langle \text{grouping attributes} \rangle \Sigma \langle \text{function list} \rangle (R)$

Where ,

$\langle \text{grouping attributes} \rangle$: list of attributes of the relation specified in R

$\langle \text{function list} \rangle$: list of ($\langle \text{function} \rangle$ $\langle \text{attribute} \rangle$) pairs.

$\langle \text{function} \rangle$ - such as SUM, AVERAGE, MAXIMUM, MINIMUM,COUNT

$\langle \text{attribute} \rangle$ is an attribute of the relation specified by R

The resulting relation has the grouping attributes plus one attribute for each element in the function list.

Example: To retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes

$\rho_R(Dno, No_of_employees, Average_sal)(Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE))$

The aggregate function operation.

- $\rho_R(Dno, No_of_employees, Average_sal)(Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE))$.
- $Dno \Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE)$.
- $\Sigma COUNT Ssn, AVERAGE Salary (EMPLOYEE)$.

R		
(a)	Dno	No_of_employees
	5	4
	4	3
	1	1
		Average_sal
		33250
		31000
		55000

(b)	Dno	Count_ssn	Average_salary
	5	4	33250
	4	3	31000
	1	1	55000

(c)	Count_ssn	Average_salary
	8	35125

2.5.3 Recursive Closure Operations

Recursive closure operation is applied to a recursive relationship between tuples of the same type, such as the relationship between an employee and a supervisor.

Example : Retrieve all supervisees of an employee e at all levels—that is, all employees e' directly supervised by e , all employees e'' directly supervised by each employee e' , all employees e''' directly supervised by each employee e'' and so on.

```
BORG_SSN ← πSsn(σFname='James' AND Lname='Borg(EMPLOYEE))
SUPERVISION(Ssn1, Ssn2) ← πSsn, Super_ssn(EMPLOYEE)
RESULT1(Ssn) ← πSsn1(SUPERVISION ⋈Ssn2=Ssn BORG_SSN)
```

SUPERVISION

(Borg's Ssn is 888665555)

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

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

-
-
-

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

$$\text{RESULT2}(\text{Ssn}) \leftarrow \pi_{\text{Ssn}_1}(\text{SUPERVISION} \bowtie_{\text{Ssn}_2=\text{Ssn}} \text{RESULT1})$$
RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by Borg's subordinates)

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

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

2.5.4 OUTER JOIN Operations

The JOIN operations match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a matching (or related) tuple are eliminated from the

JOIN result. Tuples with NULL values in the join attributes are also eliminated. This type of join, where tuples with no match are eliminated, is known as an **inner join**.

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation.

For example, suppose that we want a list of all employee names as well as the name of the departments they manage if they happen to manage a department; if they do not manage one, we can indicate it with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by

 to retrieve the result as follows:

$$\begin{aligned} \text{TEMP} &\leftarrow (\text{EMPLOYEE} \bowtie_{\text{Ssn}=\text{Mgr_ssn}} \text{DEPARTMENT}) \\ \text{RESULT} &\leftarrow \pi_{\text{Fname}, \text{Minit}, \text{Lname}, \text{Dname}}(\text{TEMP}) \end{aligned}$$

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



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

A similar operation, **RIGHT OUTER JOIN**, denoted by  , keeps every tuple in the *second*, or right, relation S in the result of $R \bowtie S$.

A third operation, **FULL OUTER JOIN**, denoted by  , keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

2.5.5 The OUTER UNION Operation

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

the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X , are union compatible.

The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$. Two tuples t_1 in R and t_2 in S are said to match if $t_1[X] = t_2[X]$. These will be combined (unioned) into a single tuple in t . Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.

For example, an OUTER UNION can be applied to two relations whose schemas are:

$\text{STUDENT}(\text{Name}, \text{Ssn}, \text{Department}, \text{Advisor})$

$\text{INSTRUCTOR}(\text{Name}, \text{Ssn}, \text{Department}, \text{Rank})$

Tuples from the two relations are matched based on having the same combination of values of the shared attributes— Name , Ssn , Department . All the tuples from both relations are included in the result, but tuples with the same (Name , Ssn , Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute.

A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes. The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

$\text{STUDENT_OR_INSTRUCTOR}(\text{Name}, \text{Ssn}, \text{Department}, \text{Advisor}, \text{Rank})$

2.6 Examples of Queries in Relational Algebra

Query 1. Retrieve the name and address of all employees who work for the ‘Research’ department.

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

As a single in-line expression, this query becomes:

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

Query 2. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

$\text{STAFFORD_PROJS} \leftarrow \sigma_{\text{Plocation}=\text{'Stafford'}}(\text{PROJECT})$
 $\text{CONTR_DEPTS} \leftarrow (\text{STAFFORD_PROJS} \bowtie_{\text{Dnum}=\text{Dnumber}} \text{DEPARTMENT})$
 $\text{PROJ_DEPT_MGRS} \leftarrow (\text{CONTR_DEPTS} \bowtie_{\text{Mgr_sn}=\text{Sn}} \text{EMPLOYEE})$
 $\text{RESULT} \leftarrow \pi_{\text{Pnumber}, \text{Dnum}, \text{Lname}, \text{Address}, \text{Bdate}}(\text{PROJ_DEPT_MGRS})$

Query 3. Find the names of employees who work on all the projects controlled by department number 5.

```

DEPT5_PROJS ←  $\rho_{(Pno)}(\pi_{Pnumber}(\sigma_{Dnum=5}(PROJECT)))$ 
EMP_PROJ ←  $\rho_{(Ssn, Pno)}(\pi_{Esn, Pno}(WORKS\_ON))$ 
RESULT_EMP_SSNS ← EMP_PROJ ÷ DEPT5_PROJS
RESULT ←  $\pi_{Lname, Fname}(RESULT\_EMP\_SSNS * EMPLOYEE)$ 

```

Query 4. Make a list of project numbers for projects that involve an employee whose last name is ‘Smith’, either as a worker or as a manager of the department that controls the project.

```

SMITHS(Essn) ←  $\pi_{Ssn}(\sigma_{Lname='Smith'}(EMPLOYEE))$ 
SMITH_WORKER_PROJS ←  $\pi_{Pno}(WORKS\_ON * SMITHS)$ 
MGRS ←  $\pi_{Lname, Dnumber}(EMPLOYEE \bowtie_{Ssn=Mgr\_ssn} DEPARTMENT)$ 
SMITH_MANAGED_DEPTS(Dnum) ←  $\pi_{Dnumber}(\sigma_{Lname='Smith'}(MGRS))$ 
SMITH_MGR_PROJS(Pno) ←  $\pi_{Pnumber}(SMITH\_MANAGED\_DEPTS * PROJECT)$ 
RESULT ← (SMITH_WORKER_PROJS ∪ SMITH_MGR_PROJS)

```

Query 5. List the names of all employees with two or more dependents.

```

T1(Ssn, No_of_dependents) ←  $\exists_{Esn} \text{COUNT}_{Dependent\_name}(DEPENDENT)$ 
T2 ←  $\sigma_{No\_of\_dependents > 2}(T1)$ 
RESULT ←  $\pi_{Lname, Fname}(T2 * EMPLOYEE)$ 

```

Query 6. Retrieve the names of employees who have no dependents.

```

ALL_EMPS ←  $\pi_{Ssn}(EMPLOYEE)$ 
EMPS_WITH_DEPS(Ssn) ←  $\pi_{Esn}(DEPENDENT)$ 
EMPS_WITHOUT_DEPS ← (ALL_EMPS – EMPS_WITH_DEPS)
RESULT ←  $\pi_{Lname, Fname}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$ 

```

Query 7. List the names of managers who have at least one dependent.

```

MGRS(Ssn) ←  $\pi_{Mgr\_ssn}(DEPARTMENT)$ 
EMPS_WITH_DEPS(Ssn) ←  $\pi_{Esn}(DEPENDENT)$ 
MGRS_WITH_DEPS ← (MGRS ∩ EMPS_WITH_DEPS)
RESULT ←  $\pi_{Lname, Fname}(MGRS\_WITH\_DEPS * EMPLOYEE)$ 

```

Chapter 3: Mapping Conceptual Design into a Logical Design

3.1 Relational Database Design using ER-to-Relational mapping

Procedure to create a relational schema from an Entity-Relationship (ER)

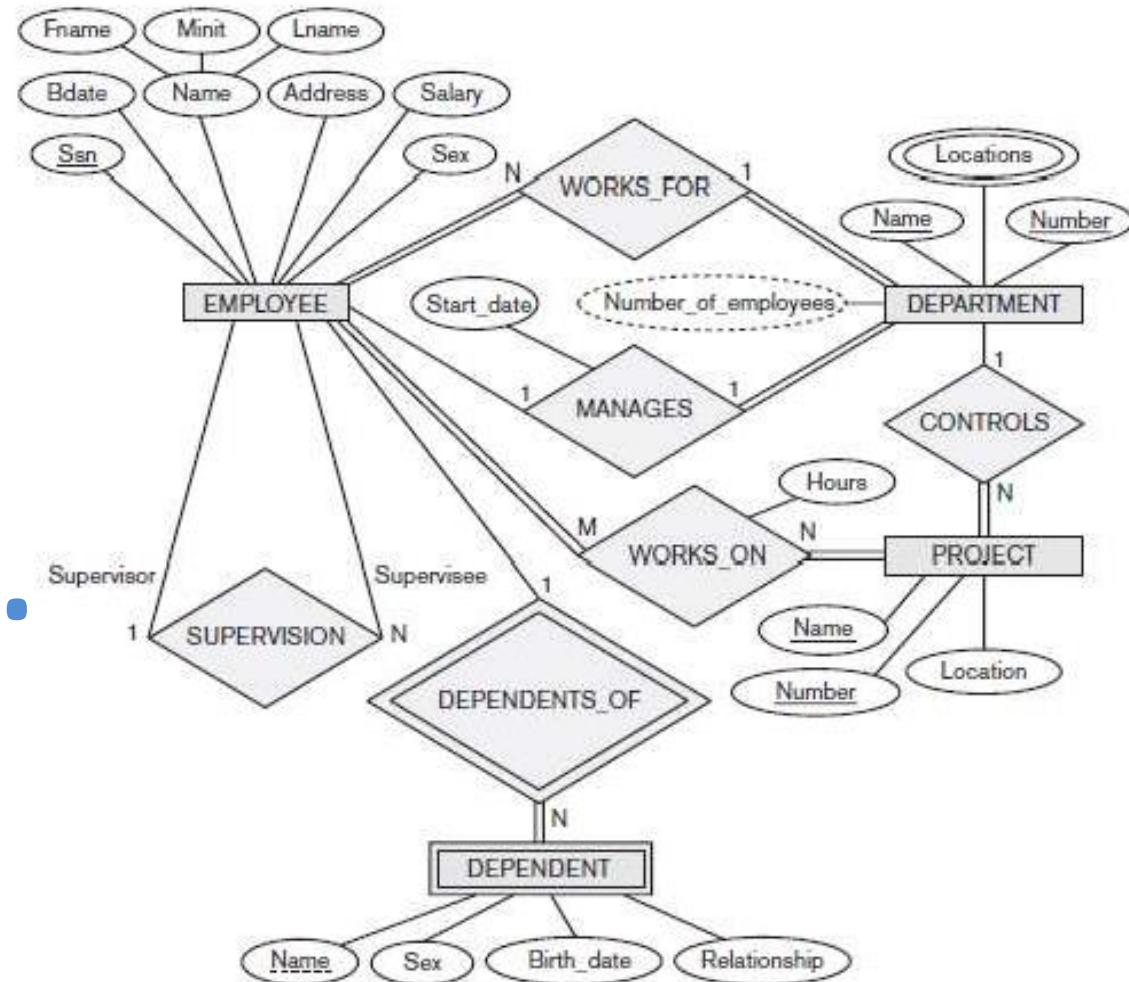


Fig 3.1: ER diagram of company database

Step 1: Mapping of Regular Entity Types

- For each regular entity type, create a relation R that includes all the simple attributes of E
- Include only the simple component attributes of a composite attribute
- Choose one of the key attributes of E as the primary key for R
- If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.

- If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R
- In our example-COMPANY database, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT
- we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively
- The relations that are created from the mapping of entity types are called **entity relations** because each tuple represents an entity instance.

EMPLOYEE

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

DEPARTMENT

Dname	<u>Dnumber</u>
-------	----------------

PROJECT

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

Step 2: Mapping of Weak Entity Types

- For each weak entity type, create a relation R and include all simple attributes of the entity type as attributes of R
- Include primary key attribute of owner as foreign key attributes of R
- In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT
- We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it as Essn
- The primary key of the DEPENDENT relation is the combination {Essn,Dependent_name}, because Dependent_name is the partial key of DEPENDENT
- It is common to choose the propagate (CASCADE) option for the referential triggered action on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity.
- This can be used for both ON UPDATE and ON DELETE.

DEPENDENT

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

Step 3: Mapping of Binary 1:1 Relationship Types

- For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R
- There are three possible approaches:
 - foreign key approach
 - merged relationship approach
 - crossreference or relationship relation approach

1. The foreign key approach

- Choose one of the relations— S , say—and include as a foreign key in S the primary key of T .
- It is better to choose an entity type with *total participation* in R in the role of S
- Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S .
- In our example, we map the 1:1 relationship type by choosing the participating entity type DEPARTMENT to serve in the role of S because its participation in the MANAGES relationship type is total
- We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it Mgr_ssn.
- We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date

2. Merged relation approach:

- merge the two entity types and the relationship into a single relation
- This is possible when *both participations are total*, as this would indicate that the two tables will have the exact same number of tuples at all times.

3. Cross-reference or relationship relation approach:

- set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.
- required for binary M:N relationships
- The relation R is called a relationship relation (or sometimes a lookup table), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple from T
- The relation R will include the primary key attributes of S and T as foreign keys to S and T .
- The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R .

- The drawback is having an extra relation, and requiring an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types

- For each regular binary 1:N relationship type R , identify the relation S that represents the participating entity type at the N -side of the relationship type.
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R
- Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S
- In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION
- For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno.
- For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn.
- The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation.

Step 5: Mapping of Binary M:N Relationship Types

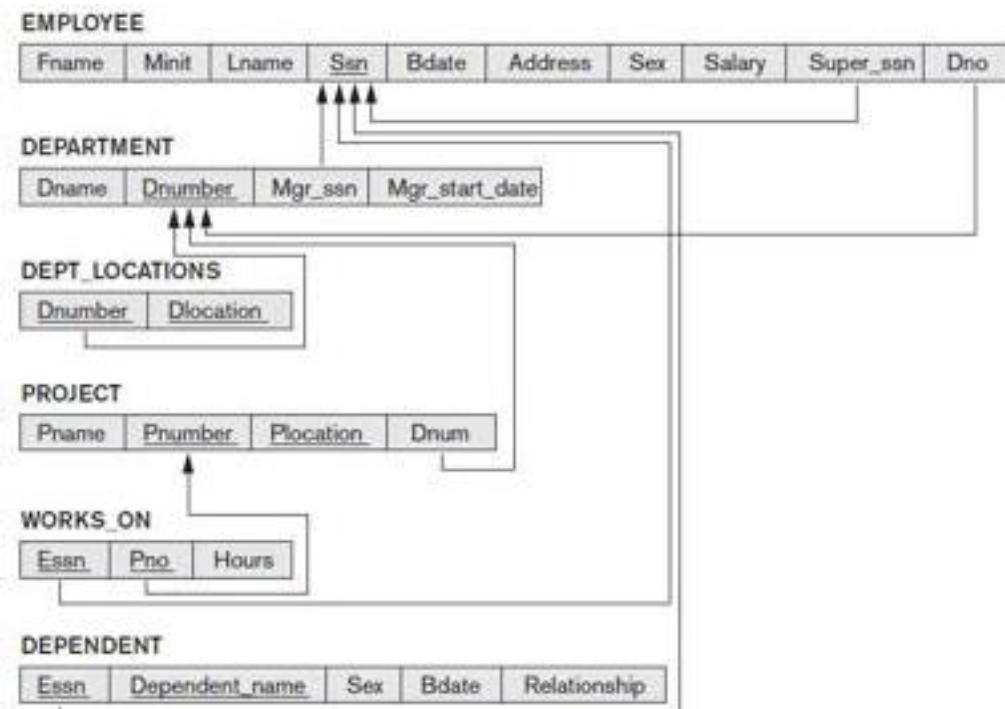
- For each binary M:N relationship type
 - Create a new relation S
 - Include primary key of participating entity types as foreign key attributes in S
 - Include any simple attributes of M:N relationship type
- In our example, we map the M:N relationship type WORKS_ON by creating the relation WORKS_ON. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively.
- We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type.
- The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}.

WORKS_ON		
Essn	Pno	Hours

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

Step 6: Mapping of Multivalued Attributes

- For each multivalued attribute
 - Create a new relation
 - Primary key of R is the combination of A and K
 - If the multivalued attribute is composite, include its simple components
- In our example, we create a relation DEPT_LOCATIONS
- The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, while Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation.
- The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}
- A separate tuple will exist in DEPT_LOCATIONS for each location that a department has
- The propagate (CASCADE) option for the referential triggered action should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE.



Step 7: Mapping of N -ary Relationship Types

- For each n -ary relationship type R
 - Create a new relation S to represent R
 - Include primary keys of participating entity types as foreign keys
 - Include any simple attributes as attributes
- The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types.
- For example, consider the relationship type SUPPLY. This can be mapped to the relation SUPPLY whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

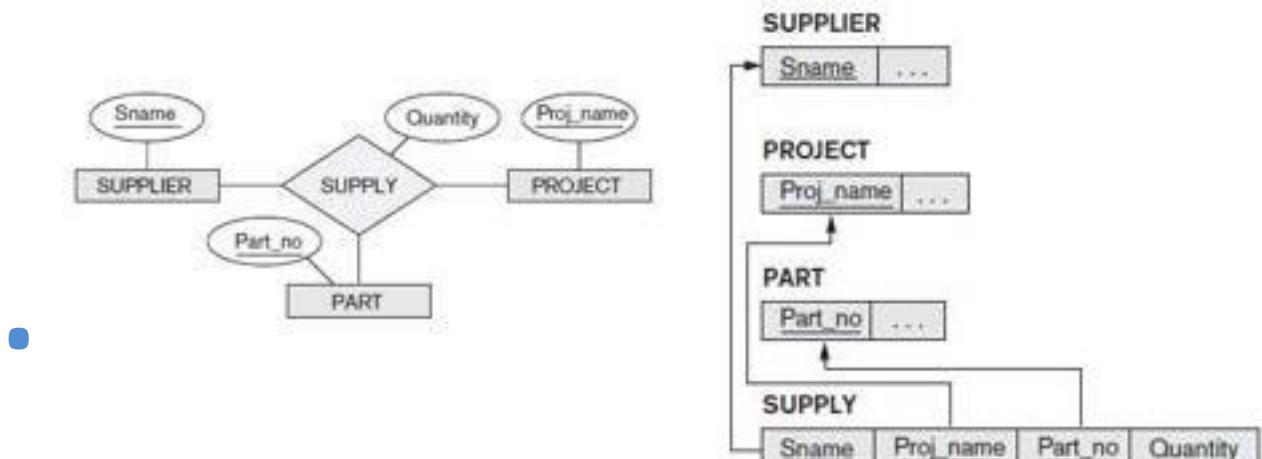


Figure 3.2: Mapping the n -ary relationship type SUPPLY

Chapter 4: SQL

4.1 Introduction

SQL was called SEQUEL (Structured English Query Language) and was designed and implemented at IBM Research. The SQL language may be considered one of the major reasons for the commercial success of relational databases. 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++.

4.2 SQL Data Definition and Data Types

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

4.2.1 Schema and Catalog Concepts in SQL

An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for *each element* in the schema. Schema elements include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement.

For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier ‘Jsmith’..

```
CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';
```

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

SQL uses the concept of a **catalog** – a named collection of schemas in an SQL environment. A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these

schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

4.2.2 The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

```
CREATE TABLE COMPANY.EMPLOYEE ...
```

rather than

```
CREATE TABLE EMPLOYEE ...
```

The relations declared through CREATE TABLE statements are called **base tables**.

Examples:

```
CREATE TABLE EMPLOYEE
( Fname          VARCHAR(15)      NOT NULL,
  Minit          CHAR,
  Lname          VARCHAR(15)      NOT NULL,
  Ssn            CHAR(9)         NOT NULL,
  Bdate          DATE,
  Address        VARCHAR(30),
  Sex            CHAR,
  Salary          DECIMAL(10,2),
  Super_ssn     CHAR(9),
  Dno            INT             NOT NULL,
  PRIMARY KEY (Ssn),
  FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
  FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
```

CREATE TABLE DEPARTMENT

(Dname	VARCHAR(15)	NOT NULL,
Dnumber	INT	NOT NULL,
Mgr_ssn	CHAR(9)	NOT NULL,
Mgr_start_date	DATE,	
PRIMARY KEY (Dnumber),		
UNIQUE (Dname),		
FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn));		

CREATE TABLE DEPT_LOCATIONS

(Dnumber	INT	NOT NULL,
Dlocation	VARCHAR(15)	NOT NULL,
PRIMARY KEY (Dnumber, Dlocation),		
FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber));		

CREATE TABLE PROJECT

(Pname	VARCHAR(15)	NOT NULL,
Pnumber	INT	NOT NULL,
Plocation	VARCHAR(15),	
Dnum	INT	NOT NULL,
PRIMARY KEY (Pnumber),		
UNIQUE (Pname),		
FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber));		

CREATE TABLE WORKS_ON

(Essn	CHAR(9)	NOT NULL,
Pno	INT	NOT NULL,
Hours	DECIMAL(3,1)	NOT NULL,
PRIMARY KEY (Essn, Pno),		
FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),		
FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber));		

CREATE TABLE DEPENDENT

(Essn	CHAR(9)	NOT NULL,
Dependent_name	VARCHAR(15)	NOT NULL,
Sex	CHAR,	
Bdate	DATE,	
Relationship	VARCHAR(8),	
PRIMARY KEY (Essn, Dependent_name),		
FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn));		

4.2.3 Attribute Data Types and Domains in SQL

Basic data types

1. Numeric data types includes

- integer numbers of various sizes (INTEGER or INT, and SMALLINT)
- floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION).
- Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where
 - i - precision, total number of decimal digits
 - j - scale, number of digits after the decimal point

2. Character-string data types

- fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters
- varying length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters
- When specifying a literal string value, it is placed between single quotation marks (‘ ’), and it is *case sensitive*
- For fixed length strings, a shorter string is padded with blank characters to the right
- For example, if the value ‘Smith’ is for an attribute of type CHAR(10), it is padded with five blank characters to become ‘Smith’
- Padded blanks are generally ignored when strings are compared
- Another variable-length string data type called ~~CHARACTER~~ ~~LARGE OBJECT~~ or CLOB is also available to specify columns that have large text values, such as documents
- The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G)
- For example, CLOB(20M) specifies a maximum length of 20 megabytes.

3. Bit-string data types are either of

- fixed length n —BIT(n)—or varying length—BIT VARYING(n), where n is the maximum number of bits.
- The default for n , the length of a character string or bit string, is 1.

- Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'
 - Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images.
 - The maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G)
 - For example, BLOB(30G) specifies a maximum length of 30 gigabits.
4. A Boolean data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN
5. The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD
6. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.
Only valid dates and times should be allowed by the SQL implementation.
7. TIME WITH TIME ZONE data type includes an additional six positions for specifying the displacement zone, which is in the range +13:00 to -12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

Additional data types

1. Timestamp data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier.
2. INTERVAL data type. This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

It is possible to specify the data type of each attribute directly or a domain can be declared, and the domain name used with the attribute Specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

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

We can use SSN_TYPE in place of CHAR(9) for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT

4.3 Specifying Constraints in SQL

Basic constraints that can be specified in SQL as part of table creation:

- key and referential integrity constraints
- Restrictions on attribute domains and NULLs
- constraints on individual tuples within a relation

4.3.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause **DEFAULT <value>** to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute.

```
CREATE TABLE DEPARTMENT
(
    ...
    Mgr_ssn CHAR(9) NOT NULL DEFAULT '888665555',
    -----
    -----
)

```

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition . For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

```
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER
```

CHECK (D_NUM > 0 AND D_NUM < 21);

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department number such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

4.3.2 Specifying Key and Referential Integrity Constraints

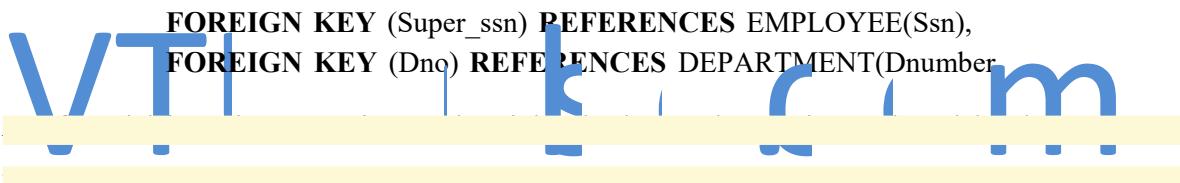
The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly. For example, the primary key of DEPARTMENT can be specified as:

Dnumber INT **PRIMARY KEY**;

The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

Dname VARCHAR(15) **UNIQUE**;

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



integrity violation is to **reject** the update operation that will cause a violation, which is known as the **RESTRICT** option.

The schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint. The options include **SET NULL**, **CASCADE**, and **SET DEFAULT**. An option must be qualified with either **ON DELETE** or **ON UPDATE**

- **FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber) ON DELETE SET DEFAULT ON UPDATE CASCADE**
- **FOREIGN KEY (Super_ssN) REFERENCES EMPLOYEE(SsN) ON DELETE SET NULL ON UPDATE CASCADE**
- **FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) ON DELETE CASCADE ON UPDATE CASCADE**

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT.

The action for CASCADE ON DELETE is to delete all the referencing tuples whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples . It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for “relationship” relations such as WORKS_ON; for relations that represent multivalued attributes, such as DEPT_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

4.3.3 Giving Names to Constraints

The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint.

4.3.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple individually and are checked whenever a tuple is inserted or modified

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

```
CHECK (Dept_create_date <= Mgr_start_date);
```

4.4 Basic Retrieval Queries in SQL

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

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

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

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>;
```

Where,

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

Examples:

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’;
```

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which a . The WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition**.

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;
```

In the WHERE clause, the condition Dname = ‘Research’ is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE. A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join query**.

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

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Plocation='Stafford';
```

The join condition `Dnum = Dnumber` relates a project tuple to its controlling department tuple, whereas the join condition `Mgr_ssn = Ssn` relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a *combination* of one project, one department, and one employee that satisfies the join conditions. The projection attributes are used to choose the attributes to be displayed from each combined tuple.

4.4.2 Ambiguous Attribute Names, Aliasing, Renaming, 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 multitable query refers to two or more attributes with the same name, we must **qualify** the attribute name with the relation name to prevent ambiguity.

period.

Example: Retrieve the name and address of all employees who work for the ‘Research’ department

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

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice. For example consider the query: For each employee, retrieve the employee’s first and last name and the first and last name of his or her immediate supervisor.

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

In this case, we are required to declare alternative relation names E and S, called **aliases or tuple variables**, for the EMPLOYEE relation. An alias can follow the keyword **AS**, or it can directly follow the relation name for example, by writing `EMPLOYEE E`, `EMPLOYEE S`. It is also possible to **rename** the relation attributes within the query in SQL by giving them aliases. For example, if we write

```
EMPLOYEE AS E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)
```

in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on

4.4.3 Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT – all possible tuple combinations – of these relations is selected.

Example: Select all EMPLOYEE Ssns and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname in the database.

```
SELECT Ssn
FROM EMPLOYEE;
SELECT Ssn, Dname
FROM EMPLOYEE, DEPARTMENT;
```

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, the following query retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5

- **SELECT * FROM EMPLOYEE WHERE Dno=5;**
- SELECT * FROM EMPLOYEE, DEPARTMENT WHERE Dname='Research'**
- AND Dno=Dnumber;**
- SELECT * FROM EMPLOYEE, DEPARTMENT;**

4.4.4 Tables as Sets in SQL

SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

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

Example : Retrieve the salary of every employee and all distinct salary values

- (a) **SELECT ALL Salary FROM EMPLOYEE;**
- (b) **SELECT DISTINCT Salary FROM EMPLOYEE;**

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra. There are

- set union (**UNION**)
- set difference (**EXCEPT**) and
- set intersection (**INTERSECT**)

The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. These set operations apply only to union-compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.

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

```
(SELECT DISTINCT Pnumber FROM PROJECT, DEPARTMENT,
EMPLOYEE WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='Smith' )
UNION
( SELECT DISTINCT Pnumber FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno AND Essn=Ssn AND Lname='Smith' );
```

4.4.5 Substring Pattern Matching and Arithmetic Operators

Several more features of SQL

The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This can be used for string **pattern matching**. Partial strings are specified using two reserved characters:

- % replaces an arbitrary number of zero or more characters
- _ (underscore) replaces a single character

For example, consider the following query: Retrieve all employees whose address is in Houston, Texas

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Address
LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1950s, we can use Query

```
SELECT Fname, Lname FROM EMPLOYEE
WHERE Bdate LIKE '____5_____';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD%EF' ESCAPE '\' represents the literal string 'AB_CD%EF' because \ is specified as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (' ') if they are to be included in a string because they are used to begin and end strings. If an apostrophe ('') is needed, it is represented as two consecutive apostrophes ("") so that it will not be interpreted as ending the string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue the following query:

```
SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE E.Ssn=W.Essn AND W.Pno=P.Pnumber AND P.Pname='ProductX';
```

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

```
SELECT * FROM EMPLOYEE WHERE (Salary BETWEEN 30000 AND
40000) AND Dno = 5;
```

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

4.4. S

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

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

```
SELECT D.Dname, E.Lname, E.Fname, P.Pname
FROM DEPARTMENT D, EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE D.Dnumber= E.Dno AND E.Ssn= W.Essn AND W.Pno= P.Pnumber
ORDER BY D.Dname, E.Lname, E.Fname;
```

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause can be written as

ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC

4.5 INSERT, DELETE, and UPDATE Statements in SQL

4.5.1 The INSERT Command

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.

Example: **INSERT INTO EMPLOYEE VALUES (‘Richard’, ‘K’, ‘Marini’, ‘653298653’, ‘1962-12-30’, ‘98 Oak Forest, Katy, TX’, ‘M’, 37000, ‘653298653’, 4);**

INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn)

VALUES (‘Richard’, ‘Marini’, 4, ‘653298653’);

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

A variation of the **INSERT** command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

U3A: CREATE TABLE WORKS_ON_INFO(

```
    Emp_name VARCHAR(15),
    Proj_name VARCHAR(15),
    Hours_per_week DECIMAL(3,1) );
```

U3B: INSERT INTO WORKS_ON_INFO

```
( Emp_name, Proj_name, Hours_per_week )
SELECT E.Lname, P.Pname, W.Hours
FROM PROJECT P, WORKS_ON W, EMPLOYEE E
WHERE P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B. We can now query WORKS_ON_INFO as we would any other relation;

4.5.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. The deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL.

Example:

```
DELETE FROM EMPLOYEE WHERE Lname='Brown';
```

Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.

4.5.3 The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected Tuples. 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

```
UPDATE PROJECT SET Plocation = 'Bellaire', Dnum = 5 WHERE Pnumber=10;
```

As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL.

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the ‘Research’ department a 10 percent raise in salary, as shown by the following query

```
UPDATE EMPLOYEE  
SET Salary = Salary * 1.1  
WHERE Dno = 5;
```

Each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

4.6 Additional Features of SQL

- SQL has various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases.
- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.
- SQL has language constructs for specifying the *granting and revoking of privileges* to users.
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**.
- SQL and relational databases can interact with new technologies such as XML

Question Bank

1. Define the following terms as they apply to the relational model of data:
 - i) domain ii) attribute iii) n-tuple iv) relation schema v) relation state
 - vi) degree of a relation vii) relational database schema viii) relational database state.
2. What is the difference between a key and a superkey?
3. Discuss the various reasons that lead to the occurrence of NULL values in relations.
4. Discuss the characteristics of relations
5. Discuss the various restrictions on data that can be specified on a relational database in the form of constraints.
6. Suppose that each of the following Update operations is applied directly to the company database state. Discuss all integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.
 - a. Insert <‘Robert’, ‘Scott’, ‘943775543’, ‘1972-F’, 21, ‘2365 Newcastle -06- Rd,Bellaire, TX’, M, 58000, ‘888665555’, 1> into EMPLOYEE.
 - b. Insert <‘ProductA’, 4, ‘Bellaire’, 2> into PROJECT.
 - c. Insert <‘Production’, 4, ‘943775543’, ‘2007-10-01’> into DEPARTMENT.
 - d. Insert <‘677678989’, NULL, ‘40.0’> into WORKS_ON.
 - e. Insert <‘453453453’, ‘John’, ‘M’, ‘1990-12-12’, ‘spouse’> into DEPENDENT.
 - f. Delete the WORKS_ON tuples with Essn = ‘333445555’.
 - g. Delete the EMPLOYEE tuple with Ssn = ‘987654321’.
 - h. Delete the PROJECT tuple with Pname = ‘ProductX’.
 - i. Modify the Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to ‘123456789’ and ‘2007-10-01’, respectively.
 - j. Modify the Super_ssn attribute of the EMPLOYEE tuple with Ssn = ‘999887777’ to ‘943775543’.
 - k. Modify the Hours attribute of the WORKS_ON tuple with Essn = ‘999887777’ and Pno = 10 to ‘5.0’.
7. Explain the following unary operations with syntax and example
 - i) SELECT ii) PROJECT iii) RENAME

8. Explain the following binary operations with syntax and example
 - i) UNION ii) INTERSECTION iii) MINUS iv) CROSS PRODUCT V) DIVISION
9. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?
10. Discuss the various types of *join* operations.
11. Discuss the notation used in relational systems to represent queries internally.
12. Illustrate with an example, significance of generalized projection.
13. Illustrate with an example, Aggregate Functions and Grouping
14. Illustrate with an example, Recursive Closure Operations
15. How are the OUTER JOIN operations different from the INNER JOIN operations?
16. How is the OUTER UNION operation different from UNION?
17. Specify the following queries on the COMPANY relational database schema using the relational operators
 - a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.
 - b. List the names of all employees who have a dependent with the same first name as themselves.
 - c. Find the names of all employees who are directly supervised by 'Franklin Wong'.
 - d. For each project, list the project name and the total hours per week (by all employees) spent on that project.
 - e. Retrieve the names of all employees who do not work on any project.
 - f. Retrieve the average salary of all female employees.
18. Discuss the correspondences between the ER model constructs and the relational model constructs. Show how each ER model construct can be mapped to the relational model
19. Discuss the data types that are allowed for SQL attributes
20. Write SQL update statements to do the following on the database schema shown in Figure(1)
 - a. Insert a new student, <'Johnson', 25, 1, 'Math'>, in the database.
 - b. Change the class of student 'Smith' to 2.
 - c. Insert a new course, <'Knowledge Engineering', 'CS4390', 3, 'CS'>.
 - d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

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

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

Fig (2):student scheme and database state

21. Briefly discuss how the different update operations on a relation deal with constraint violations?

22. Consider the following schema for a COMPANY database:

EMPLOYEE (Fname, Lname, Ssn, Address, Super-ssn, Salary, Dno)

DEPARTMENT (Dname, Dnumber, Mgr-ssn, Mgr-start-date) DEPT-

LOCATIONS (Dnumber, Dlocation)

PROJECT (Pname, Pnumber, Plocation, Dnum)

WORKS-ON (Ess!!, Pno, Hours)

DEPENDENT (Essn, Dependent-name, Sex, Bdate, Relationship)

Write the queries in relational algebra.

- i) Retrieve the name and address of all employees who work for' Sales' department.
- ii) Find the names of employees who work on all the projects controlled by the department number 3.
- iii) List the names of all employees with two or more dependents.
- iv) Retrieve the names of employees who have no dependents.



Module 3

Chapter 1: SQL- Advances Queries

1.1 More Complex SQL Retrieval Queries

Additional features allow users to specify more complex retrievals from database

1.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value

Example

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute CollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

NULL is considered to be different from every other NULL value in the various database records.

be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used

Table 5.1 Logical Connectives in Three-Valued Logic

		TRUE	FALSE	UNKNOWN
		TRUE	FALSE	UNKNOWN
(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

The rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.

SQL allows queries that check whether an attribute value is NULL using the comparison operators **IS** or **IS NOT**.

Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

1.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**

Example1: List the project numbers of projects that have an employee with last name ‘Smith’ as manager

```
SELECT DISTINCT Pnumber FROM PROJECT WHERE
Pnumber IN
(SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='smith');
```

Example2: List the project numbers of projects that have an employee with last name ‘Smith’ as either manager or as worker.

```
SELECT DISTINCT Pnumber FROM PROJECT WHERE
Pnumber IN
(SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='smith')
OR
Pnumber IN
(SELECT Pno FROM WORKS_ON, EMPLOYEE WHERE Essn=Ssn AND
Lname='smith');
```

We make use of comparison operator **IN**, which compares a value v with a set (or multiset) of values V and evaluates to **TRUE** if v is one of the elements in V .

The first nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as manager. The second nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. For example, the following query will select the Essns of all employees who work the same (project, hours) combination on some project that employee ‘John Smith’ (whose Ssn = ‘123456789’) works on

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN ( SELECT      Pno, Hours
                           FROM        WORKS_ON
                           WHERE       Essn='123456789' );
```

In this example, the IN operator compares the subtuple of values in parentheses (Pno,Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

Nested Queries::Comparison Operators

Other comparison operators can be used to compare a single value v to a set or multiset V . The $=$ ANY (or $=$ SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. The keyword ALL can also be combined with each of these operators. For example, the comparison condition ($v > \text{ALL } V$) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V . For example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL ( SELECT Salary
FROM EMPLOYEE
WHERE Dno=5 );
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query*, and another in a relation in the FROM clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query**.

To avoid potential errors and ambiguities, create tuple variables (aliases) for all tables referenced in SQL query

Example: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn
FROM DEPENDENT AS D
WHERE E.Fname=D.Dependent_name
AND E.Sex=D.Sex );
```

In the above nested query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex.

1.1.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**.

Example:

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn
FROM DEPENDENT AS D
WHERE E.Fname=D.Dependent_name
AND E.Sex=D.Sex );
```

The nested query is evaluated once for each tuple (or combination of tuples) in the outer query. we can think of query in above example as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

1.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS Functions

The EXISTS function in SQL is used to check whether the result of a correlated nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value

- **TRUE** if the nested query result contains at least one tuple, or
- **FALSE** if the nested query result contains no tuples.

For example, the query to retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee can be written using EXISTS functions as follows:

```
SELECT E.Fname, E.Lname
```

```
FROM EMPLOYEE AS E
WHERE EXISTS ( SELECT *
FROM DEPENDENT AS D
WHERE E.Ssn=D.Essn AND E.Sex=D.Sex
AND E.Fname=D.Dependent_name);
```

Example: List the names of managers who have at least one dependent

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE EXISTS ( SELECT *
FROM DEPENDENT
WHERE Ssn=Essn )
AND
EXISTS ( SELECT *
FROM DEPARTMENT
WHERE Ssn=Mgr_ssn );
```

In general, EXISTS(Q) returns **TRUE** if there is at least one tuple in the result of the nested query Q, and it returns **FALSE** otherwise.



NOT EXISTS Functions

NOT EXISTS(Q) returns **TRUE** if there are no tuples in the result of nested query Q, and it returns **FALSE** otherwise.

Example: Retrieve the names of employees who have no dependents.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS ( SELECT *
FROM DEPENDENT
WHERE Ssn=Essn );
```

For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Example: Retrieve the name of each employee who works on all the projects controlled by department number 5

```
SELECT Fname, Lname
```

```

FROM EMPLOYEE
WHERE NOT EXISTS ( ( SELECT Pnumber
FROM PROJECT
WHERE Dnum=5)
EXCEPT ( SELECT Pno
FROM WORKS_ON
WHERE Ssn=Essn );

```

UNIQUE Functions

UNIQUE(Q) returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

1.1.5 Explicit Sets and Renaming of Attributes in SQL

In SQL it is possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses.

Example: Retrieve the Social Security numbers of all employees who work on project numbers 1, 2,

or 3. ●

```

SELECT DISTINCT Essn
FROM WORKS_ON
WHERE Pno IN (1, 2, 3);

```

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name

Example: Retrieve the last name of each employee and his or her supervisor

```

SELECT E.Lname AS Employee_name,
S.Lname AS Supervisor_name
FROM EMPLOYEE AS E,
EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;

```

1.1.6 Joined Tables in SQL and Outer Joins

An SQL join clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each. SQL specifies four types of JOIN

1. INNER,
2. OUTER
3. EQUIJOIN and
4. NATURAL JOIN

INNER JOIN

An inner join is the most common join operation used in applications and can be regarded as the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join- predicate (the condition). The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all records in the tables (combining every record in table A with every record in table B) then return all records which satisfy the join predicate

Example: `SELECT * FROM employee`

- `INNER JOIN department ON`
`employee.dno = department.dnumber;`

EQUIJOIN and NATURAL JOIN

An **EQUIJOIN** is a specific type of comparator-based join that uses only equality comparisons in the join-predicate. Using other comparison operators (such as `<`) disqualifies a join as an equijoin.

NATURAL JOIN is a type of EQUIJOIN where the join predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns.

SELECT	Fname, Lname, Address
FROM	EMPLOYEE NATURAL JOIN
	DEPARTMENT
WHERE	Dname='Research';

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause.

CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

OUTER JOIN

An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into

- Left outer joins
- Right outer joins
- Full outer joins

No implicit join-notation for outer joins exists in standard SQL.

▶ LEFT OUTER JOIN

- ▶ Every tuple in left table must appear in result
- ▶ If no matching tuple
 - Padded with NULL values for attributes of right table

Query Retrieve the names of employees and their supervisors

Q8A: **SELECT** E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name
FROM EMPLOYEE **AS** E, EMPLOYEE **AS** S
WHERE E.Super_ssn=S.Ssn;

Implicit inner join

only employees who have a supervisor are included in the result; an EMPLOYEE tuple whose value for Super_ssn is NULL is excluded.

Q8B: **SELECT** E.Lname **AS** Employee_name,
S.Lname **AS** Supervisor_name
FROM EMPLOYEE **AS** E **LEFT OUTER JOIN** EMPLOYEE **AS** S
ON E.Super_ssn=S.Ssn);

If the user requires that all employees be included, an OUTER JOIN must be used explicitly

▶ **RIGHT OUTER JOIN**

- ▶ Every tuple in right table must appear in result
- ▶ If no matching tuple
 - Padded with NULL values for the attributes of left table

▶ **FULL OUTER JOIN**

- ▶ a full outer join combines the effect of applying both left and right outer joins.
 - ▶ Where records in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row.
 - ▶ For those records that do match, a single row will be produced in the result set (containing fields populated from both tables).
-
- ▶ Not all SQL implementations have implemented the new syntax of joined tables.
 - ▶ In some systems, a different syntax was used to specify outer joins by using the comparison operators `+ =`, `= +`, and `+ = +` for left, right, and full outer join, respectively
 - ▶ For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

Q8C: **SELECT** E.Lname, S.Lname
 FROM EMPLOYEE E, EMPLOYEE S
 WHERE E.Super_ssn `+ =` S.Ssn;

MULTIWAY JOIN

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**.

Example: For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum=Dnumber)
JOIN EMPLOYEE ON Mgr_ssn=Ssn)
WHERE Plocation=‘Stafford’;
```



1.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. A number of built-in aggregate 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 can be applied to a set or 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.

Examples

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

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

- 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.

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
WHERE Dname=‘Research’;
```

- Count the number of distinct salary values in the database.

```
SELECT COUNT (DISTINCT Salary)
FROM EMPLOYEE;
```

4. To retrieve the names of all employees who have two or more dependents

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE ( SELECT COUNT (*)
FROM DEPENDENT
WHERE Ssn=Essn ) >= 2;
```

1.1.8 Grouping: The GROUP BY and HAVING Clauses

Grouping is used to create subgroups of tuples before summarization. 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 non overlapping 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)**.

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).

Example: For each department, retrieve the department number, the number of employees in the department, and their average salary.

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

The diagram illustrates the grouping process. It shows a large arrow pointing from the original EMPLOYEE table to a smaller result table. The result table has three rows corresponding to Dno values 5, 4, and 1. Each row contains the count of employees (4, 3, 1) and the average salary (33250, 31000, 55000). The result table is labeled "Result of Q24".

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789	...	30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelby	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

Grouping EMPLOYEE tuples by the value of Dno

If NULLs exist in the grouping attribute, then 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 query

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

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

Above query 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.

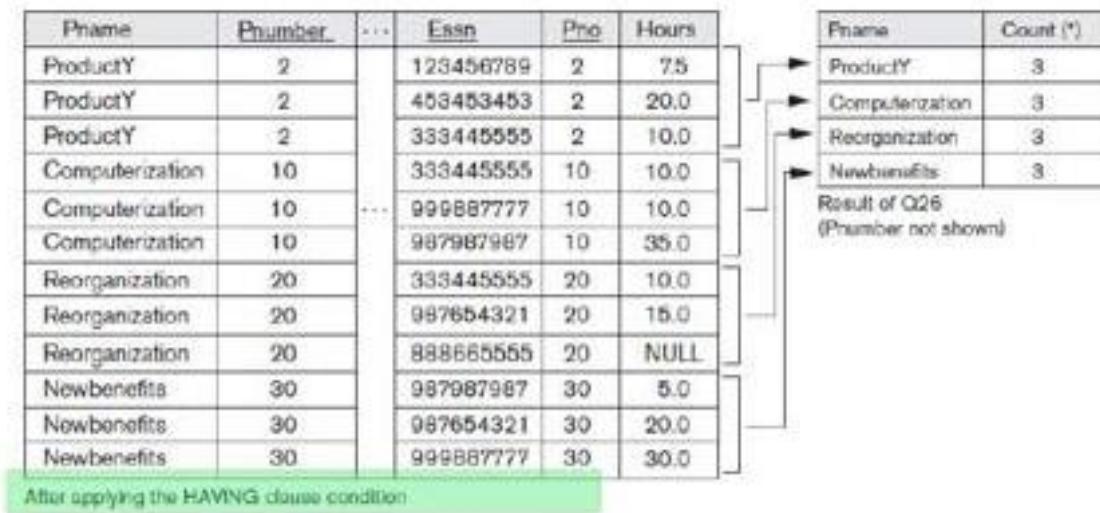
HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

Example: 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.

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

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	Essn	Pno	Hours
ProductX	1	123456789	1	32.5
ProductX	1	453453453	1	20.0
ProductY	2	123456789	2	7.5
ProductY	2	453453453	2	20.0
ProductY	2	333445555	2	10.0
ProductZ	3	666884444	3	40.0
ProductZ	3	333445555	3	10.0
Computerization	10	333445555	10	10.0
Computerization	10	999887777	10	10.0
Computerization	10	987987987	10	35.0
Reorganization	20	333445555	20	10.0
Reorganization	20	987654321	20	15.0
Reorganization	20	888665555	20	NULL
Newbenefits	30	987987987	30	5.0
Newbenefits	30	987654321	30	20.0
Newbenefits	30	999887777	30	30.0



Example: For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
SELECT Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno AND Ssn=Essn AND Dno=5
GROUP BY Pnumber, Pname;
```

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

```
SELECT Dnumber, COUNT (*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno AND Salary>40000 AND
(SELECT Dno
FROM EMPLOYEE
GROUP BY Dno
HAVING COUNT (*) > 5);
```

1.1.9 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

The **SELECT** clause lists the attributes or functions to be retrieved. The **FROM** clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The **WHERE** clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. **GROUP BY** specifies grouping attributes, whereas **HAVING** specifies a condition on the groups being selected rather than on the individual tuples. Finally, **ORDER BY** specifies an order for displaying the result of a query.

A query is evaluated conceptually by first applying the **FROM** clause to identify all tables involved in the query or to materialize any joined tables followed by the **WHERE** clause to select and join tuples, and then by **GROUP BY** and **HAVING**. **ORDER BY** is applied at the end to sort the query result. Each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute



In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages.

- The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the **WHERE** clause, or by using joined relations in the **FROM** clause, or with some form of nested queries and the **IN** comparison. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.
- The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way

1.2 Specifying Constraints as Assertions and Actions as Triggers

1.2.1 Specifying General Constraints as Assertions in SQL

Assertions are used to specify additional types of constraints outside scope of built-in relational model constraints. In SQL, users can specify general constraints via declarative assertions, using the **CREATE ASSERTION** statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

General form :

```
CREATE ASSERTION <Name_of_assertion> CHECK (<cond>)
```

For the assertion to be satisfied, the condition specified after **CHECK** clause must return true.

For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT * FROM EMPLOYEE E, EMPLOYEE M,
DEPARTMENT D WHERE E.Salary>M.Salary AND
E.Dno=D.Dnumber AND D.Mgr_ssn=M.Ssn ) );
```

The constraint name **SALARY_CONSTRAINT** is followed by the keyword **CHECK**, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. Any WHERE clause condition can be used, but many constraints can be specified using the **EXISTS** and **NOT EXISTS** style of SQL conditions.

By including this query inside a **NOT EXISTS** clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty

Example: consider the bank database with the following tables

- *branch (branch_name, branch_city, assets)*
- *customer (customer_name, customer_street, customer_city)*
- *account (account_number, branch_name, balance)*
- *loan (loan_number, branch_name, amount)*
- *depositor (customer_name, account_number)*
- *borrower (customer_name, loan_number)*

1. Write an assertion to specify the constraint that the Sum of loans taken by a customer does not exceed 100,000

```
CREATE ASSERTION sumofloans
CHECK (100000>= ALL
SELECT customer_name,sum(amount)
FROM borrower b, loan l
WHERE b.loan_number=l.loan_number
GROUP BY customer_name );
```

2. Write an assertion to specify the constraint that the Number of accounts for each customer in a given branch is at most two

```
CREATE ASSERTION NumAccounts
CHECK ( 2 >= ALL
SELECT customer_name,branch_name, count(*)
FROM account A , depositor D
WHERE A.account_number = D.account_number
GROUP BY customer_name, branch_name );
```

1.2.2 Introduction to Triggers in SQL

cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. The CREATE TRIGGER statement is used to implement such actions in SQL.

General form:

```
CREATE TRIGGER <name>
BEFORE | AFTER | <events>
FOR EACH ROW |FOR EACH STATEMENT
WHEN (<condition>)
<action>
```

A trigger has three components

1. **Event:** When this event happens, the trigger is activated

- Three event types : Insert, Update, Delete
- Two triggering times: Before the event

After the event

2. Condition (optional): If the condition is true, the trigger executes, otherwise skipped

3. Action: The actions performed by the trigger

When the **Event** occurs and **Condition** is true, execute the **Action**

Create Trigger ABC

Before Insert On
Students

Create Trigger XYZ

After Update On Students

This trigger is activated when an insert statement is issued, but before the new record is inserted

This trigger is activated when an update statement is issued and after the update is executed

Does the trigger execute for each updated or deleted record, or once for the entire statement ?. We define such granularity as follows:

Create Trigger <name>

Before| After Insert| Update| Delete

This is the event

For Each Row | For Each Statement

This is the granularity

Create Trigger XYZ

After Update ON <tablename>

For each statement

....

This trigger is activated once (per UPDATE statement) after all records are updated

Create Trigger XYZ

Before Delete ON <tablename>

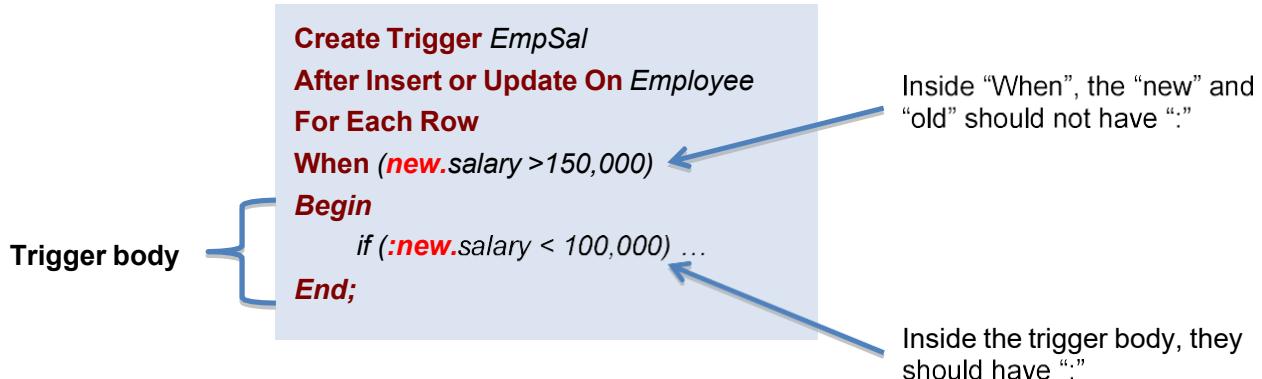
For each row

....

This trigger is activated before deleting each record

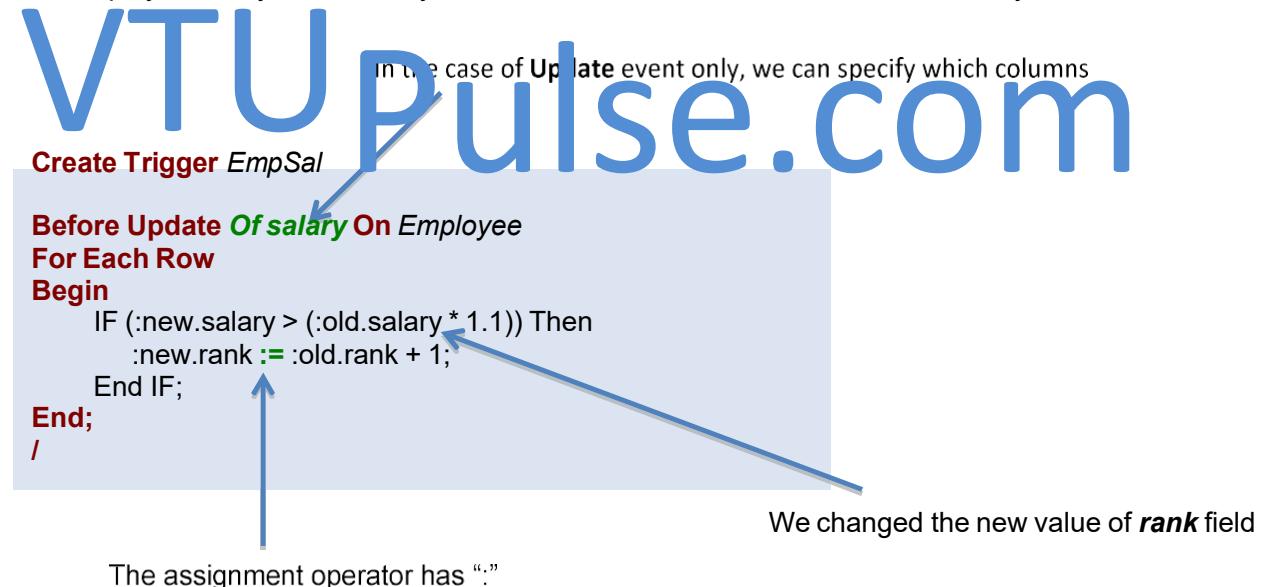
In the action, you may want to reference:

- The new values of inserted or updated records (`:new`)
- The old values of deleted or updated records (`:old`)

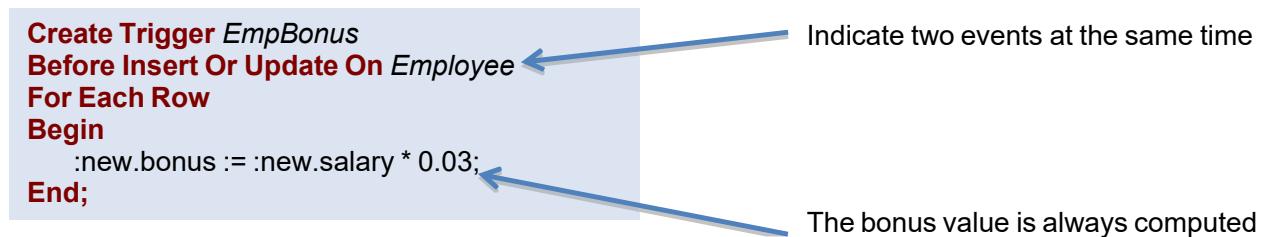


Examples:

- If the employee salary increased by more than 10%, then increment the rank field by 1.



- Keep the bonus attribute in Employee table always 3% of the salary attribute



3. Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database

- Several events can trigger this rule:
 - inserting a new employee record
 - changing an employee's salary or
 - changing an employee's supervisor
- Suppose that the action to take would be to call an external stored procedure **SALARY_VIOLATION** which will notify the supervisor

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR(NEW.Supervisor_ssN,NEW.Ssn );
```

- The trigger is given the name **SALARY_VIOLATION**, which can be used to remove or deactivate the trigger later
- In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor
- The action is to execute the stored procedure **INFORM_SUPERVISOR**

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates.

Assertions vs. Triggers

- Assertions ~~can only check the data~~, they only check certain conditions. Triggers are more powerful because they can check conditions and also modify the data
- Assertions are not linked to specific tables in the database and not linked to specific events. Triggers are linked to specific tables and specific events
- All assertions can be implemented as triggers (one or more). Not all triggers can be implemented as assertions

Example: Trigger vs. Assertion

All new customers opening an account must have opening balance $\geq \$100$. However, once the account is opened their balance can fall below that amount.



We need triggers, assertions cannot be used



Trigger Event: Before Insert

```
Create Trigger OpeningBal
Before Insert On Customer
For Each Row
Begin
    IF (:new.balance is null or :new.balance < 100) Then
        RAISE_APPLICATION_ERROR(-20004, 'Balance should be >= $100');
    End If;
End;
```

1.3 Views (Virtual Tables) in SQL

1.3.1 Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations 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, referring to the COMPANY database, 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 three tables EMPLOYEE,WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the 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.

1.3.2 Specification of Views in SQL

In SQL, 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 new 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.

Example 1:

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

Example 2:

```
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 Dname;
```

In example 1, we did not specify any new attribute names for the view WORKS_ON1. In this case, WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

Example 2 explicitly specifies new attribute 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.

WORKS_ON1			
Fname	Lname	Pname	Hours

DEPT_INFO		
Dept_name	No_of_emps	Total_sal

We can now specify SQL queries on a view or virtual table 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 the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as :

```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname='ProductX';
```

The same query would require the specification of two joins if specified on the base relations directly. one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism.

A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date.

If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it. For example : **DROP VIEW WORKS_ON1;**

1.3.3 View Implementation, View Update and Inline Views

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested.

- One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. For example, the query

```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname='ProductX';
```

would be automatically modified to the following query by the DBMS:

```
SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn AND Pno=Pnumber
AND Pname='ProductX';
```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are going to be applied to the same view within a short period of time.

- The second strategy, called **view materialization**, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date.

Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a materialized view table when a database update is applied to one of the defining base tables.

The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways. Hence, it is often not possible for the DBMS to determine which of the updates is intended.

To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1: UPDATEWORKS_ON1
      SET Pname = 'ProductY'
      WHERE Lname='Smith' AND Fname='John'
            AND Pname='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create additional side effects that affect the result of other queries.

For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a) : UPDATEWORKS_ON
      SET Pno= (SELECT Pnumber
                FROM PROJECT
                WHERE Pname='ProductY' )
      WHERE Essn IN ( SELECT Ssn
                      FROM EMPLOYEE
                      WHERE Lname='Smith' AND Fname='John' )
            AND
            Pno= (SELECT Pnumber
                  FROM PROJECT
                  WHERE Pname='ProductX' );
```

WHERE

(b) : **UPDATE** PROJECT **SET** Pname = 'ProductY'

WHERE Pname = 'ProductX';

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'.

It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This request is shown as UV2:

UV2: UPDATE DEPT_INFO

SET Total_sal=100000

WHERE Dname='Research';

A large number of updates on the underlying base relations can satisfy this view update.

Generally, a view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing one of the possible updates as the most likely one.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have default values specified*.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view *is to be updated*. This allows the system to check for view updatability and to plan an execution strategy for view updates. It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

1.4 Schema Change Statements in SQL

Schema evolution commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema.

1.4.1 The DROP Command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no longer needed, the **DROP SCHEMA** command can be used.

There are two drop behavior options: **CASCADE** and **RESTRICT**. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADING option is used as follows:

DROP SCHEMA COMPANY CASCADE;

If the **RESTRICT** option is chosen in place of **CASCADE**, the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed. To use the **RESTRICT** option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the **DROP TABLE** command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, , we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT CASCADE;

If the **RESTRICT** option is chosen instead of **CASCADE**, a table is dropped only if it is not referenced in any constraints (for example, by foreign key definitions in another relation) or views or by any other elements. With the **CASCADE** option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

The **DROP TABLE** command not only deletes all the records in the table if successful, but also removes the table definition from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the **DELETE** command should be used instead of **DROP TABLE**.

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

1.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

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

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

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple. If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is not allowed in this case.

To drop a column, we must choose either **CASCADE** or **RESTRICT** for drop behavior. If **CASCADE** is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If **RESTRICT** is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column.

For example, the following command removes the attribute Address from the EMPLOYEE base table:

● **ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;**

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

ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT

‘333445555’;

Alter Table - Alter/Modify Column

To change the data type of a column in a table, use the following syntax:

**ALTER TABLE table_name
MODIFY column_name datatype;**

For example we can change the data type of the column named "DateOfBirth" from date to year in the "Persons" table using the following SQL statement:

**ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;**

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.



Chapter 2: Database Application Development

2.1 Introduction

We often encounter situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we may want to integrate a database applications with GUI or we may want to integrate with other existing applications.

2.2 Accessing Databases from applications

SQL commands can be executed from within a program in a host language such as C or Java. A language to which SQL queries are embedded are called Host language.

2.2.1 Embedded SQL

The use of SQL commands within a host language is called **Embedded SQL**. Conceptually, embedding SQL commands in a host language program is straight forward. SQL statements can be used wherever a statement in the host language is allowed. SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Any host language variable used to pass arguments into an SQL command must be declared in SQL.

There are two complications:

1. Data types recognized by SQL may not be recognized by the host language and vice versa
 - This mismatch is addressed by casting data values appropriately before passing them to or from SQL commands.
2. SQL is set-oriented
 - Addressed using cursors

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host language variables must be prefixed by a colon(:) in SQL statements and be declared between the commands

EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION

The declarations are similar to C, are separated by semicolons. For example, we can declare variables c_sname, c_sid, c_rating, and c_age (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

EXEC SQL BEGIN DECLARE SECTION

```
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
```

EXEC SQL END DECLARE SECTION

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, c_sname has the type CHARACTER(20) when referred to in an SQL statement, c_sid has the type INTEGER, crating has the type SMALLINT, and c_age has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, **SQLCODE** and **SQLSTATE**.

- **SQLCODE** is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.
- **SQLSTATE**, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported.

One of these two variables must be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char [6] , that is, a character string five characters long.

Embedding SQL statements

All SQL statements embedded within a host program must be clearly marked with the details dependent on the host language. In C, SQL statements must be prefixed by **EXEC SQL**. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

Example: The following embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the sailors relation

```
EXEC SQL INSERT INTO sailors VALUES (:c_sname, :c_sid, :c_rating,:c_age);
```

The **SQLSTATE** variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the **WHENEVER** command to simplify this task:

EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [CONTINUE|GOTO stmt]

If **SQLERROR** is specified and the value of **SQLSTATE** indicates an exception, control is transferred to **stmt**, which is presumably responsible for error and exception handling. Control is also transferred to **stmt** if **NOT FOUND** is specified and the value of **SQLSTATE** is 02000, which denotes **NO DATA**.

2.2.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on sets of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation- this mechanism is called a **cursor**

We can declare a cursor on any relation or on any SQL query. Once a cursor is declared, we can

- **open** it (positions the cursor just before the first row)
- **Fetch** the next row
- **Move** the cursor (to the next row,to the row after the next n, to the first row or previous row etc by specifying additional parameters for the fetch command)
- **Close** the cursor

Cursor allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- We usually need to open a cursor if the embedded statement is a **SELECT**. we can avoid opening a cursor if the answer contains a single row
- **INSERT**, **DELETE** and **UPDATE** statements require no cursor. some variants of **DELETE** and **UPDATE** use a cursor.

Examples:

- i) Find the name and age of a sailor, specified by assigning a value to the host variable **c_sid**, declared earlier

```
EXEC SQL SELECT s.sname,s.age
INTO :c_sname, :c_age
FROM Sailaor s
WHERE s.sid=:c.sid;
```

The **INTO** clause allows us assign the columns of the single answer row to the host variable c_sname and c_age. Therefore, we do not need a cursor to embed this query in a host language program.

- ii) Compute the name and ages of all sailors with a rating greater than the current value of the host variable c_minrating

```
SELECT s.sname,s.age
FROM sailors s WHERE s.rating>:c_minrating;
```

The query returns a collection of rows. The INTO clause is inadequate. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT s.sname,s.age
FROM sailors s
WHERE s.rating>:c_minrating;
```

This code can be included in a C program and once it is executed, the cursor sinfo is defined.

We can open the cursor by using the syntax:

```
OPEN sinfo;
```

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When the cursor is opened, it is positioned just before the first row.

We can use the FETCH command to read the first row of cursor sinfo into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement, we can read all the rows computed by the query, one row at time.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

- iii) To retrieve the name, address and salary of an employee specified by the variable ssn

```
//Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2)   prompt("Enter a Social Security Number: ", ssn) ;
3)   EXEC SQL
4)     SELECT Fname, Minit, Lname, Address, Salary
5)     INTO :fname, :minit, :lname, :address, :salary
6)     FROM EMPLOYEE WHERE Ssn = :ssn ;
7)   if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)   else printf("Social Security Number does not exist: ", ssn) ;
9)   prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
[WITH HOLD]
FOR some query
[ORDER BY order-item-list ]
[FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or updatable cursor (FOR UPDATE). If it is updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if sinfo is an updatable cursor and open, we can execute the following statement:

```
UPDATE Sailors S
SET S.rating = S.rating -1
WHERE CURRENT of sinfo;
```

A cursor is updatable by default unless it is a scrollable or insensitive cursor in which case it is read-only by default.

If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the **FETCH** command can be used to position the cursor in very flexible ways; otherwise, only the basic **FETCH** command, which retrieves the next row, is allowed

If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior.

A holdable cursor is specified using the **WITH HOLD** clause, and is not closed when the transaction is committed.

Optional **ORDER BY** clause can be used to specify a sort order. The order-item-list is a list of order-items. An order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the **ORDER BY** clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on

ORDER BY minage ASC, rating DESC

The answer is sorted first in ascending order by minage, and if several rows have the same minage value, these rows are sorted further in descending order by rating

Rating	minage
8	25.5
3	25.5
7	35.0

Dynamic SQL

Dynamic SQL Allow construction of SQL statements on-the-fly. Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed. SQL provides some facilities to deal with such situations; these are referred to as **Dynamic SQL**.

Example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :csqlstring;
EXEC SQL EXECUTE readytogo;
```

- The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command
- The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*
- The third statement executes the command

2.3 An Introduction to JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. A DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be compiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language.

- In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *Without recompilation*.

- While Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable
- In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously
- ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection
- All direct interaction with a specific DBMS happens through a DBMS-specific driver.

A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager a driver does not necessarily need to interact with a DBMS that understands SQL. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections. An application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes. While a connection is open, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

2.3.1 Architecture

The architecture of JDBC has four main components:

- Application
- Driver manager
- Drivers
- Data sources

Application

- initiates and terminates the connection with a data source
- sets transaction boundaries, submits SQL statements and retrieves the results

Driver manager

- Load JDBC drivers and pass JDBC function calls from the application to the correct driver
- Handles JDBC initialization and information calls from the applications and can log all function calls
- Performs some rudimentary error checking

Drivers

- Establishes the connection with the data source
- Submits requests and returns request results
- Translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard

Data sources

- Processes commands from the driver and returns the results

Drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:

Type I Bridges:

- This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS.
- An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge.
- Advantage:
 - it is easy to piggyback the application onto an existing installation, and no new drivers have to be installed.
- Drawbacks:
 - The increased number of layers between data source and application affects performance
 - the user is limited to the functionality that the ODBC driver supports.

Type II Direct Translation to the Native API via Non-Java Driver:

- This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source.
- The driver is usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source.
- Advantage
 - This architecture performs significantly better than a JDBC-ODBC bridge.
- Disadvantage
 - The database driver that implements the API needs to be installed on each computer that runs the application.

Type III~~Network Bridges:

- The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations.
- In this case, the driver on the client site is not DBMS-specific.
- The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server.
- The middleware server can then use a Type II JDBC driver to connect to the data source.

Type IV-Direct Translation to the Native API via Java Driver:

- Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets
- In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system.
- This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

2.4 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling.

The classes and interfaces are part of the `java.sql` package. JDBC 2.0 also includes the `javax.sql` package, the JDBC Optional Package. The package `javax.sql` adds, among other things, the capability of connection pooling and the Row-Set interface.

2.4.1 JDBC Driver Management

In JDBC, data source drivers are managed by the `DriverManager` class, which maintains a list of all currently loaded drivers. The `DriverManager` class has methods `registerDriver`, `deregisterDriver`, and `getDrivers` to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. The following Java example code explicitly loads a JDBC driver:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

There are two other ways of registering a driver. We can include the driver with `-Djdbc.drivers=oracle/jdbc.driver` at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to changes at the driver level.

After registering the driver, we connect to the data source.

2.4.2 Connections

A session with a data source is started through creation of a `Connection` object; Connections are specified through a JDBC URL, a URL that uses the `jdbc` protocol. Such a URL has the form

`jdbc:<subprotocol>:<otherParameters>`

```
String url = .. jdbc:oracle:www.bookstore.com:3083..
Connection connection;
try
{
    Connection connection =
        DriverManager.getConnection(url, userId,password);
}
catch(SQLException excpt)
{
    System.out.println(excpt.getMessage());
    return;
}
```

Program code: Establishing a Connection with JDBC

In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If `autocommit` is set for a connection, then each SQL statement is

considered to be its own transaction. If **autocommit** is off, then a series of statements that compose a transaction can be committed using the `commit()` method of the `Connection` class, or aborted using the `rollback()` method. The `Connection` class has methods to set the autocommit mode (`Connection.setAutoCommit`) and to retrieve the current autocommit mode (`getAutoCommit`). The following methods are part of the `Connection` interface and permit setting and getting other properties:

- `public int getTransactionIsolation() throws SQLException` and
`public void setTransactionIsolation(int 1) throws SQLException.`
 - These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation are possible, and argument / can be set as follows:
 - `TRANSACTION_NONE`
 - `TRANSACTION_READ_UNCOMMITTED`
 - `TRANSACTION_READ_COMMITTED`
 - `TRANSACTION_REPEATABLE_READ`
 - `TRANSACTION_SERIALIZABLE`
- `public boolean getReadOnly() throws SQLException` and
`public void setReadOnly(boolean readOnly) throws SQLException.`
 - These two functions allow the user to specify whether the transactions executed through this connection are read only.
- `public boolean isClosed() throws SQLException.`
 - Checks whether the current connection has already been closed.
- `setAutoCommit` and `get AutoCommit`.

In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

2.4.3 Executing SQL Statements

JDBC supports three different ways of executing statements:

- `Statement`
- `PreparedStatement`, and
- `CallableStatement`.

The **Statement** class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.

The **PreparedStatement** class dynamically generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object is created.

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, ?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(1, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);
int numRows = pstmt.executeUpdate();
```

program code: SQL Update Using a PreparedStatement Object

The SQL query specifies the query string, but uses "?" for the values of the parameters, which are set later using methods `setString`, `setFloat`, and `setInt`. The placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements. The method `setString` is one way to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use `clearParameters()` before setting parameter values in order to remove any old data.

There are different ways of submitting the query string to the data source. In the example, we used the **executeUpdate** command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The `executeUpdate` method returns

- an integer indicating the number of rows the SQL statement modified;
- 0 for successful execution without modifying any rows.

The `executeQuery` method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a `ResultSet` object.

2.4.4 ResultSets

ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions. In its most basic form, the **ResultSet** object allows us to read one row of the output of the query at a time. Initially, the **ResultSet** is positioned before the first row, and we have to retrieve the first row with an explicit call to the **next()** method. The next method returns false if there are no more rows in the query answer, and true otherwise. The code fragment shown below illustrates the basic usage of a **ResultSet** object:

```
ResultSet rs=stmt.executeQuery(sqlQuery);
// rs is now a cursor
// first call to rs.next() moves to the first record
// rs.next() moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next())
{
    // process the data
}
```

The **next()** allows us to retrieve the logically next row in the query answer, we can move about the query answer in other ways too:

- **previous()** moves back one row.
- **absolute(int num)** moves to the row with the specified number.
- **relative(int num)** moves forward or backward (if num is negative) relative to the current position. **relative (-1)** has the same effect as previous.
- **first()** moves to the first row, and **last()** moves to the last row.

Matching Java and SQL Data Types

In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL (e.g., passing information between the application and the data source through shared variables) arise again. To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types. Table 2.4.4 shows the accessor methods in a **ResultSet** object for the most common SQL datatypes.

With these accessor methods, we can retrieve values from the current row of the query result referenced by the **ResultSet** object. There are two forms for each accessor method. One method retrieves values by column index, starting at one, and the other retrieves values by column name.

The following example shows how to access fields of the current ResultSet row using accessor methods.

```
ResultSet rs=stmt.executeQuery(sqlQuery);

String sqlQuerYi
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next())
{
    isbn = rs.getString(l);
    title = rs.getString(" TITLE");
    // process isbn and title
}
```

SQL Type	Java class	ResultSet get method
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.util.Date	getDATE()
TIME	java.sql.Time	getTIME()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

Table 2.4.4 : Reading SQL Datatypes from a ResultSet Object

2.4.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in `java.sql` can throw an exception of the type `SQLException` if an error occurs. The information includes `SQLState`, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard `getMessage()` method inherited from `Throwable`, `SQLException` has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- `public String getSQLState()` returns an `SQLState` identifier based on the SQL:1999 specification
- `public int getErrorCode ()` retrieves a vendor-specific error code.

- public SQLException getNextException() gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQLWarning is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. Warnings are not thrown like other exceptions, and they are not caught as part of the try-catch block around a java.sql statement. We need to specifically test whether warnings exist. **Connection**, **Statement**, and **ResultSet** objects all have a **getWarnings()** method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through **clearWarnings()**. **Statement** objects clear warnings automatically on execution of the next statement; **ResultSet** objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown below:

```
try
{
    stmt = con.createStatement();
    warning = con.getWarnings();
    while( warning != null)
    {
        // handleSQLWarnings // code to process warning
        warning = warning.getNextWarning(); //get next warning
    }
    con.clearWarnings();
    stmt.executeUpdate( queryString );
    warning = stmt.getWarnings();
    while( warning != null)
    {
        // handleSQLWarnings // code to process warning
        warning = warning.getNextWarning(); //get next warning
    }
} // end try
catch ( SQLException SQLe )
{
    // code to handle exception
} // end catch
```

2.4.6 Examining Database Metadata

We can use the `DatabaseMetaData` object to obtain information about the database system itself, as well as information from the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaData();
System.out.println("Driver Information:");
System.out.println("Name:" + md.getDriverName()
    + "; version:" + md.getDriverVersion());
```

The `DatabaseMetaData` object has many more methods (in JDBC 2.0, exactly 134). Some of the methods are:

- **public ResultSet getCatalogs() throws SQLException.** This function returns a `ResultSet` that can be used to iterate over all **public int getMaxConnections() throws SQLException** the catalog relations. This function returns the maximum number of connections possible.

Example: code fragment that examines all database metadata

```
DatabaseMetaData dmd = con.getMetaData();
ResultSet tablesRS = dmd.getTables(null,null,null,null);
String tableName;
while(tablesRS.next())
{
    tableName = tablesRS.getString("TABLE_NAME");
    // print out the attributes of this table
    System.out.println("The attributes of table"
        + tableName + " are:");
    ResultSet columnsRS = dmd.getColumns(null,null,tableName, null);
    while (columnsRS.next())
    {
        System.out.print(columnsRS.getString("COLUMN_NAME")
            + " ");
    }
    // print out the primary keys of this table
    System.out.println("The keys of table" + tableName + " are:");
    ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
    while (keysRS.next ())
    {
        System.out.print(keysRS.getString("COLUMN_NAME") + " ");
    }
}
```

```

    }

```

7 steps for jdbc :

1. Import the package
-- import java.sql.*;
2. Load and register the driver
--Class.forName();
3. Establish the connection
-- Connection con;
4. Create a Statement object
-- Statement st;
5. Execute a query
-- st.execute();
6. Process the result
7. Close the connection

Step 2: load the corresponding JDBC driver

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Step 3: create a session with data source through creation of Connection object.

```
Connection connection = DriverManager.getConnection(database_url,
                                                userId, password);
```

EX: Connection con= DriverManager.getConnection

```
("jdbc:oracle:thin:@localhost:1521:xeid","system","ambika");
```

Step 4:create a statement object

- JDBC supports three different ways of executing statements:
- Statement
- PreparedStatement and
- CallableStatement.
- The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.
- The PreparedStatement class dynamically generates precompiled SQL statements that can be used several times
- CallableStatement are used to call stored procedures from JDBC. CallableStatement is a subclass of PreparedStatement and provides the same functionality.
- Example:

```
Statement st=con.createStatement();
```

Step 5: executing a query

```
String query="select * from students where usn='4VV15CS001';
```

```
ResultSet rs=st.executeQuery(query);
```

Step 6: process the result

```
String sname=rs.getString(2);
```

```
System.out.println(sname);
```

Step 7: close the connection

```
con.close();
```

```
import java.sql.*;

public class Demo {
    public static void main(String[] args) {
        try {
            String query="select * from students where usn='4VV15CS001'";
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe","system","ambika");
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery(query);
            String s=rs.getString(1);
            System.out.println(s);
            con.close();
        }
        catch(Exception e)
        {
        }
    }
}
```

2.5 SQLJ: SQL-JAVA

SQLJ enables applications programmers to embed SQL statements in Java code in a way that is compatible with the Java design philosophy

Example: SQLJ code fragment that selects records from the Books table that match a given author.

```

String title; Float price; String author;

#sql iterator Books (String title, Float price);

Books books;

#sql books = {

    SELECT title, price INTO :title, :price

    FROM Books WHERE author = :author

};

while (books.next()) {

    System.out.println(books.title() + ", " + books.price());

}

books.close();

```

SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class.

Usage of an iterator in SQLJ goes through five steps:

1. Declare the Iterator Class: In the preceding code, this happened through the statement

```
#sql iterator Books (String title, Float price);
```

This statement creates a new Java class that we can use to instantiate objects.

2. Instantiate an Iterator Object from the New Iterator Class:

We instantiated our iterator in the statement Books books;;

3. Initialize the Iterator Using a SQL Statement:

In our example, this happens through the statement #sql books =....

4. Iteratively, Read the Rows From the Iterator Object:

This step is very similar to reading rows through a ResultSet object in JDBC.

5. Close the Iterator Object.

There are two types of iterator classes:

- named iterators
- positional iterators

For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name. This method is used in our example.

For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a `FETCH ... INTO` construct, similar to Embedded SQL

We can make the iterator a positional iterator through the following statement:

```
#sql iterator Books (String, Float);
```

We then retrieve the individual rows from the iterator as follows:

```
while (true)
{
    #sql { FETCH :books INTO :title, :price, };
    if (books.endFetch())
    { break; }
    // process the book
}
```

2.6 STORED PROCEDURES

Stored procedure is a set of logical group of SQL statements which are grouped to perform a specific task.

Benefits :

- reduces the amount of information transfer between client and database server
- Compilation step is required only once when the stored procedure is created. Then after it does not require recompilation before executing unless it is modified and reutilizes the same execution plan whereas the SQL statements need to be compiled every time whenever it is sent for execution even if we send the same SQL statement every time
- It helps in re usability of the SQL code because it can be used by multiple users and by multiple clients since we need to just call the stored procedure instead of writing the same SQL statement every time. It helps in reducing the development time

Syntax:

```
Create or replace procedure <procedure Name> [(arg1 datatype, arg2 datatype)]
```

Is/As

```
<declaration>
```

Begin

```
<SQL Statement>
```

Exception**2.6.1 Creating a Simple Stored Procedure**

Consider the following schema:

```
Student(usn:string,sname:string)
```

us now write a stored procedure to retrieve the count of students with sname 'Akshay'

```
create or replace procedure ss
```

```
:-
```

```
stu_cnt int;
```

```
begin
```

```
select count(*) into stu_cnt from students where sname='AKSHAY';
```

```
dbms_output.put_line('the count of student is :' || stu_cnt);
```

```
end ss;
```

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT.

- IN parameters are arguments to the stored procedure
- OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process
- INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values

Example:

```
CREATE PROCEDURE AddInventory (
    IN book_isbn CHAR(10),
    IN addedQty INTEGER)
    UPDATE Books SET qty_in_stock = qtyin_stock + addedQty
    WHERE bookisbn = isbn
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language.

For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
    char isbn[10];
    long qty;
EXEC SQL END DECLARE SECTION
// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Procedures without parameters are called **static procedures** and with parameters are called **dynamic procedures**.

Example: stored procedure with parameter

```
create or replace procedure emp(Essn int)
as
    eName varchar(20);
begin
    select fname into eName from employee where ssn=Essn and dno=5;
    dbms_output.put_line(' the employee name is :||Essn ||eName);
end emp;
```

2.6.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CALL storedProcedureName(arg1, arg2, .. ,argN);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatement class. A stored procedure could contain multiple SQL statements or a series of SQL statements—thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt= con. prepareCall(" {call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQuery();
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class

#sql Iterator CustomerInfo(int cid, String cname, int count);

// create the cursor

CustomerInfo customerinfo;

// call the stored procedure

#sql customerinfo = {CALL ShowNumberOfOrders};

while (customerinfo.next())

{
    System.out.println(customerinfo.cid() + "," +
    customerinfo.count());
}
```

2.6.3 SQL/PSM

SQL/Persistent Stored Modules is an ISO standard mainly defining an extension of SQL with procedural language for use in stored procedures.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,... , parameterN)
local variable declarations
procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameter1, ..., parameterN)
    RETURNS sqlDataType
    local variable declarations
    function code;
```

Example:

```
CREATE FUNCTION RateCustomer (IN custId INTEGER, IN year INTEGER)
    RETURNS INTEGER
    DECLARE rating INTEGER;
    DECLARE numOrders INTEGER;
    SET numOrders = (SELECT COUNT(*) FROM Orders O WHERE O.tid = custId);
    IF (numOrders > 10) THEN rating=2;
    ELSEIF (numOrders>5) THEN rating=1;
    ELSE rating=0;
    END IF;
    RETURN rating;
```

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;
ELSEIF statements;
ELSE statements;
END IF
```

- Loops are of the form

LOOP

statements:

END LOOP

Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables. We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.

•

Chapter 3: Internet Applications

3.1 Introduction

Data-intensive is used to describe applications with a need to process large volumes of data. The volume of data that is processed can be in the size of terabytes and petabytes and this type of data is also referred as big data. Data-intensive computing is used in many applications ranging from social networking to computational science where a large amount of data needs to be accessed, stored, indexed and analyzed. It is more challenging as the amount of data keeps on accumulating over time and the rate at which the data is generating also increases

3.2 THE THREE-TIER APPLICATION ARCHITECTURE

Data-intensive Internet applications can be understood in terms of three different functional components:

1. Data management
2. Application logic
3. Presentation

→ component that handles data management usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

3.2.1 Single-Tier

Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface. The application typically ran on a mainframe, and users accessed it through dumb terminals that could perform only data input and display.



Figure 3.2.1 : A Single-Tier Architecture

- **Benefit**

- easily maintained by a central administrator

- **Drawback:**

- Users expect graphical interfaces that require much more computational power than simple dumb terminals.
- Do not scale to thousands of users

3.2.2 Two-tier architectures

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. What part of the functionality the client implements, and what part is left to the server, can vary.

In the traditional client server architecture, the client implements just the graphical user interface—such clients are often called **thin clients**; the server implements both the business logic and the data management.

Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the server level; such clients are often called **thick clients**.

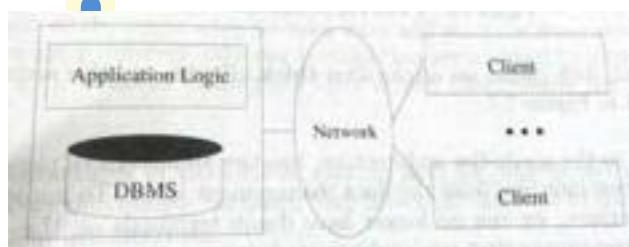


Figure3.2.2(a) : A Two-Server Architecture: thin client

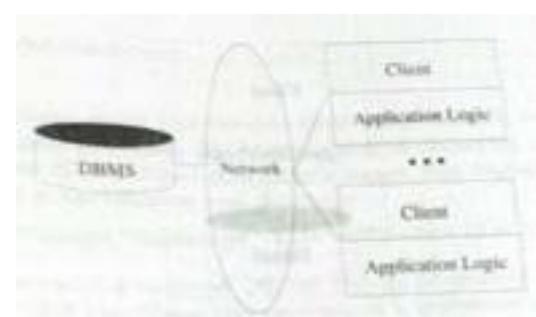


Figure3.2.2(a) : A Two-Server Architecture: thick client

The thick-client model has several disadvantages when compared to the thin client model

1. There is no central place to update and maintain the business logic, since the application code runs at many client sites.
2. A large amount of trust is required between the server and the clients. As an example, the DBMS of a bank has to trust the application executing at an ATM machine to leave the database in a consistent state.
3. Thick-client architecture does not scale with the number of clients; it typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

Single-tier architecture v/s Two-tier architectures

- Compared to the single-tier architecture, two-tier architectures physically separate the user interface from the data management layer
- To implement two tier architectures, we can no longer have dumb terminals on the client side, we require computers that run sophisticated presentation code and possibly, application logic

3.2.3 Three-Tier Architectures

The thin-client two-tier architecture essentially separates presentation issues from the rest of the application. The three-tier architecture goes one step further, and also separates application logic from data management:

- Presentation Tier
- Middle Tier
- Data Management Tier

Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites

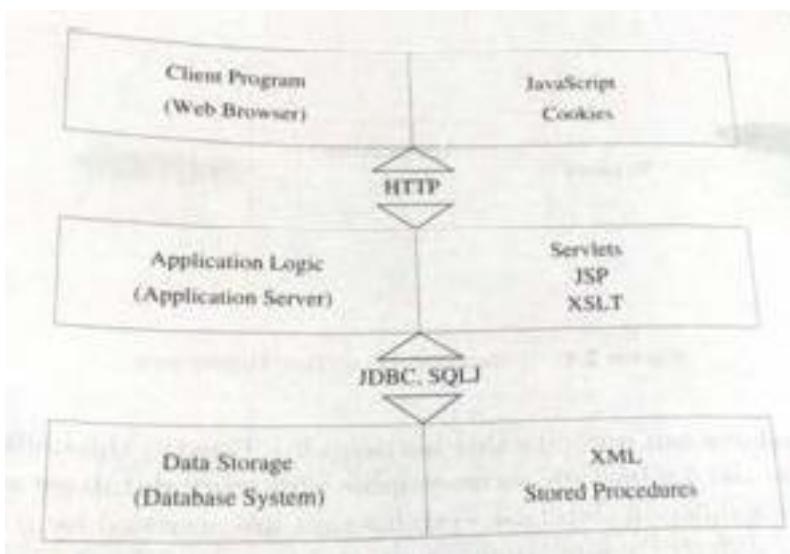


Figure 3.2.3: Technologies for the Three Tiers

i

HTML Forms

HTML forms are a common way of communicating data from the client tier to the middle tier.

The general format of a form :

- METHOD: The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices: GET and POST

HTML tags except another FORM element

Passing Arguments to Server-Side Scripts

There are two different ways to submit HTML Form data to the webserver. If the method GET is used, then the contents of the form are assembled into a query URI (as discussed next) and sent to the server. If the method POST is used, then the contents of the form are encoded as in the GET method, but the contents are sent in a separate data block instead of appending them directly to the URI. Thus, in the GET method the form contents are directly visible to the user as the constructed URI, whereas in the POST method, the form contents are sent inside the HTTP request message body and are not visible to the user.

JavaScript

JavaScript is a scripting language at the client tier with which we can add programs to webpages that run directly at the client. JavaScript is often used for the following types of computation at the client:

- **Browser Detection:** JavaScript can be used to detect the browser type and load a browser-specific page.
- **Form Validation:** JavaScript is used to perform simple consistency checks on form fields
- **Browser Control:** This includes opening pages in customized windows; examples include the annoying pop-up advertisements that you see at many websites, which are programmed using JavaScript.

JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag

```
<SCRIPT LANGUAGE=" JavaScript" SRC="validateForm.js"> </SCRIPT>
```

The SCRIPT tag has the attribute LANGUAGE, which indicates the language in which the script is written. For JavaScript, we set the language attribute to JavaScript. Another attribute of the SCRIPT tag is the SRC attribute, which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a '.js' extension.

Style Sheets

Style sheet is a method to adapt the same document contents to different presentation formats. A style sheet contains instructions that tell a web how to translate the data of a document into a presentation that is suitable for the client's display. The use of style sheets has many advantages:

- we can reuse the same document many times and display it differently depending on the context
- we can tailor the display to the reader's preference such as font size, color style, and even level of detail.
- we can deal with different output formats, such as different output devices (laptops versus cell phones), different display sizes (letter versus legal paper), and different display media (paper versus digital display)
- we can standardize the display format within a corporation and thus apply style sheet conventions to documents at any time.
- changes and improvements to these display conventions can be managed at a central place.

There are two style sheet languages:

- XSL
- CSS

Cascading Style Sheets (CSS)

- CSS was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves
- CSS defines how to display HTML elements.
- Styles are normally stored in style sheets, which are files that contain style definitions.
- Many different HTML documents, such as all documents in a website, can refer to the same CSS.
- Thus, we can change the format of a website by changing a single file.
- Each line in a CSS sheet consists of three parts; a selector, a property, and a value. They are syntactically arranged in the following way:

selector {property: value}

- The selector is the element or tag whose format we are defining.
- The property indicates the tag's attribute whose value we want to set in the style sheet
- Example: BODY {BACKGROUND-COLOR: yellow}

P {MARGIN-LEFT: 50px; COLOR: red}

XSL

- XSLT is a language for transforming XML documents. It can be used to transform the input XML document into a XML document with another structure
- For example, with XSLT we can change the order of elements that we are displaying (e.g.; by sorting them), process elements more than once, suppress elements in one place and present them in another, and add generated text to the presentation

3.2.3.2 Overview of the Middle Tier

The middle layer runs code that implements the business logic of the application. The middle tier code is responsible for supporting all the different roles involved in the application. For example, in an Internet shopping site implementation, we would like

- customers to be able to browse the catalog and make purchases
- administrators to be able to inspect current inventory, and
- data analysts to ask summary queries about purchase histories
- Each of these roles can require support for several complex actions

The first generation of middle-tier applications was stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. Programmers quickly realized that

interaction with a stand-alone application was quite costly. The overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. Most of today's large-scale websites use an application server to run application code at the middle tier. Application server provides the run-time for several technologies that can be used to program middle-tier application components.

CGI: The Common Gateway Interface

The Common Gateway Interface connects HTML forms with application programs.

- It is a protocol that defines how arguments from forms are passed to programs at the server side
- CGI is the part of the Web server that can communicate with other programs running on the server
- With CGI, the Web server can call up a program, while passing user-specific data to the program (such as what host the user is connecting from, or input the user has supplied using HTML form syntax)
- The program then processes that data and the server passes the program's response back to the Web browser.

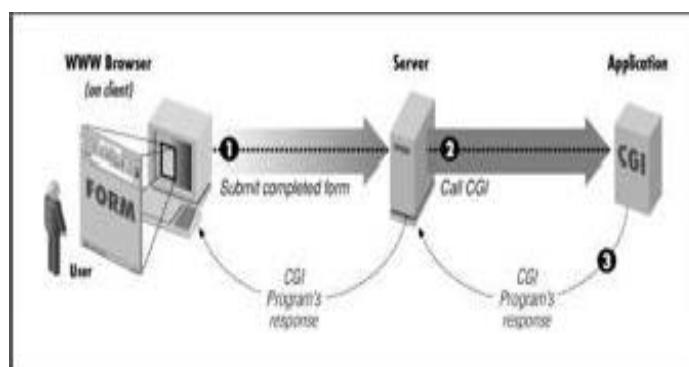


Figure: Simple diagram of CGI

```

<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find_books.cgi" METHOD=POST>
  Type an author name:<br>
  <INPUT TYPE="text" NAME="authorName">
  <INPUT TYPE="submit" value="Send it">
  <INPUT TYPE="reset" VALUE="Clear form" >
</FORM>
  
```

```
</FORM>
</BODY></HTML>
```

Program fragment: A Sample 'web Page Where Form Input Is Sent to a CGI Script

Application Servers

Application logic can be enforced through server-side programs that are invoked using the CGI protocol. However, since each page request results in the creation of a new process, this solution does not scale well to a large number of simultaneous requests. An application server maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request. They facilitate concurrent access to several heterogeneous data sources (e.g., by providing JDBC drivers), and provide session management services.

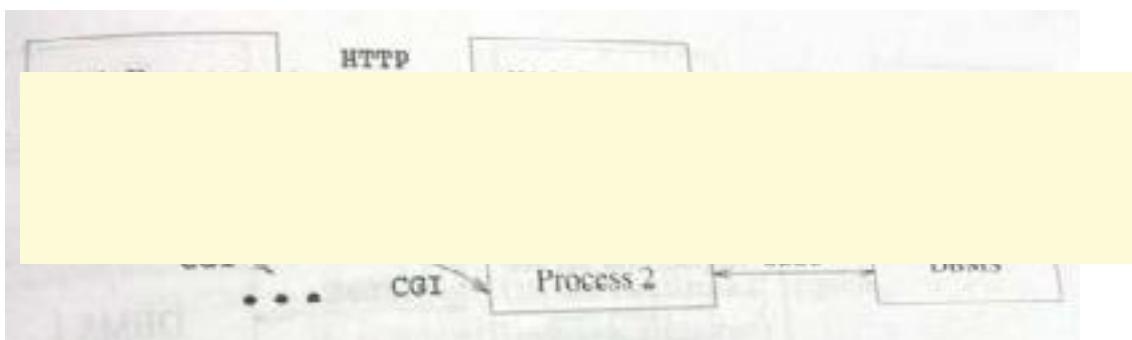


Fig: Process Structure with CGI Scripts

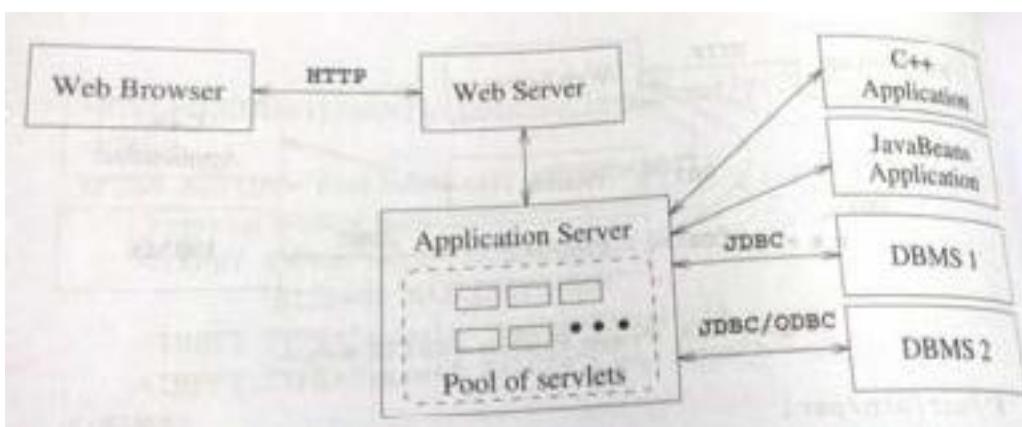


Fig: Process Structure in the Application Server Architecture

Servlets

Java servlets are pieces of Java code that run on the middle tier, in either webservers or application servers. Servlets can build webpages, access databases, and maintain state. Servlets usually handle requests from HTML forms and maintain state between the client and the server.

Servlets are compiled Java classes executed and maintained by a servlet container. The servlet container manages the lifespan of individual servlets by creating and destroying them. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by webservers.

JavaServer Pages

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases

JavaServer pages (.JSPs) interchange the roles of output and application logic. JavaServer pages are written in HTML with servlet-like code embedded in special HTML tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic, whereas servlets are better suited for complex application logic.

Maintaining State

There is a need to maintain a user's state across different pages. As an example, consider a user who wants to make a purchase at the Barnes and Nobble website. The user must first add items into her shopping basket, which persists while she navigates through the site. Thus, we use the notion of state mainly to remember information as the user navigates through the site.

The HTTP protocol is stateless. We call an interaction with a webserver stateless if no information is retained from one request to the next request. We call an interaction with a webserver stateful, or we say that state is maintained, if some memory is stored between requests to the server, and different actions are taken depending on the contents stored.

Since we cannot maintain state in the HTTP protocol, where should we maintain state? There are basically two choices:

- We can maintain state in the middle tier, by storing information in the local main memory of the application logic, or even in a database system
- Alternatively, we can maintain state on the client side by storing data in the form of a cookie.

Maintaining State at the Middle Tier

At the middle tier, we have several choices as to where we maintain state.

- First, we could store the state at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck
- An alternative is to store state in main memory at the middle tier. The drawbacks are that this information is volatile and that it might take up a lot of main memory
- We can also store state in local files at the middle tier, as a compromise between the first two approaches.

Maintaining State at the Presentation Tier: Cookies

A **cookie** is a collection of $(name, value)$ pairs that can be manipulated at the presentation and middle tiers. Cookies are easy to use in Java servlets and Java server Pages. They survive several client sessions because they persist in the browser cache even after the browser is closed. One disadvantage of cookies is that they are often perceived as being invasive, and many users disable cookies in their Web browser; browsers allow users to prevent cookies from being saved on their machines. Another disadvantage is that the data in a cookie is currently limited to 4KB, but for most applications this is not a bad limit.

Advantages of Three-Tier Architecture

The

and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.

- **Thin Clients:** Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- **Integrated Data Access:** In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- **Scalability to Many Clients:** Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately.
- **Software Development Benefits:** By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application

tier can be built out of reusable components that can be individually developed, debugged, and tested.



Question Bank

1. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?
2. Describe the six clauses in the syntax of an SQL retrieval query. Show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?
3. Describe conceptually how an SQL retrieval query will be executed by specifying the conceptual order of executing each of the six clauses.
4. Explain how the GROUP BY clause works. What is the difference between the WHERE and HAVING clause?
5. Explain insert, delete and update statements in SQL and give example for each.
6. Write a note on:

i) Views in SQL

- Explain DROP command with an example.
- How is view created and dropped? What problems are associated with updating views?
- How are triggers and assertions defined in SQL? Explain.

- Consider the following schema for a COMPANY database:

EMPLOYEE (Fname, Lname, Ssn, Address, Super-ssn, Salary, Dno)

DEPARTMENT (Dname, Dnumber, Mgr-ssn, Mgr-start-date)

DEPT-LOCATIONS (Dnumber, Dlocation)

PROJECT (Pname, Pnumber, Plocation, Dnum)

WORKS-ON (Essn, Pno, Hours)

DEPENDENT (Essn, Dependent-name, Sex, Bdate, Relationship)

write the SQL query for the following:

- i) List the names of managers who have at least one dependent.
- ii) Retrieve the list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.
- iii) For each project, retrieve the project number, the project name, and the number of employees who work on that project.
- iv) 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.
- v) For each project, retrieve the project number, the project name, and the number of employees from department 4 who work on the project.

11. Consider the following tables:

Works(Pname,Cname,Salary)

Lives(Pname,Street,City)

Located-in(Cname,City)

Manager(Pname,mgpname)

write the SQL query for the following:

i) Find the names of all persons who live in the city 'Mumbai';

ii) Retrieve the Rs.50,000 of all person of 'Infosys' whose salary

Rs.50,000.

iii) Find the names of all persons who live and work in the same city.

v) Find the average salary of all 'Infosyians'.

12. Consider the following schema

Sailors(sid,sname,rating,age)

Boats(bid,bname,color)

Reserves(sid,bid,day)

write the SQL query for the following:

i) Retrieve the sailors name who have reserved red and green boats.

ii) Retrieve the sailors names with age over 20 years and reserved black boat.

iii) Retrieve the number of boats which are not reserved.

iv) Retrieve the sailors names who have reserved green boat on Monday.

v) Retrieve the sailors names who is oldest sailor with rating 10.

13. Consider the following schema and write the SQL queries:

STUDENT-ID,SNAME,MAJOR,GPA)

FACULTY(FACULTY_ID,FNAME,DEPT,DESIGNATION,SALARY)

COURSE(COURSE_ID,CNAME,FACULTY_ID)

ENROLL(COURSE_ID,STUDENT_ID,GRADE)

i) Retrieve the student name who is studying under faculties of "Mechanical dept".

ii) Retrieve the student name who have enrolled under any of the courses in which 'kumar' has enrolled.

iii) Retrieve the faculty name who earn salary which is greater than the average salary of all the faculties.

iv) Retrieve the sname who are not bee taught by faculty 'kumar'.

v) Retrieve the faculty names who are assistant professors of CSE dept.

14. How do we use SQL statements within a host language? How do we check for errors in statement execution?

15. Define cursor. what properties can cursors have?
16. What is Dynamic SQL and how is it different from Embedded SQL?
17. What is JDBC and what are its advantages?
18. What are the components of the JDBC architecture? Describe four different architectural alternatives for JDBC drivers.
19. With an example, explain SQLJ?
20. Illustrate with an example stored procedure. Mention its benefits.
21. What is a three-tier architecture? What advantages does it offer over single tier and two-tier architectures? Give a short overview of the functionality at each of the three tiers.



Module 4

Chapter 1: Database Design Theory

- 4.0 Introduction
- 4.1 Objective
- 4.2 Introduction to DB design
- 4.3 Informal Design Guidelines for Relation Schemas
 - 4.3.1 Imparting Clear Semantics to Attributes in Relations
 - 4.3.2 Redundant Information in Tuples and Update Anomalies
 - 4.3.3 NULL Values in Tuples
 - 4.3.4 Generation of Spurious Tuples
- 4.4 Functional Dependencies
 - 4.4.1 Normalization of Relations
 - 4.4.2 Practical Use of Normal Forms
 - 4.4.3 Definitions of Keys and Attributes Participating in Keys
 - 4.4.4 First Normal Form
 - 4.4.5 Second Normal Form
 - 4.4.6 Third Normal Form
- 4.5 General Definition of Second and Third Normal Form
- 4.6 Boyce-Codd Normal Form
- 4.7 Multivalued Dependency and Fourth Normal Form
 - 4.7.1 Formal Definition of Multivalued Dependency
- 4.8 Join Dependencies and Fifth Normal Form
- 4.9 Inference Rules for Functional Dependencies
- 4.10 Equivalence of Sets of Functional Dependencies
- 4.11 Sets of Functional Dependencies
- 4.12 Properties of Relational Decompositions
- 4.13 Algorithms for Relational Database Schema Design
 - 4.13.1 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas
 - 4.13.2 Nonadditive Join Decomposition into BCNF Schemas
 - 4.13.3 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas
- 4.14 About Nulls, Dangling Tuples, and Alternative Relational Designs
 - 4.14.1 Problems with NULL Values and Dangling Tuples
- 4.15 Other Dependencies and Normal Forms
 - 4.15.1 Inclusion Dependencies
 - 4.15.2 Template Dependencies
 - 4.15.3 Functional Dependencies Based on Arithmetic Functions and Procedures
 - 4.15.4 Domain-Key Normal Form
- 4.16 Assignment Questions
- 4.17 Expected Outcome
- 4.18 Further Reading

4.0 Introduction

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. This module discuss the basic and higher normal forms.

4.1 Objectives

- ❖ To study the process of normalization and refine the database design
- ❖ To normalize the tables upto 4NF and 5NF
- ❖ To study lossless and lossy join operations
- ❖ To study inference rules
- ❖ To study other dependencies and Normal Forms.

4.2 Introduction to DB design

Each relation schema consists of a number of attributes, and the relational database schema consists of a number of relation schemas. So far, we have assumed that attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or Enhanced-ER (EER) data model. These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations.

can discuss the goodness of relation schemas:

1. The logical (or conceptual) level—how users interpret the relation schemas and the meaning of their attributes.
2. The implementation (or physical storage) level—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations

An Example

- STUDENT relation with attributes: studName, rollNo, gender, studDept
- DEPARTMENT relation with attributes: deptName, officePhone, hod
- Several students belong to a department
- studDept gives the name of the student's department

Correct schema:

Student

Department

Incorrect schema:

Studdept

StudName	rollNo	gender	deptName	officePhone	HOD

Problems with bad schema

- **Redundant storage of data:**
 - Office Phone & HOD info -stored redundantly once with each student that belongs to the department
 - wastage of disk space
- **A program that updates Office Phone of a department**
 - must change it at several places
 - more running time
 - error -prone

4.3 Informal Design Guidelines for Relation Schemas

- Four informal guidelines that may be used as measures to determine the quality of relation schema design:
 - 1.
 2. Reducing the redundant information in tuples
 3. Reducing the NULL values in tuples
 4. Disallowing the possibility of generating spurious tuples
- These measures are not always independent of one another

4.3.1 Imparting Clear Semantics to Attributes in Relations

- **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple
- Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them
- The easier it is to explain the semantics of the relation, the better the relation schema design will be

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

- if a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning
- if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1

EMP_DEPT						
ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
EMP_PROJ						
SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION	

Fig: schema diagram for company database

- Both the relation schemas have clear semantics
- A tuple in the EMP_DEPT relation schema represents a single employee but includes additional information—the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager.
- A tuple in the EMP_PROJ relates an employee to a project but also includes the employee name (Ename), project name (Pname), and project location (Plocation)
- logically correct but they violate Guideline 1 by mixing attributes from distinct real-world entities:
 - EMP_DEPT mixes attributes of employees and departments
 - EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship
- They may be used as views, but they cause problems when used as base relations

4.3.2 Redundant Information in Tuples and Update Anomalies

- One goal of schema design is to minimize the storage space used by the base relations
- 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 with that for an EMP_DEPT base relation
- In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department

- In contrast, each department's information appears only once in the DEPARTMENT relation. Only the department number Dnumber is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	gender	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
Alois	J	Zeloya	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
Ramish	K	Narayan	666884444	1982-09-15	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

Figure 1: One possible database state for the COMPANY relational database schema**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

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				
Ename	Dependent_name	gender	Bdate	Relationship
333445555	Alice	F	1988-04-05	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

Esn	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

Figure 1 : One possible database state for the COMPANY relational database schema

EMP_DEPT							
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dm	Redundancy
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555	
Wong, Franklin T.	333445555	1955-12-08	63B 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	

EMP_PROJ					
Ssn	Pnumber	Hours	Redundancy		
			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

Fig: Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 1

- Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These can be classified into:
 - insertion anomalies
 - deletion anomalies,
 - modification anomalies

Insertion Anomalies

- Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

 1. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs
 - For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT
 - In the design of Employee in fig 1, 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
 2. It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation.

- This violates the entity integrity for EMP_DEPT because Ssn is its primary key
- This problem does not occur in the design of Figure 1 because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies

- The problem of deletion anomalies is related to the second insertion anomaly situation just discussed
 - If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database
 - This problem does not occur in the database of Figure 2 because DEPARTMENT tuples are stored separately.

Modification Anomalies

- In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent
- If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong

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
- These guidelines may sometimes have to be violated in order to improve the performance of certain queries.

4.3.3 NULL Values in Tuples

- If many of the attributes do not apply to all tuples in the relation, we end up with many **NULLs** in those tuples

level

- may also lead to problems with understanding the meaning of the attributes
- may also lead to problems with specifying JOIN operations
- how to account for them when aggregate operations such as COUNT or SUM are applied
- SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.
- Moreover, NULLs can have multiple interpretations, such as the following:
 - The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
 - The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
 - The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Guideline 3

- As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL
- If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation
- Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns with the appropriate key columns
- For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

4.3.4 Generation of Spurious Tuples

- Consider the two relation schemas EMP_LOCS and EMP_PROJ1 which can be used instead of the single EMP_PROJ

EMP_LOCS

Ename	Plocation
-------	-----------

PK.

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
-----	---------	-------	-------	-----------

PK.

- A tuple in EMP_LOCS means that the employee whose name is Ename works on *some project* whose location is Plocation
- A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation.

EMP_LOCS	
Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaysa, Alicia J.	Stafford
Jabber, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1				
San	Prumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987087987	10	35.0	Computerization	Stafford
987087987	30	5.0	Newbenefits	Stafford
987054321	30	20.0	Newbenefits	Stafford
987054321	20	15.0	Reorganization	Houston
888065555	20	NULL	Reorganization	Houston

- 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 the original EMP_PROJ relation are called spurious tuples because they represent situations that are not valid.
- The spurious tuples are marked by a question mark

San	Prumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
123456789	2	7.5	ProductY	Sugarland
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
333445555	20	10.0	Reorganization	Houston

- 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.

Guideline 4

- Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated
- Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

4.4 Functional Dependencies

- Formal tool for analysis of relational schemas that enables us to detect and describe some of the problems in precise terms

Definition of Functional Dependency

- A functional dependency is a constraint between two sets of attributes from the database.

ther
n at
the

values of the attributes in X, one can determine the corresponding value of the Y attribute.

- The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the left-hand side of the FD, and Y is called the right-hand side.
- A functional dependency is a property of the semantics or meaning of the attributes.
- The database designers will use their understanding of the semantics of the attributes of R to specify the functional dependencies that should hold on all relation states (extensions) r of R.
- Consider the relation schema EMP_PROJ;

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
-----	---------	-------	-------	-------	-----------

- From the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

- a. $\text{Ssn} \rightarrow \text{Ename}$
- b. $\text{Pnumber} \rightarrow \{\text{Pname}, \text{Plocation}\}$
- c. $\{\text{Ssn}, \text{Pnumber}\} \rightarrow \text{Hours}$
- These functional dependencies specify that
 - (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename)
 - (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and
 - (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).
- Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename*, and so on.
- Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states (or legal extensions)** of R
- A functional dependency is a property of the relation schema R , not of a particular legal relation state r of R
- Therefore, an FD cannot be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R

Diagrammatic notation for displaying FDs

- Each FD is displayed as a horizontal line
- The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD
- The right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

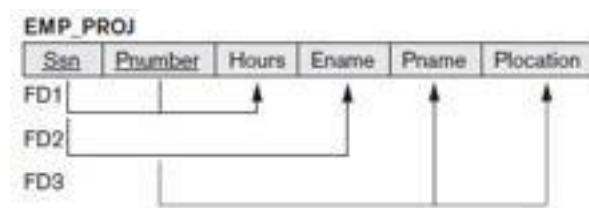


Fig: diagrammatic notation for displaying FDs

Example:

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

- The following FDs *may hold* because the four tuples in the current extension have no violation of these constraints:

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

- The following *do not hold* because we already have violations of them in the given extension:

- [redacted]
- [redacted]
- [redacted]

Normal Forms Based on Primary Keys

We assume that a

- Set of functional dependencies is given for each relation
- Each relation has a designated primary key
- This information combined with the tests (conditions) for normal forms drives the normalization process for relational schema design
- First three normal forms for relation takes into account all candidate keys of a relation rather than the primary key

4.4.1 Normalization of Relations

- The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**.
- Initially, Codd proposed three normal forms, which he called first, second, and third normal form
- All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation
- A fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively
- **Normalization of data** can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of
 - (1) minimizing redundancy and
 - (2) minimizing the insertion, deletion, and update anomalies

- It can be considered as a “filtering” or “purification” process to make the design have successively better quality
- Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties.
- Thus, the normalization procedure provides database designers with the following:
 - A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
 - A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be normalized to any desired degree
- **Definition:** The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized

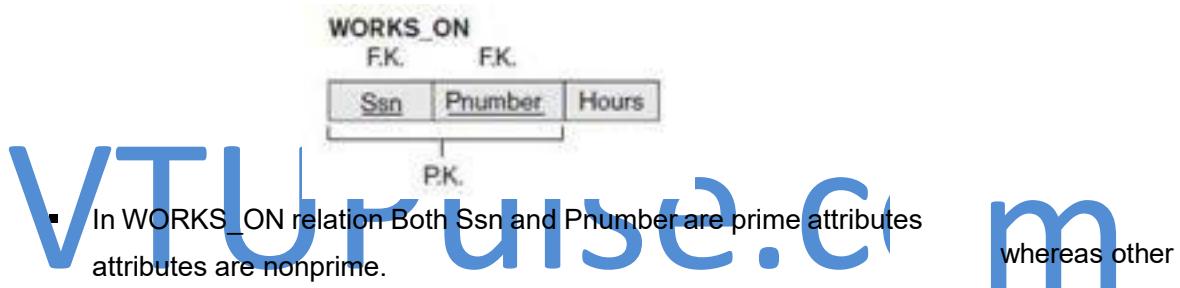
4.4.2 Practical Use of Normal Forms

- Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- Database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.
- The database designers *need not* normalize to the highest possible normal form
- Relations may be left in a lower normalization status, such as 2NF, for performance reasons
- **Definition: Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

4.4.3 Definitions of Keys and Attributes Participating in Keys

- **Superkey:** specifies a uniqueness constraint that no two distinct tuples in any state of R can have the same value
- **key K** is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more
- Example:
 - The attribute set {Ssn} is a key because no two employees tuples can have the same value for Ssn

- Any set of attributes that includes Ssn—for example, {Ssn, Name, Address}—is a superkey
- If a relation schema has more than one key, each is called a **candidate key**
- One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called **secondary keys**
- In a practical relational database, each relation schema must have a primary key
- If no candidate key is known for a relation, the entire relation can be treated as a default superkey
- For example {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key
- **Definition.** 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 WORKS_ON relation Both Ssn and Pnumber are prime attributes whereas other attributes are nonprime.

4.4.4 First Normal Form

- Defined to disallow multivalued attributes, composite attributes, and their combinations
- It states that the domain of an attribute must include **only atomic** (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute
- 1NF disallows relations within relations or relations as attribute values within tuples
- The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) values.
- Consider the DEPARTMENT relation schema shown in Figure below

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

↑ | ↑ ↑

- Primary key is Dnumber
- We assume that each department can have a number of locations

- The DEPARTMENT schema and a sample relation state are shown in Figure below

DEPARTMENT

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

- As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure
- 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→Dlocations because each set is considered a single member of the attribute domain
- In either case, the DEPARTMENT relation is not in 1NF

There are three main ways to decompose a non-1NF relation into 1NF relations.

1.

DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation}. 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. In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing redundancy in the relation

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

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. T

Querying on this attribute becomes more difficult; for example, consider how you would write the query: List the departments that have 'Bellaire' as one of their locations in this design.

- Of the three solutions, 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
- First normal form also disallows multivalued attributes that are themselves composite.
- These are called **nested relations** because each tuple can have a relation within it.

(a)		EMP_PROJ		Proj	
Ssn	Ename	Pnumber	Hours		

- Figure above shows how the EMP_PROJ relation could appear if nesting is allowed
- Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project.
- The schema of this EMP_PROJ relation can be represented as follows:
 $\text{EMP_PROJ}(\text{Ssn}, \text{Ename}, \{\text{PROJS}(\text{Pnumber}, \text{Hours})\})$
- Ssn is the primary key of the EMP_PROJ relation and Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber
- To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation
- Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2,

EMP_PROJ1	
Ssn	Ename

EMP_PROJ2		
Ssn	Pnumber	Hours

EMP_PROJ

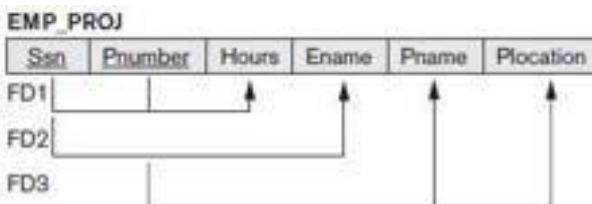
Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

any

any

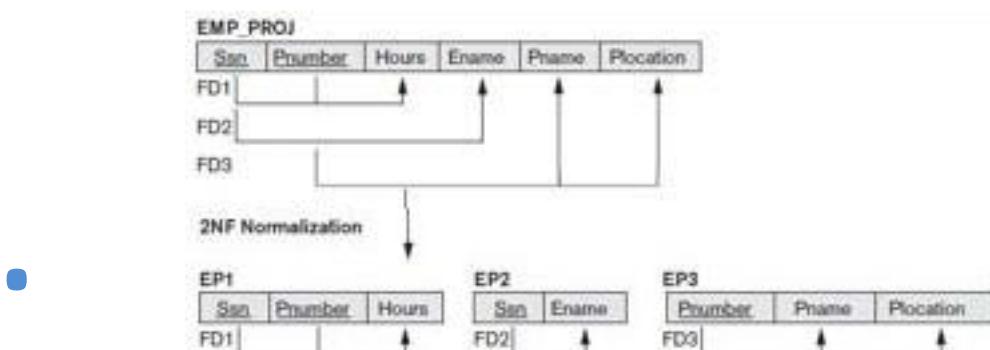
attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y

- if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$



- In the above figure, $\{Ssn, Pnumber\} \rightarrow Hours$ is a full dependency (neither $Ssn \rightarrow Hours$ nor $Pnumber \rightarrow Hours$ holds)
- $\{Ssn, Pnumber\} \rightarrow Ename$ is partial because $Ssn \rightarrow Ename$ 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
- 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 is in 1NF but is not in 2NF.
- The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3
- The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.
- If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which **nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent**.
- Therefore, the functional dependencies FD1, FD2, and FD3 lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure below, each of which is in 2NF.

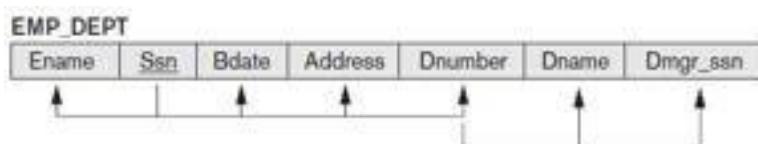


4.4.6 Third Normal Form

- Transitive functional dependency**

there exists a set of attribute Z that are neither a primary nor a subset of any key of R(candidate key) and both X \rightarrow Z and Y \rightarrow Z holds

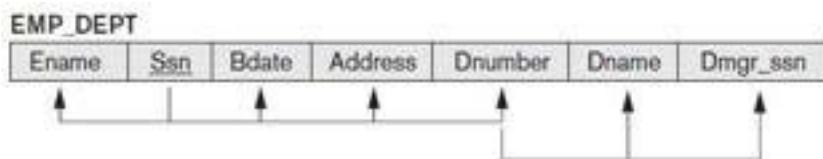
- Example:**



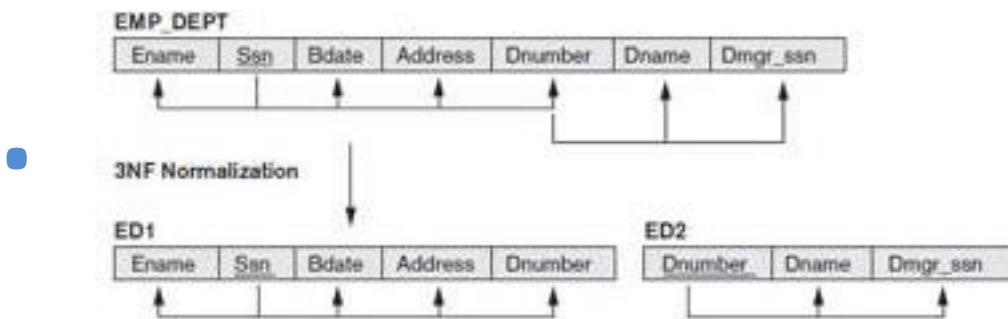
- SSN \rightarrow DMGRSSN** is a transitive FD since SSN \rightarrow DNUMBER and DNUMBER \rightarrow DMGRSSN hold

Dnumber is neither a key itself nor a subset of the key of EMP_DEPT

- **SSN → ENAME is non-transitive** since there is no set of attributes X where $\text{SSN} \rightarrow X$ and $X \rightarrow \text{ENAME}$
- **Definition: A relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key**
- The relation schema EMP_DEPT is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber



- We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2



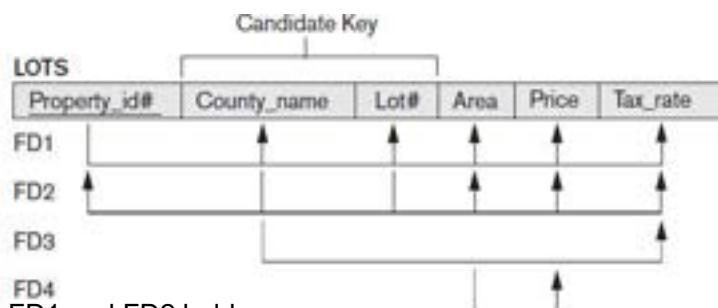
- ED1 and ED2 represent independent entity facts about employees and departments
- A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples
- Problematic FD
 - Left-hand side is part of primary key
 - Left-hand side is a non-key attribute
- 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations
- In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

4.5 General Definition of Second and Third Normal Form

- Takes into account all candidate keys of a relation into account
- **Definition of 2NF:** A relation schema R is in **second normal form (2NF)** if every nonprime attribute A in R is not partially dependent on any key of R
- Consider the relation schema LOTS which describes parcels of land for sale in various counties of a state
- Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.



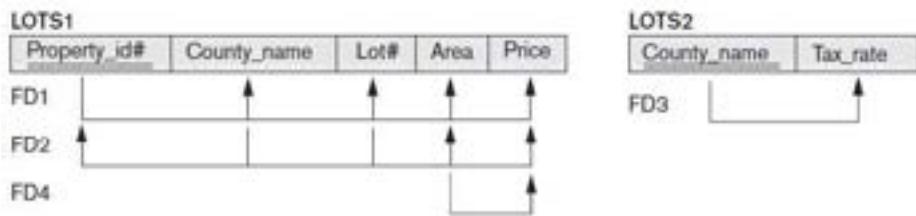
dependencies FD1 and FD2 hold

- FD1: Property_id → {County_name, Lot#, Area, Price, Tax_rate}
- FD2: {County_name, Lot#} → {Property_id#, Area, Price, Tax_rate}
- FD3: County_name → Tax_rate
- FD4: Area → Price

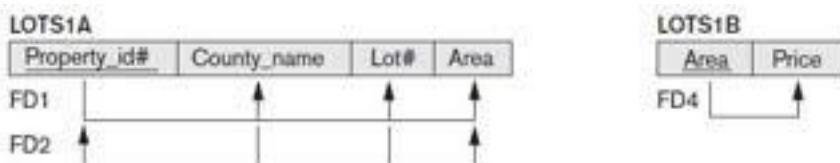
We choose Property_id# as the primary key, but no special consideration will be given to this key over the other candidate key

FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county)

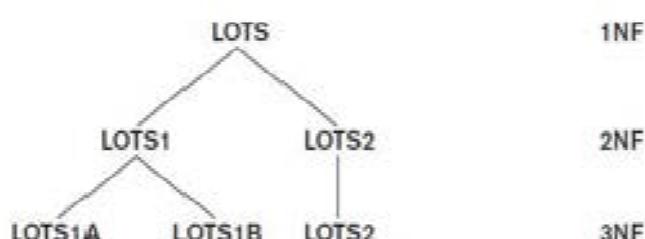
- FD4 says that the price of a lot is determined by its area regardless of which county it is in.
- The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3
- To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2
-
-



- We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2.
- Both LOTS1 and LOTS2 are in 2NF.
- Definition of 3NF:** A relation schema R is in **third normal form (3NF)** if, whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R
- According to this definition, LOTS2 is in 3NF
- FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1
- To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B

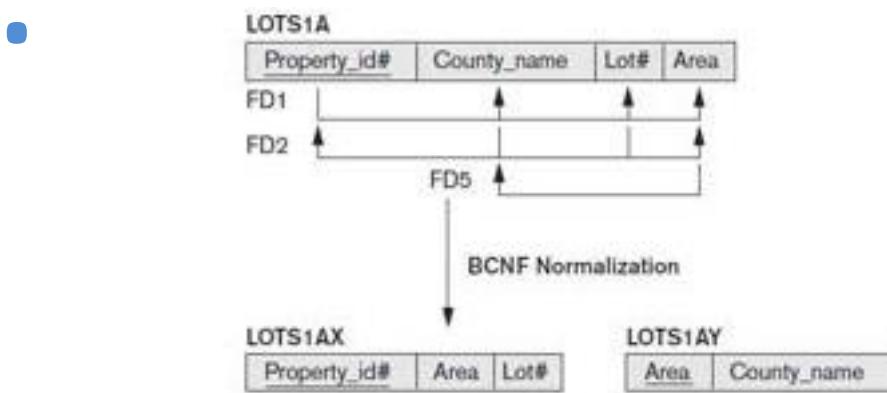


- We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the lefthand side of FD4 that causes the transitive dependency) into another relation LOTS1B.
- Both LOTS1A and LOTS1B are in 3NF



4.6 Boyce-Codd Normal Form

- **Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF
- Every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF
- **Definition.** A relation schema R is in **BCNF** if whenever a nontrivial functional dependency $X \rightarrow A$ holds in R , then X is a superkey of R
- The formal definition of BCNF is the condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF
- In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A
- FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF
- We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.



- In practice, most relation schemas that are in 3NF are also in BCNF
- Only if $X \rightarrow A$ holds in a relation schema R with X not being a superkey and A being a prime attribute will R be in 3NF but not in BCNF
- Example: consider the relation TEACH with the following dependencies:

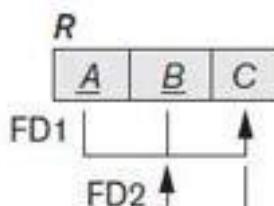
TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

FD1: {Student, Course} → Instructor

FD2: Instructor → Course -- means that each instructor teaches one course

- {Student, Course} is a candidate key for this relation
- The dependencies shown follow the pattern in Figure below with Student as A, Course as B, and Instructor as C



- Hence this relation is in 3NF but not BCNF
- Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:
 1. $R1(\underline{\text{Student}}, \underline{\text{Instructor}})$ and $R2(\underline{\text{Student}}, \underline{\text{Course}})$
 2. $R1(\underline{\text{Course}}, \underline{\text{Instructor}})$ and $R2(\underline{\text{Course}}, \underline{\text{Student}})$
 3. $R1(\underline{\text{Instructor}}, \underline{\text{Course}})$ and $R2(\underline{\text{Instructor}}, \underline{\text{Student}})$
- It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF
- Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:
 - The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
 - The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

- We are not able to meet the functional dependency preservation ,but we must meet the non additive join property
- **Nonadditive Join Test for Binary Decomposition:**
A decomposition $D=\{R_1, R_2\}$ of R has the lossless join property with respect to a set of functional dependencies F on R if and only if either
 - The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+ or
 - The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+
- The third decomposition meets the test

$R_1 \cap R_2$ is Instructor

$R_1 - R_2$ is Course
- Hence, the proper decomposition of TEACH into BCNF relations is:
TEACH1(Instructor, Course) and TEACH2(Instructor, Student)
- In general, a relation R not in BCNF can be decomposed so as to meet the nonadditive join property by the following procedure. It decomposes R successively into set of relations that are in BCNF:
Let R be the relation not in BCNF, let $X \subseteq R$, and let $X \rightarrow A$ be the FD that causes violation of BCNF. R may be decomposed into two relations:

$R - A$

XA

If either $R - A$ or XA is not in BCNF, repeat the process

4.7 Multivalued Dependency and Fourth Normal Form

- For example, consider the relation EMP shown in Figure below:

EMP		
<u>Ename</u>	<u>Pname</u>	<u>Dname</u>
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

- A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname
- An employee may work on several projects and may have several dependents
- The employee's projects and dependents are independent of one another

- To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project
- In the relation state shown in the EMP, the employee Smith works on two projects 'X' and 'Y' and has two dependents 'John' and 'Anna' and therefore there are 4 tuples to represent these facts together
- The relation EMP is an **all-key relation** (with key made up of all attributes) and therefore no f.d's and as such qualifies to be a BCNF relation
- There is a redundancy in the relation EMP-the dependent information is repeated for every project and project information is repeated for every dependent
- To address this situation, the concept of multivalued dependency(MVD) was proposed and based on this dependency, the fourth normal form was defined
- **Multivalued dependencies** are a consequence of 1NF which disallows an attribute in a tuple to have a set of values, and the accompanying process of converting an unnormalized relation into 1NF
- Informally, whenever two independent 1:N relationships are mixed in the same

A multivalued dependency $X \rightarrow\!\! \rightarrow Y$ specified on relation schema R, where X and Y are attributes of R, and R(A, B, C), an MVD may arise.

t1 and t2 exist in r such that $t1[X] = t2[X]$, then two tuples t3 and t4 should also exist in r with the following properties where we use Z to denote $(R - (X \cup Y))$

- $t3[X] = t4[X] = t1[X] = t2[X]$.
- $t3[Y] = t1[Y] \text{ and } t4[Y] = t2[Y]$.
- $t3[Z] = t2[Z] \text{ and } t4[Z] = t1[Z]$.

EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

Let X= Ename, Y=Pname

$t1[Ename]=t2[Ename]=Smith$

$$\begin{aligned} Z &= (EMP - (Ename \cup Pname)) \\ &= Dname \end{aligned}$$

- $t3(Ename)=t4(Ename)=t1(Ename)=t2(Ename)=Smith$

- $t_3(Pname) = t_1(Pname) = X$ and $t_4(Pname) = t_2(Pname) = Y$
- $t_3(Dname) = t_2(Dname) = Anna$ and $t_4(Dname) = t_1(Dname) = John$
- Whenever $X \rightarrow\!\!\rightarrow Y$ holds, we say that **X multidetermines Y** . Because of the symmetry in the definition, whenever $X \rightarrow\!\!\rightarrow Y$ holds in R , so does $X \rightarrow\!\!\rightarrow Z$. Hence, $X \rightarrow\!\!\rightarrow Y$ implies $X \rightarrow\!\!\rightarrow Z$, and therefore it is sometimes written as $X \rightarrow\!\!\rightarrow Y \mid Z \rightarrow\!\!\rightarrow$
- An MVD $X \rightarrow\!\!\rightarrow Y$ in R is called a **trivial MVD** if
 - Y is a subset of X , or
 - $X \cup Y = R$

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

- For example, the relation EMP_PROJECTS has the trivial MVD
 $Ename \rightarrow\!\!\rightarrow Pname$
- An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**
- If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples
- In the EMP relation the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname)
- This redundancy is clearly undesirable.
- We now present the definition of **fourth normal form (4NF)**, which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations
- Definition:** A relation schema R is in **4NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every
 - The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

- We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS
- Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs $Ename \rightarrow\!\!\rightarrow Pname$ in EMP_PROJECTS and $Ename \rightarrow\!\!\rightarrow Dname$ in EMP_DEPENDENTS are trivial MVDs
- No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either

- We can state the following points:
 - An all-key relation is always in BCNF since it has no FDs
 - An all-key relation such as the EMP, which has no FDs but has the MVD $Ename \rightarrow\!\!\rightarrow Pname \mid Dname$, is not in 4NF
 - A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF
 - The decomposition removes the redundancy caused by the MVD

4.8 Join Dependencies and Fifth Normal Form

- ▪ A **join dependency (JD)**, denoted by $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , specifies a constraint on the states r of R . The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, \dots, R_n . Hence, for every such r we have

$$*(\pi_{R_1}(r), \pi_{R_2}(r), \dots, \pi_{R_n}(r)) = r$$

- A join dependency $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , is a **trivial JD** if one of the relation schemas R_i in $JD(R_1, R_2, \dots, R_n)$ is equal to R .

Fifth normal form (project-join normal form)

- A relation schema R is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency $JD(R_1, R_2, \dots, R_n)$ in F^+ every R_i is a superkey of R .
- A database is said to be in 5NF, if and only if,
 - It's in 4NF

- If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.

SUPPLY

<u>Sname</u>	<u>Part_name</u>	<u>Proj_name</u>
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

Fig: The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3) R_1

<u>Sname</u>	<u>Part_name</u>
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

 R_2

<u>Sname</u>	<u>Proj_name</u>
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

 R_3

<u>Part_name</u>	<u>Proj_name</u>
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Fig: Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3 .

Chapter 2: Normalization Algorithms

4.9 Inference Rules for Functional Dependencies

- Let F be the set of functional dependencies that are specified on relation schema R
- The schema designer specifies the functional dependencies that are semantically obvious
- Numerous other functional dependencies hold in all legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F
- Those other dependencies can be inferred or deduced from the FDs in F .
- **For example:**
 - If each department has one manager, so that Dept_no uniquely determines Mgr_ssn ($\text{Dept_no} \rightarrow \text{Mgr_ssn}$), and a manager has a unique phone number called Mgr_phone ($\text{Mgr_ssn} \rightarrow \text{Mgr_phone}$),
 - Then these two dependencies together imply that

$$\text{Dept_no} \rightarrow \text{Mgr_phone}$$
 - This is an **inferred FD** and need *not* be explicitly stated in addition to the two given FDs.

- **Definition.** Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F ; it is denoted by F^+ .
- For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema EMP_DEPT

EMP_DEPT						
ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN

- $F = \{$
 - $\text{SSN} \rightarrow \{\text{Ename}, \text{Bdate}, \text{Address}, \text{Dnumber}\}$,
 - $\text{Dnumber} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}$
- Some of the additional functional dependencies that we can *infer* from F are the following:
 - $\text{SSN} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}$
 - $\text{SSN} \rightarrow \text{SSN}$

- $Dnumber \rightarrow Dname$
- An FD $X \rightarrow Y$ is **inferred from** a set of dependencies F specified on R if $X \rightarrow Y$ holds in every legal relation state r of R
- The closure F^+ of F is the set of all functional dependencies that can be inferred from F
- Set of **inference rules** can be used to infer new dependencies from a given set of dependencies
- We use the notation $F \models X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies F
- we use an abbreviated notation when discussing functional dependencies. We concatenate attribute variables and drop the commas for convenience
- The FD $\{X, Y\} \rightarrow Z$ is abbreviated to $XY \rightarrow Z$, and the FD $\{X, Y, Z\} \rightarrow \{U, V\}$ is abbreviated to $XYZ \rightarrow UV$.
- Three rules IR1 through IR3 are well-known inference rules for functional dependencies.
- They are proposed by Armstrong and hence known as **Armstrong's axioms**
 - IR1 (reflexive rule): If $X \supseteq Y$, then $X \rightarrow Y$.
 - IR2 (augmentation rule): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
 - IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
- The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious.
- Because IR1 generates dependencies that are always true, such dependencies are called **trivial**.
- Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**.
- The augmentation rule (IR2) says that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency
- According to IR3, functional dependencies are transitive
- There are three other inference rules that follow from IR1, IR2 and IR3. They are:
 - IR4 (decomposition, or projective, rule): $\{X \rightarrow YZ\} \models X \rightarrow Y$
 - IR5 (union, or additive, rule): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$
 - IR6 (pseudotransitive rule): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$
- The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD $X \rightarrow \{A_1, A_2, \dots, A_n\}$ into the set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$.
- The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$ into the single FD $X \rightarrow \{A_1, A_2, \dots, A_n\}$.
- The pseudotransitive rule (IR6) allows us to replace a set of attributes Y on the left hand side of a dependency with another set X that functionally determines Y , and can be

derived from IR2 and IR3 if we augment the first functional dependency $X \rightarrow Y$ with W (the augmentation rule) and then apply the transitive rule.

- In other words, the set of dependencies F^+ , which we called the **closure** of F , can be determined from F by using only inference rules IR1 through IR3.
- A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X .
- **Definition.** For each such set of attributes X , we determine the set X^+ of attributes that are functionally determined by X based on F ; X^+ is called the **closure of X under F** .
- Algorithm 16.1 can be used to calculate X^+ .

Algorithm 16.1. Determining X^+ , the Closure of X under F

Input: A set F of FDs on a relation schema R , and a set of attributes X , which is a subset of R .

```

 $X^+ := X;$ 
repeat
    old $X^+ := X^+;$ 
    for each functional dependency  $Y \rightarrow Z$  in  $F$  do
        if  $X^+ \supseteq Y$  then  $X^+ := X^+ \cup Z$ ;
until ( $X^+ = \text{old}X^+$ );

```

- Algorithm 16.1 starts by setting X^+ to all the attributes in X .
- By IR1, we know that all these attributes are functionally dependent on X .
- Using inference rules IR3 and IR4, we add attributes to X^+ , using each functional dependency in F .
- We keep going through all the dependencies in F (the repeat loop) until no more attributes are added to X^+ during a complete cycle (of the for loop) through the dependencies in F .
- For example, consider the relation schema EMP_PRO. From the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PRO:

$$\begin{aligned}
F = & \{ Ssn \rightarrow Ename, \\
& Pnumber \rightarrow \{Pname, Plocation\}, \\
& \{Ssn, Pnumber\} \rightarrow Hours \}
\end{aligned}$$

- Using Algorithm 16.1, we calculate the following closure sets with respect to F :

- $\{Ssn\}^+ = \{Ssn, Ename\}$

- $\{Pnumber\}^+ = \{Pnumber, Pname, Plocation\}$
- $\{Ssn, Pnumber\}^+ = \{Ssn, Pnumber, Ename, Pname, Plocation, Hours\}$

4.10 Equivalence of Sets of Functional Dependencies

Definition: A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+ ; that is, if every dependency in E can be inferred from F ; alternatively, we can say that E is **covered by F** .

Definition: Two sets of functional dependencies E and F are **equivalent** if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F , and every FD in F can be inferred from E ; that is, E is equivalent to F if both the conditions— E covers F and F covers E —hold.

4.11 Sets of Functional Dependencies

A set of functional dependencies F to be **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y is a proper subset of X , and still have a set of dependencies that is equivalent to F .
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F .

Algorithm 16.2. Finding a Minimal Cover F for a Set of Functional Dependencies E

Input: A set of functional dependencies E .

1. Set $F := E$.
2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \dots, A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$.
3. For each functional dependency $X \rightarrow A$ in F
 - for each attribute B that is an element of X
 - if $\{F - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}$ is equivalent to F
 - then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F .
4. For each remaining functional dependency $X \rightarrow A$ in F
 - if $\{F - \{X \rightarrow A\}\}$ is equivalent to F ,
 - then remove $X \rightarrow A$ from F .

- Step 2 places FDs in a canonical form for subsequent testing
- Step 3 constitutes removal of an extraneous attribute B contained in the left-hand side X of a functional dependency $X \rightarrow A$ from F when possible
- Step 4 constitutes removal of a redundant functional dependency $x \rightarrow A$ from F when possible
- Example 1: Let the given set of FDs be $E : \{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$. We have to find the minimal cover of E .
 - All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm and can proceed to step 2
 - In step 2 we need to determine if $AB \rightarrow D$ has any redundant attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?
 - Since $B \rightarrow A$, by augmenting with B on both sides (IR2), we have $BB \rightarrow AB$, or $B \rightarrow AB$ (i). However, $AB \rightarrow D$ as given (ii).
 - Hence by the transitive rule (IR3), we get from (i) and (ii), $B \rightarrow D$. Thus $AB \rightarrow D$ may be replaced by $B \rightarrow D$.
 - We now have a set equivalent to original E , say $E' : \{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.
 - In step 3 we look for a redundant FD in E' . By using the transitive rule on $B \rightarrow D$ and $D \rightarrow A$, we derive $B \rightarrow A$. Hence $B \rightarrow A$ is redundant in E' and can be eliminated.

Algorithm 16.2(a). Finding a Key K for R Given a set F of Functional Dependencies

Input: A relation R and a set of functional dependencies F on the attributes of R .

1. Set $K := R$,
2. For each attribute A in K
 - |compute $(K - A)^+$ with respect to F ;
 - |if $(K - A)^+$ contains all the attributes in R , then set $K := K - \{A\}$ |;

- Therefore, the minimal cover of E is $\{B \rightarrow D, D \rightarrow A\}$.

We start by setting K to all the attributes of R ; we then remove one attribute at a time and check whether the remaining attributes still form a superkey.

- Algorithm 16.2(a) determines only *one* key out of the possible candidate keys for R ; the key returned depends on the order in which attributes are removed from R in step 2.

4.12 Properties of Relational Decompositions

Universal relation schema

- **Universal relation schema** $R = \{A_1, A_2, \dots, A_n\}$ includes *all* the attributes of the database
- **universal relation assumption:** every attribute name is unique
- The set F of functional dependencies that should hold on the attributes of R is specified by the database designers
- Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, \dots, R_m\}$ that will become the relational database schema ; D is called a **decomposition** of R .

Attribute Preservation condition of a Decomposition

- Each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are *lost*; formally, we have

$$\bigcup_{i=1}^m R_i = R$$

- Another goal of decomposition is to have each individual relation R_i in the decomposition D be in BCNF or 3NF
- Additional properties of decomposition are needed to prevent from generating spurious tuples

Desirable Properties of Decompositions

- Not all decomposition of a schema are useful
- We require two properties to be satisfied:
 - i) Dependency Preservation Property
 - ii) Nonadditive (Lossless) Join Property

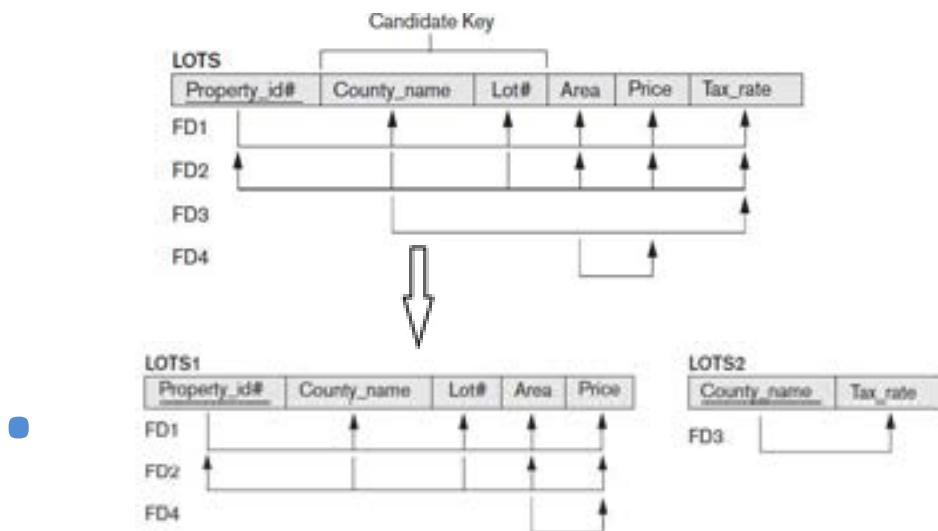
Dependency Preservation Property

- relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i

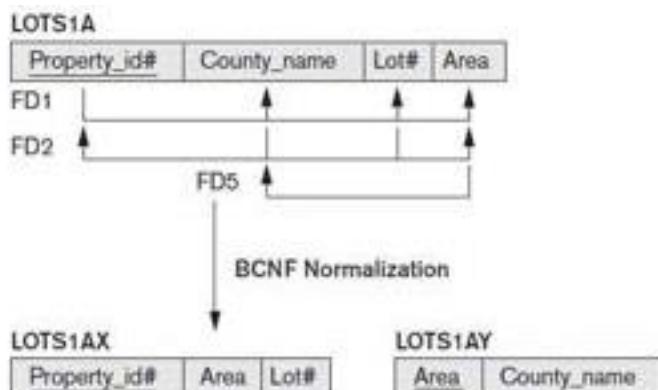
We want to preserve the dependencies because each dependency in F represents a constraint on the database

If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation

- It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D .
- It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F
- **Example:** Dependency Preserving Decomposition



- **Example:** Decomposition that does not Preserve Dependency



- The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations
- The term **lossy design** refer to a design that represents a loss of information
- If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information

Algorithm 16.3. Testing for Nonadditive Join Property

Input: A universal relation R , a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R , and a set F of functional dependencies.

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (* *comment**).

1. Create an initial matrix S with one row i for each relation R_i in D , and one column j for each attribute A_j in R .
2. Set $S(i, j) := b_{ij}$ for all matrix entries. (* each b_{ij} is a distinct symbol associated with indices (i, j) *).
3. For each row i representing relation schema R_i
 - {for each column j representing attribute A_j
 - {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};}; (* each a_j is a distinct symbol associated with index (j) *).

4. Repeat the following loop until a *complete loop execution* results in no changes to S
 - {for each functional dependency $X \rightarrow Y$ in F
 - {for all rows in S that have the same symbols in the columns corresponding to attributes in X
 - {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that *same a* symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column ;} ;} ;};
 5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

Example

- (a) $R = \{\text{Ssn, Ename, Pnumber, Phname, Plocation, Hours}\}$ $D = \{R_1, R_2\}$
 $R_1 = \text{EMP_LOCS} = \{\text{Ename, Plocation}\}$
 $R_2 = \text{EMP_PROJ1} = \{\text{Ssn, Pnumber, Hours, Phname, Plocation}\}$

$F = \{\text{Ssn} \rightarrow\!\!-\!\!> \text{Ename}; \text{Pnumber} \rightarrow\!\!-\!\!> \{\text{Phname, Plocation}\}; (\text{Ssn}, \text{Pnumber}) \rightarrow\!\!-\!\!> \text{Hours}\}$

	Slno	Ename	Pnumber	Pname	Plocation	Hours
R ₁	b ₁₁	a ₂	b ₁₃	b ₁₄	a ₅	b ₁₆
R ₂	a ₁	b ₂₂	a ₃	a ₄	a ₅	a ₆

(No changes to matrix after applying functional dependencies)

- (c) $R = \{\text{Ssn, Ename, Pnumber, Pname, Plocation, Hours}\}$ $D = \{R_1, R_2, R_3\}$
 $R_1 = \text{EMP} = \{\text{Ssn, Ename}\}$
 $R_2 = \text{PROJ} = \{\text{Pnumber, Pname, Plocation}\}$
 $R_3 = \text{WORKS_ON} = \{\text{Ssn, Pnumber, Hours}\}$

$$F = \{ \text{Ssn} \rightarrow \text{Enrname}; \text{Phumber} \rightarrow (\text{Pname}, \text{Plocation}); (\text{Ssn}, \text{Phumber}) \rightarrow \text{Hours} \}$$

	San	Ename	Pnumber	Pname	Plocation	Hours
R ₁	a ₁	a ₂	b ₁₃	b ₁₄	b ₁₅	b ₁₆
R ₂	b ₂₁	b ₂₂	a ₃	a ₄	a ₅	b ₂₆
R ₃	a ₆	b ₂₃	a ₇	b ₂₄	b ₂₅	a ₈

(Original matrix S at start of algorithm)

	Stn	Ename	Pnumber	Pname	Plocation	Hours
R ₁	a ₁	a ₂	b ₁₃	b ₁₄	b ₁₅	b ₁₆
R ₂	b ₂₁	b ₂₂	a ₃	a ₄	a ₅	b ₂₆
R ₃	a ₁	b ₂₀ , a ₂	a ₃	b ₂₁ , a ₄	b ₂₂ , a ₅	a ₆

(Matrix S after applying the first two functional dependencies; last row is all “ a ” symbols so we stop)

Testing Binary Decompositions for the Nonadditive Join Property

Property NJB (Nonadditive Join Test for Binary Decompositions). A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R if and only if either

- The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+ , or
- The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+

4.13 Algorithms for Relational Database Schema Design

Two algorithms for creating a relational decomposition from universal relation

1. The first algorithm decomposes a universal relation into dependency preserving 3NF relations that also possess the nonadditive join property
2. The second algorithm decomposes a universal relation schema into BCNF schemas that possess the nonadditive join property

4.13.1 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

Algorithm 16.4. Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

- **Input:** A universal relation R and a set of functional dependencies F on the attributes of R .
1. Find a minimal cover G for F (use Algorithm 16.2).
 2. For each left-hand-side X of a functional dependency that appears in G , create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation)
 - 5 If none of the relation schemas in D contains a key of R , then create one more relation schema in D that contains attributes that form a key of R
 - 6 Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S

- **Example:** Consider the following universal relation:

$U(\text{Emp_ssn}, \text{Pno}, \text{Esal}, \text{Ephone}, \text{Dno}, \text{Pname}, \text{Plocation})$

- Emp_ssn, Esal, Ephone refer to the Social Security number, salary, and phone number of the employee. Pno, Pname, and Plocation refer to the number, name, and location of the project. Dno is department number.
- The following dependencies are present:
 - FD1: $\text{Emp_ssn} \rightarrow \{\text{Esal}, \text{Ephone}, \text{Dno}\}$
 - FD2: $\text{Pno} \rightarrow \{\text{Pname}, \text{Plocation}\}$
 - FD3: $\text{Emp_ssn}, \text{Pno} \rightarrow \{\text{Esal}, \text{Ephone}, \text{Dno}, \text{Pname}, \text{Plocation}\}$
- By virtue of FD3, the attribute set $\{\text{Emp_ssn}, \text{Pno}\}$ represents a key of the universal relation.
- Hence F , the set of given FDs includes $\{\text{Emp_ssn} \rightarrow \text{Esal}, \text{Ephone}, \text{Dno}; \text{Pno} \rightarrow \text{Pname}, \text{Plocation}; \text{Emp_ssn}, \text{Pno} \rightarrow \text{Esal}, \text{Ephone}, \text{Dno}, \text{Pname}, \text{Plocation}\}$.
- By applying the minimal cover , in step 3 we see that Pno is a redundant attribute in $\text{Emp_ssn}, \text{Pno} \rightarrow \text{Esal}, \text{Ephone}, \text{Dno}$. Moreover, Emp_ssn is redundant in $\text{Emp_ssn}, \text{Pno} \rightarrow \text{Pname}, \text{Plocation}$.
- Hence the minimal cover consists of FD1 and FD2 only
- Minimal cover G: $\{\text{Emp_ssn} \rightarrow \text{Esal}, \text{Ephone}, \text{Dno}; \text{Pno} \rightarrow \text{Pname}, \text{Plocation}\}$
- By applying Algorithm 16.4 to the above Minimal cover G, we get a 3NF design consisting of two relations with keys Emp_ssn and Pno as follows:

- - $R1(\text{Emp_ssn}, \text{Esal}, \text{Ephone}, \text{Dno})$
 - $R2(\text{Pno}, \text{Pname}, \text{Plocation})$

- In step 3, we generate a relation corresponding to the key($\text{Emp_ssn}, \text{Pno}$) of U. Hence, the resulting design contains:

$R1(\text{Emp_ssn}, \text{Esal}, \text{Ephone}, \text{Dno})$
 $R2(\text{Pno}, \text{Pname}, \text{Plocation})$
 $R3(\text{Emp_ssn}, \text{Pno})$

This design achieves both the desirable properties of dependency preservation and non additive join

4.13.2 Nonadditive Join Decomposition into BCNF Schemas

Algorithm 16.5. Relational Decomposition into BCNF with Nonadditive

Join Property

.Input: A universal relation R and a set of functional dependencies F on the attributes of R .

1. Set $D := \{R\}$;
2. While there is a relation schema Q in D that is not in BCNF do
 - {
 - choose a relation schema Q in D that is not in BCNF;

find a functional dependency $X \rightarrow Y$ in Q that violates BCNF;

replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;

} ;

- Each time through the loop in Algorithm 16.5, we decompose one relation schema Q that is not in BCNF into two relation schemas.
- According to Property NJB for binary decompositions and Claim 2, the decomposition D has the nonadditive join property
- At the end of the algorithm, all relation schemas in D will be in BCNF
- Example: TEACH relation schema decomposed into TEACH1(Instructor, Student) and TEACH2(Instructor, Course) because the dependency FD2 Instructor \rightarrow Course violates BCNF.
- In step 2 of Algorithm 16.5, it is necessary to determine whether a relation schema Q is in BCNF or not.
- whenever a relation schema Q has a BCNF violation, there exists a pair of attributes A and B in Q such that $\{Q \{A, B\} A \rightarrow\}$ by computing the closure $\{Q \{A, B\}\}^+$ for each pair of attributes $\{A, B\}$ of Q , and checking whether the closure includes A (or B), we can determine whether Q is in BCNF.

4.13.3 Dependency-Preserving and Nonadditive (Lossless) Join

● Decomposition into 3NF Schemas

- It is *not possible to have all three of the following:*
 - (1) guaranteed nonlossy design,
 - (2) guaranteed dependency preservation, and
 - (3) all relations in BCNF
- The first condition is a must and cannot be compromised.
- The second condition is desirable, but not a must, and may have to be relaxed if we insist on achieving BCNF.
- Now we give an alternative algorithm where we achieve conditions 1 and 2 and only guarantee 3NF.
- A simple modification to Algorithm 16.4, shown as Algorithm 16.6, yields a decomposition D of R that does the following:
 - Preserves dependencies
 - Has the nonadditive join property

Is such that each resulting relation schema in the decomposition is in 3NF

Algorithm 16.6. Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R .

1. Find a minimal cover G for F (use Algorithm 16.2).
2. For each left-hand-side X of a functional dependency that appears in G , create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R , then create one more relation schema in D that contains attributes that form a key of R .⁷ (Algorithm 16.2(a) may be used to find a key.)
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S .

Step 3 involves identifying a key K of R . Algorithm 16.2(a) can be used to identify a key K of R based on the set of given functional dependencies F .

Example 1 of Algorithm 16.6. Let us revisit the example given earlier at the end of Algorithm 16.4. The minimal cover G holds as before. The second step produces relations R_1 and R_2 as before. However, now in step 3, we will generate a relation corresponding to the key $\{\text{Emp_ssn}, \text{Pno}\}$. Hence, the resulting design contains:

- $R_1 (\underline{\text{Emp_ssn}}, \text{Esal}, \text{Ephone}, \text{Dno})$
- $R_2 (\underline{\text{Pno}}, \text{Pname}, \text{Plocation})$
- $R_3 (\underline{\text{Emp_ssn}}, \underline{\text{Pno}})$

This design achieves both the desirable properties of dependency preservation and nonadditive join.

4.14 About Nulls, Dangling Tuples, and Alternative Relational Designs

4.14.1 Problems with NULL Values and Dangling Tuples

Whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be devoted to watching for potential NULL values in foreign keys.

This can cause unexpected loss of information in queries that involve joins on that foreign key.

- If NULLs occur in other attributes, such as Salary, their effect on built-in functions such as SUM and AVERAGE must be carefully evaluated.

Dangling tuples may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation in Figure 16.2(a) further into EMPLOYEE_1 and EMPLOYEE_2, shown in Figure 16.3(a) and 16.3(b)

(a)

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnum
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4

DEPARTMENT

Dname	Dnum	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

(b)

Ename	Ssn	Bdate	Address	Dnum	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 Fire Oak, 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

(c)

Ename	Ssn	Bdate	Address	Dnum	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 Fire Oak, 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
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL	NULL	NULL
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX	NULL	NULL	NULL

Figure 16.2: Issues with NULL-value joins. (a) Some EMPLOYEE tuples have NULL for the join attribute Dnum
 (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

(a) EMPLOYEE_1

Ename	Ssn	Bdate	Address
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX

(b) EMPLOYEE_2

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	NULL
888664444	NULL

(c) EMPLOYEE_3

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Figure 16.3: The dangling tuple problem. (a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from Figure 16.2(a) except Dnum). (b) The relation EMPLOYEE_2 (includes Dnum attribute with NULL values). (c) The relation EMPLOYEE_3 (includes Dnum attribute but does not include tuples for which Dnum has NULL values).

- If we apply the NATURAL JOIN operation to EMPLOYEE_1 and EMPLOYEE_2, we get the original EMPLOYEE relation.
- we may use the alternative representation, shown in Figure 16.3(c), where we *do not include a tuple in EMPLOYEE_3 if the employee has not been assigned a department (instead of including a tuple with NULL for Dnum as in EMPLOYEE_2)*.
- If we use EMPLOYEE_3 instead of EMPLOYEE_2 and apply a NATURAL JOIN on EMPLOYEE_1 and EMPLOYEE_3, the tuples for Berger and Benitez will not appear in the result; these are called **dangling tuples** in EMPLOYEE_1 because they are represented in only one of the two relations that represent employees, and hence are lost if we apply an (INNER) JOIN operation.

4.15 Other Dependencies and Normal Forms

4.15.1 Inclusion Dependencies

Inclusion dependencies were defined in order to formalize two types of interrelational constraints:

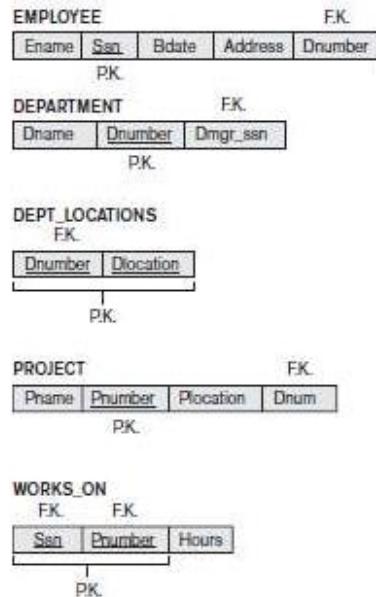
- The foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations.
- The constraint between two relations that represent a class/subclass relationship also has no formal definition in terms of the functional,multivalued, and join dependencies.

Definition. An **inclusion dependency** $R.X < S.Y$ between two sets of attributes— X of relation schema R , and Y of relation schema S —specifies the constraint that, at any specific time when r is a relation state of R and s a relation state of S , we must have

$$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

- The subset relationship does not necessarily have to be a proper subset. Obviously, the sets of attributes on which the inclusion dependency is specified— X of R and Y of S —must have the same number of attributes.
- In addition, the domains for each pair of corresponding attributes should be compatible.
- For example, we can specify the following inclusion dependencies on the relationalvschema in Figure 15.1:

Figure 15.1
A simplified COMPANY relational database schema.



- DEPARTMENT.Dmgr_ssn < EMPLOYEE.Ssn
- WORKS_ON.Ssn < EMPLOYEE.Ssn
- EMPLOYEE.Dnumber < DEPARTMENT.Dnumber
- PROJECT.Dnum < DEPARTMENT.Dnumber
- WORKS_ON.Pnumber < PROJECT.Pnumber
- DEPT_LOCATIONS.Dnumber < DEPARTMENT.Dnumber

- All the preceding inclusion dependencies represent **referential integrity constraints**.

- We can also use inclusion dependencies to represent **class/subclass**. For example, in the relational schema of Figure 9.6, we can specify the following inclusion dependencies:
 - EMPLOYEE.Ssn < PERSON.Ssn
 - ALUMNUS.Ssn < PERSON.Ssn
 - STUDENT.Ssn < PERSON.Ssn

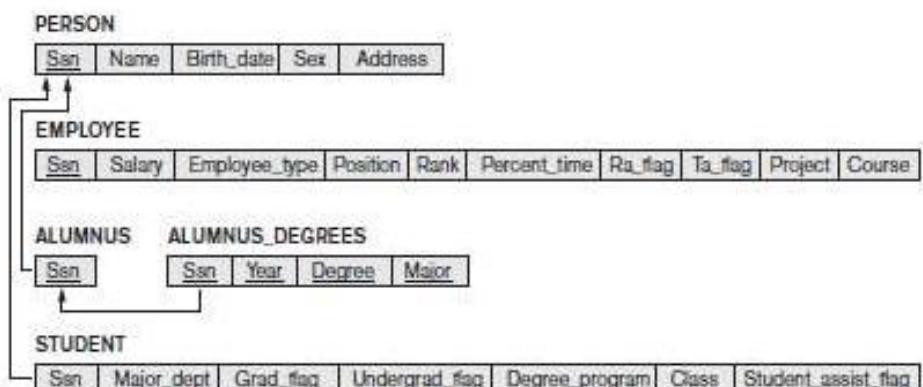


Figure 9.6
Mapping the EER specialization lattice in Figure 8.8 using multiple options.

4.15.2 Template Dependencies

- Template dependencies provide a technique for representing constraints in relations that typically have no easy and formal definitions.
- There are two types of templates:
 - tuple-generating templates and
 - constraint generating templates.
- A template consists of a number of **hypothesis tuples** that are meant to show an example of the tuples that may appear in one or more relations.
- The other part of the template is the **template conclusion**.
- For tuple-generating templates, the conclusion is a *set of tuples* that must also exist in the relations if the hypothesis tuples are there.

- For constraint-generating templates, the template conclusion is a *condition* that must hold on the hypothesis tuples.
- Using constraint generating templates, we are able to define **semantic constraints**—those that are beyond the scope of the relational model in terms of its data definition language and notation.
- Figure 16.5 shows how we may define functional, multivalued, and inclusion dependencies by templates.

Figure 16.5

Templates for some common type of dependencies.

- Template for functional dependency $X \rightarrow Y$.
- Template for the multivalued dependency $X \rightarrow\!\!\! \rightarrow Y$.
- Template for the inclusion dependency $R[X] < S[Y]$.

(a)	$R = \{A, B, C, D\}$									
	Hypothesis	$X = \{A, B\}$ $Y = \{C, D\}$								
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>a₁</td><td>b₁</td><td>c₁</td><td>d₁</td></tr> <tr><td>a₁</td><td>b₁</td><td>c₂</td><td>d₂</td></tr> </table>	a ₁	b ₁	c ₁	d ₁	a ₁	b ₁	c ₂	d ₂	
a ₁	b ₁	c ₁	d ₁							
a ₁	b ₁	c ₂	d ₂							
	Conclusion	$c_1 = c_2 \text{ and } d_1 = d_2$								
(b)	$R = \{A, B, C, D\}$									
	Hypothesis	$X = \{A, B\}$ $Y = \{C\}$								
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>a₁</td><td>b₁</td><td>c₁</td><td>d₁</td></tr> <tr><td>a₁</td><td>b₁</td><td>c₂</td><td>d₂</td></tr> </table>	a ₁	b ₁	c ₁	d ₁	a ₁	b ₁	c ₂	d ₂	
a ₁	b ₁	c ₁	d ₁							
a ₁	b ₁	c ₂	d ₂							
	Conclusion	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>a₁</td><td>b₁</td><td>c₂</td><td>d₁</td></tr> <tr><td>a₁</td><td>b₁</td><td>c₁</td><td>d₂</td></tr> </table>	a ₁	b ₁	c ₂	d ₁	a ₁	b ₁	c ₁	d ₂
a ₁	b ₁	c ₂	d ₁							
a ₁	b ₁	c ₁	d ₂							
(c)	$R = \{A, B, C, D\}$	$S = \{E, F, G\}$	$X = \{C, D\}$ $Y = \{E, F\}$							
	Hypothesis	$\begin{array}{ c c c } \hline a_1 & b_1 & c_1 & d_1 \\ \hline \end{array}$								
	Conclusion	$\begin{array}{ c c c } \hline c_1 & d_1 & g \\ \hline \end{array}$								

- Figure 16.6 shows how we may specify the constraint that an employee's salary cannot be higher than the salary of his or her direct supervisor on the relation schema EMPLOYEE

Figure 16.6

Templates for the constraint that an employee's salary must be less than the supervisor's salary.

EMPLOYEE = {Name, Ssn, . . . , Salary, Supervisor_ssn}

	a	b	c	d
Hypothesis	e	d	f	g
Conclusion			$c < f$	

4.15.3 Functional Dependencies Based on Arithmetic Functions and Procedures

- Sometimes some attributes in a relation may be related via some arithmetic function or a more complicated functional relationship.
- As long as a unique value of Y is associated with every X, we can still consider that the FD $X \rightarrow Y$ exists.
- For example, in the relation

ORDER_LINE (Order#, Item#, Quantity, Unit_price, Extended_price,
Discounted_price)
- each tuple represents an item from an order with a particular quantity, and the price per unit for that item.
- In this relation, $(\text{Quantity}, \text{Unit_price}) \rightarrow \text{Extended_price}$ by the formula

$$\text{Extended_price} = \text{Unit_price} * \text{Quantity}.$$
- Hence, there is a unique value for Extended_price for every pair (Quantity, Unit_price), and thus it conforms to the definition of functional dependency.
- Moreover, there may be a procedure that takes into account the quantity discounts, the type of item, and so on and computes a discounted price for the total quantity ordered for that item.
- Therefore, we can say

$(\text{Item}\#, \text{Quantity}, \text{Unit_price}) \rightarrow \text{Discounted_price}$, or

$(\text{Item}\#, \text{Quantity}, \text{Extended_price}) \rightarrow \text{Discounted_price}.$

4.15.4 Domain-Key Normal Form

- The idea behind **domain-key normal form (DKNF)** is to specify the *ultimate normal form* that takes into account all possible types of dependencies and constraints.

- A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation
- For a relation in DKNF, it becomes very straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.
- For example, consider a relation CAR(Make, Vin#) (where Vin# is the vehicle identification number) and another relation MANUFACTURE(Vin#,Country) (where Country is the country of manufacture).
- A general constraint may be of the following form: *If the Make is either ‘Toyota’ or ‘Lexus,’ then the first character of the Vin# is a ‘J’ if the country of manufacture is ‘Japan’; if the Make is ‘Honda’ or ‘Acura,’ the second character of the Vin# is a ‘J’ if the country of manufacture is ‘Japan.’*
- There is no simplified way to represent such constraints short of writing a procedure (or general assertions) to test them.
- The procedure COMPUTE_TOTAL_PRICE above is an example of such procedures needed to enforce an appropriate integrity constraint.



Problem 1

Consider the following relation for published books:

BOOK(BookTitle, AuthorName, BookType, ListPrice, AuthorAffiliation, Publisher)

Suppose the following dependencies exist:

- $\text{BookTitle} \rightarrow \text{BookType}, \text{Publisher}$
- $\text{BookType} \rightarrow \text{ListPrice}$
- $\text{AuthorName} \rightarrow \text{AuthorAffiliation}$

What normal form is the relation in? explain your answer. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

Solution:

The relation is in 1NF and not in 2NF as no attributes are fully functionally dependent on the key (BookTitle and AuthorName). It is also not in 3NF.



not in 2NF because the partial Dependencies exist

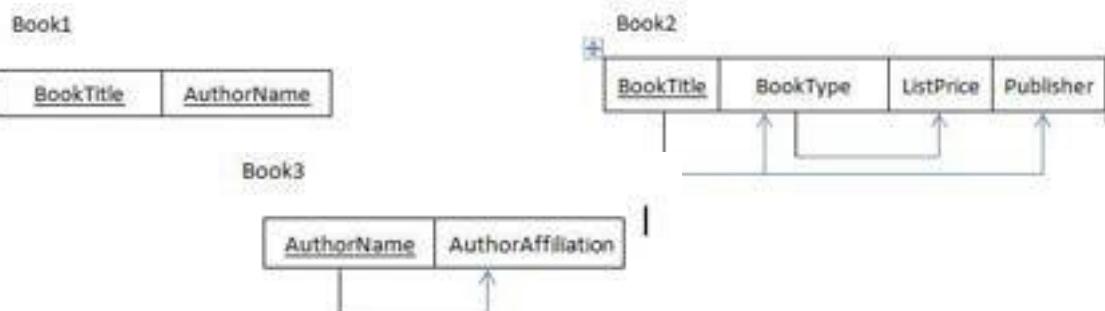
$$\{\text{BookTitle}, \text{AuthorName}\} \rightarrow \{\text{Publisher}, \text{BookType}\}$$

$$\{\text{BookTitle}, \text{AuthorName}\} \rightarrow \text{AuthorAffiliation}$$

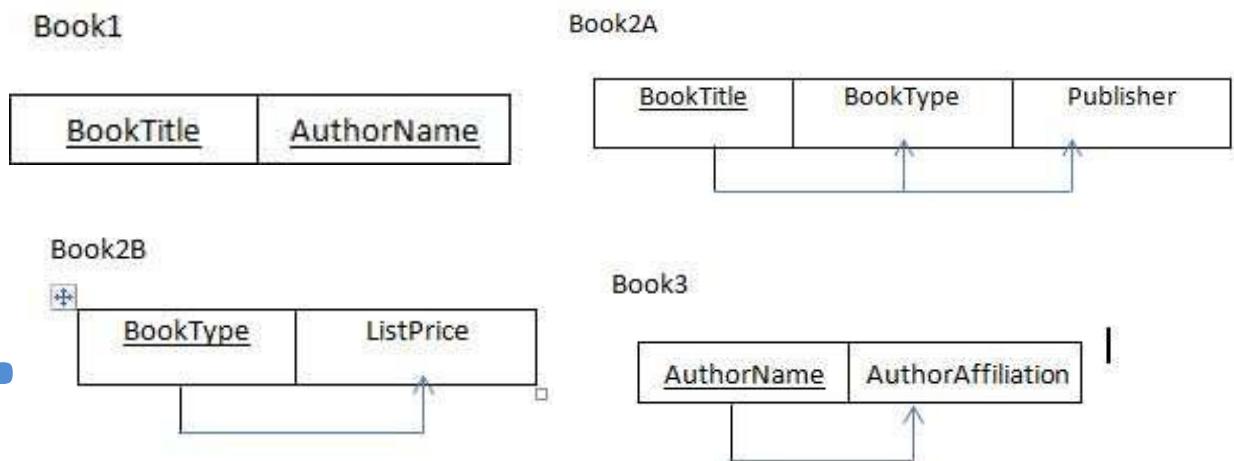
• Thus, these attributes are not fully functionally dependent on the primary key. The 2NF decomposition will eliminate the partial dependencies.

• 2NF decomposition:

- Book1(BookTitle, AuthorName)
- Book2(BookTitle, BookType, ListPrice, Publisher)
- Book3(AuthorName, AuthorAffiliation)



- The relations are not in 3NF because:
- $\text{BookTitle} \rightarrow \text{BookType} \rightarrow \text{ListPrice}$
 BookType is neither a key itself nor a subset of a key and ListPrice is not a prime attribute
- The 3NF decomposition will eliminate the transitive dependency of Listprice . 3NF decomposition:
 - Book1(BookTitle, AuthorName)
 - Book2A(BookTitle, BookType, Publisher)
 - Book2B(BookType, ListPrice)
 - Book3(AuthorName, AuthorAffiliation)



Problem 2

Consider the following relation:

CAR_SALE(Car#, DateSold, Salesman#, Commission%, DiscountAmount)

Assume that a car may be sold by multiple salesmen, and hence

{Car#, Salesman#} is the primary key.

Additional dependencies are:

$\text{Car\#} \rightarrow \text{DateSold}$

$\text{Car\#} \rightarrow \text{DiscountAmount}$

$\text{DateSold} \rightarrow \text{DiscountAmount}$

$\text{Salesman\#} \rightarrow \text{Commission\%}$

Based on the given primary key, is the relation in 1NF, 2NF, 3NF?

Why or why not?

How would you successively normalize it completely?

Solution:

- The relation is in 1NF because all attribute values are single atomic values.
- The relation is not in 2NF because:
 - $\text{Car\#} \rightarrow \text{DateSold}$
 - $\text{Car\#} \rightarrow \text{DiscountAmount}$
 - $\text{Salesman\#} \rightarrow \text{Commission\%}$

Thus, these attributes are not fully functionally dependent on the primary key.

- 2NF decomposition:
 - CAR_SALE1(Car#, DateSold, DiscountAmount)
 - CAR_SALE2(Car#, Salesman#)
 - CAR_SALE3(Salesman#, Commission%)
- The relations are not in 3NF because:
 - $\text{Car\#} \rightarrow \text{DateSold} \rightarrow \text{DiscountAmount}$

DateSold is neither a key itself nor a subset of a key and DiscountAmount is not a prime attribute.

- 3NF decomposition:
 - CAR_SALES1A(Car#, DateSold)
 - CAR_SALES1B(DateSold, DiscountAmount)
 - CAR_SALE2(Car#, Salesman#)
 - CAR_SALE3(Salesman#, Commission%)

4.16 Assignment Questions

1. Consider the following relation for published books:

BOOK(BookTitle, AuthorName, BookType, ListPrice, AuthorAffiliation, Publisher)

Suppose the following dependencies exist:

$\text{BookTitle} \rightarrow \text{BookType}, \text{Publisher}$

$\text{BookType} \rightarrow \text{ListPrice}$

$\text{AuthorName} \rightarrow \text{AuthorAffiliation}$

What normal form is the relation in? Explain your answer.

2. Consider the following relation:

CAR_SALE(Car#, DateSold, Salesman#, Commission%, DiscountAmount)

Assume that a car may be sold by multiple salesmen, and hence

{Car#, Salesman#} is the primary key.

Additional dependencies are:

$\text{Car\#} \rightarrow \text{DateSold}$

$\text{Car\#} \rightarrow \text{DiscountAmount}$

$\text{DateSold} \rightarrow \text{DiscountAmount}$

$\text{Salesman\#} \rightarrow \text{Commission\%}$

Based on the given primary key, is the relation in 1NF, 2NF, 3NF?

Why or why not?

How would you successively normalize it completely?

3. Let $R = \{\text{Ssn, Ename, Pnumber, Pname, Plocation, Hours}\}$ and $O = \{\text{R1, R2, R3}\}$ where

$\text{R1} = \text{EMP} = \{\text{Ssn, Ename}\}$

$\text{R2} = \text{PRO} = \{\text{Pnumber, Pname, Plocation}\}$

$\text{R3} = \text{WORKS-ON} = \{\text{Ssn, Pnumber, Hours}\}$

The following functional dependencies hold on relation R.

$F = \{\text{Ssn} \rightarrow \text{Ename}; \text{Pnumber} \rightarrow \{\text{Pname, Plocation}\};$

$\{\text{Ssn, Pnumber}\} \rightarrow \text{Hours}\}$

Prove that the above decomposition of relation R has the loss less join property.

4. Consider $R = \{\text{A B C D E F}\}$ FDS $\{\text{AB} \rightarrow \text{B} \rightarrow \text{E}, \text{A} \rightarrow \text{DF}\}$

Check whether decomposition is lossless.

5. What is a set of functional dependencies F said to be minimal? Give an algorithm for
 - finding a minimal cover G for F.

4.17 Expected Outcome

- ❖ To design a database which will have minimum redundancy
- ❖ To apply normalization to the designed database.
- ❖ To decompose the tables and normalize the design upto 4NF and 5 NFthe tables upto 4NF and 5NF
- ❖ To apply lossless and lossy join operations
- ❖ To apply inference rules and deduce other rules from the given set.

4.18 Further Reading

1. <https://www.smartdraw.com/entity-relationship-diagram/>
2. https://en.wikipedia.org/wiki/Database_normalization
3. www.databasteknik.se/webbkursen/relalg-lecture
4. [https://technet.microsoft.com/en-us/library/bb264565\(v=sql.90\).aspx](https://technet.microsoft.com/en-us/library/bb264565(v=sql.90).aspx)
5. pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/.../Ch16_Overview_Xacts.pdf

Module 5

Chapter 1: Transaction Processing

5.0 Introduction

5.1 Objectives

5.2 Introduction to Transaction Processing

 5.2.1 Single-User versus Multiuser Systems

 5.2.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers

 5.2.3 Why Concurrency Control Is Needed

 5.2.4 Why Recovery Is Needed

5.3 Transaction and System Concepts

 5.3.1 Transaction States and Additional Operations

 5.3.2 The System Log

 5.3.3 Commit Point of a Transaction:

 5.3.4 DBMS specific buffer Replacement policies

5.4 Desirable Properties of Transactions

5.5 Characterizing Schedules Based on Recoverability

5.6 Characterizing Schedules Based on Serializability

 5.6.1 Testing conflict serializability of a Schedule S

5.7 Transaction Support in SQL

5.8 Introduction to Concurrency Control

5.9 Two-Phase Locking Techniques for Concurrency Control

 5.9.1 Types of Locks and System Lock Tables

 5.9.2 Guaranteeing Serializability by Two-Phase Locking

5.10 Variations of Two-Phase Locking

5.11 Dealing with Deadlock and Starvation

5.11 Deadlock Detection.

5.13 Concurrency Control Based on Timestamp Ordering

 5.13.1 Timestamps

 5.13.2 The Timestamp Ordering Algorithm

5.14 Multiversion Concurrency Control Techniques

 5.14.1 Multiversion Technique Based on Timestamp Ordering

 5.14.2 Multiversion Two-Phase Locking Using Certify Locks

5.15 Validation (Optimistic) Concurrency Control Techniques

5.16 Granularity of Data Items and Multiple Granularity Locking

 5.16.1 Granularity Level Considerations for Locking

 5.16.2 Multiple Granularity Level Locking

5.17 Recovery Concepts

- 5.17.1 Recovery Outline and Categorization of Recovery Algorithms
 - 5.17.2 Caching (Buffering) of Disk Blocks
 - 5.17.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force
 - 5.17.4 Checkpoints in the System Log and Fuzzy Checkpointing
 - 5.17.5 Transaction Rollback and Cascading Rollback
 - 5.17.6 Transaction Actions That Do Not Affect the Database
- 5.18 NO-UNDO/REDO Recovery Based on Deferred Update
- 5.19 Recovery Techniques Based on Immediate Update
- 5.20 Shadow Paging
- 5.21 The ARIES Recovery Algorithm
- 5.22 Database Backup and Recovery from Catastrophic Failures
- 5.23 Assignment Questions
- 5.24 Expected Outcome
- 5.25 Further Reading



5.0 Introduction

The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples:

- airline reservations
- banking
- credit card processing,
- online retail purchasing,
- Stock markets, supermarket checkouts, and many other applications

These systems require high availability and fast response time for hundreds of concurrent users. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

5.1 Objectives

- ❖ To study transaction properties
- ❖ To study creation of schedule and maintaining schedule equivalence.
- ❖ To check whether the given schedule is serializable or not.
- ❖ To study protocols used for locking objects
- ❖ Differentiating between 2PL and Strict 2PL

5.2 Introduction to Transaction Processing

5.2.1 Single-User versus Multiuser Systems

- One criterion for classifying a database system is according to the number of users who can use the system **concurrently**

Single-User versus Multiuser Systems

- A DBMS is
- **single-user**
 - at most one user at a time can use the system
 - Eg: Personal Computer System
- **multiuser**
 - many users can use the system and hence access the database concurrently
 - Eg: Airline reservation database

- Concurrent access is possible because of **Multiprogramming**. Multiprogramming can be achieved by:
 - interleaved execution
 - Parallel Processing
- **Multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on
- A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again
- Hence, concurrent execution of processes is actually **interleaved**, as illustrated in Figure 21.1

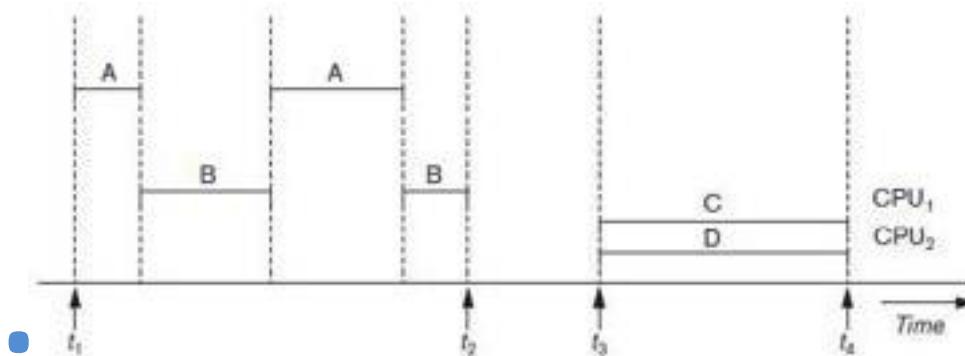


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

- Figure 21.1, shows two processes, A and B, executing concurrently in an interleaved fashion
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk
- The CPU is switched to execute another process rather than remaining idle during I/O time
- Interleaving also prevents a long process from delaying other processes.
- If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1
- Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**
- In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

5.2.2 Transactions, Database Items, Read and Write Operations, and DBMS

Buffers

- A Transaction is an executing program that forms a logical unit of database processing
- It includes one or more DB access operations such as insertion, deletion, modification or retrieval operation.
- It can be either embedded within an application program using **begin transaction** and **end transaction** statements Or specified interactively via a high level query language such as SQL
- Transaction which do not update database are known as **read only transactions**.
- Transaction which do update database are known as **read write transactions**.
- A **database** is basically represented as a collection of named data items. The size of a data item is called its **granularity**.
- A **data item** can be a database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database
- Each data item has a unique name
- **Basic DB access operations that a transaction can include are:**
 - **read_item(X)**: Reads a DB item named X into a program variable.
 - **write_item(X)**: Writes the value of a program variable into the DB item named X
- **Executing read_item(X) include the following steps:**
 1. Find the address of the disk block that contains item X
 2. Copy the block into a buffer in main memory
 3. Copy the item X from the buffer to program variable named X.
- **Executing write_item(X) include the following steps:**
 1. Find the address of the disk block that contains item X
 2. Copy the disk block into a buffer in main memory
 3. Copy item X from program variable named X into its correct location in buffer.
 4. Store the updated disk block from buffer back to disk (either immediately or later).
- Decision of when to store a modified disk block is handled by **recovery manager** of the DBMS in cooperation with operating system.
- A DB cache includes a number of data buffers.
- When the buffers are all occupied a buffer replacement policy is used to choose one of the buffers to be replaced. EG: LRU

- A transaction includes read_item and write_item operations to access and update DB.

(a)	T_1	(b)	T_2
	<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>		<pre>read_item(X); X := X + M; write_item(X);</pre>

Figure 21.2
Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2 .

- The **read-set** of a transaction is the set of all items that the transaction reads
- The **write-set** is the set of all items that the transaction writes
- For example, the read-set of T_1 in Figure 21.2 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

5.2.3 Why Concurrency Control Is Needed

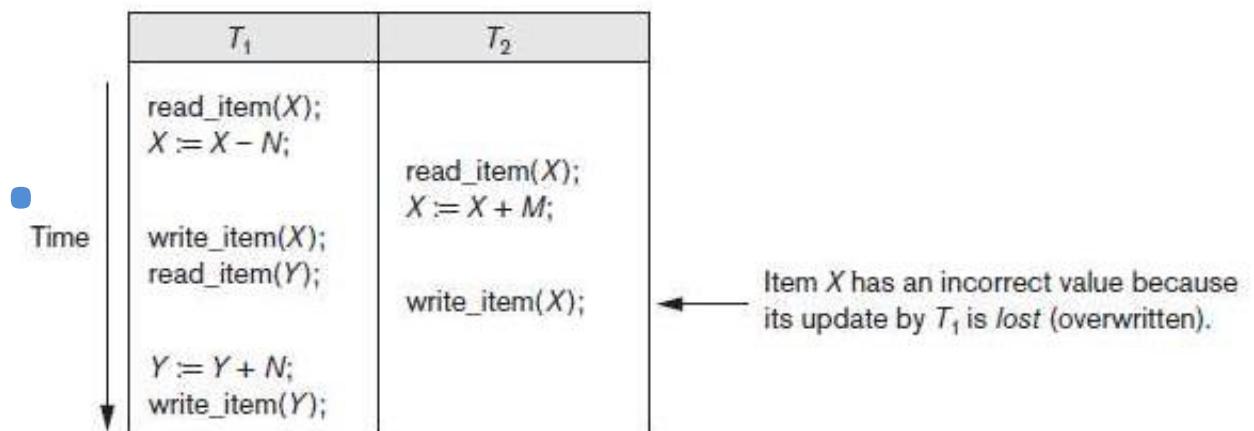
- Several problems can occur when concurrent transactions execute in an uncontrolled manner
- Example:
 - We consider an Airline reservation DB
 - Each record is stored for an airline flight which includes Number of reserved seats among other information.
 - Types of problems we may encounter:
 1. The Lost Update Problem
 2. The Temporary Update (or Dirty Read) Problem
 3. The Incorrect Summary Problem
 4. The Unrepeatable Read Problem

T_2	T_1
<pre>read_item(X); X := X + M; write_item(X);</pre>	<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>

- Transaction T1
 - transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- Transaction T2
 - reserves M seats on the first flight (X)

1. The Lost Update Problem

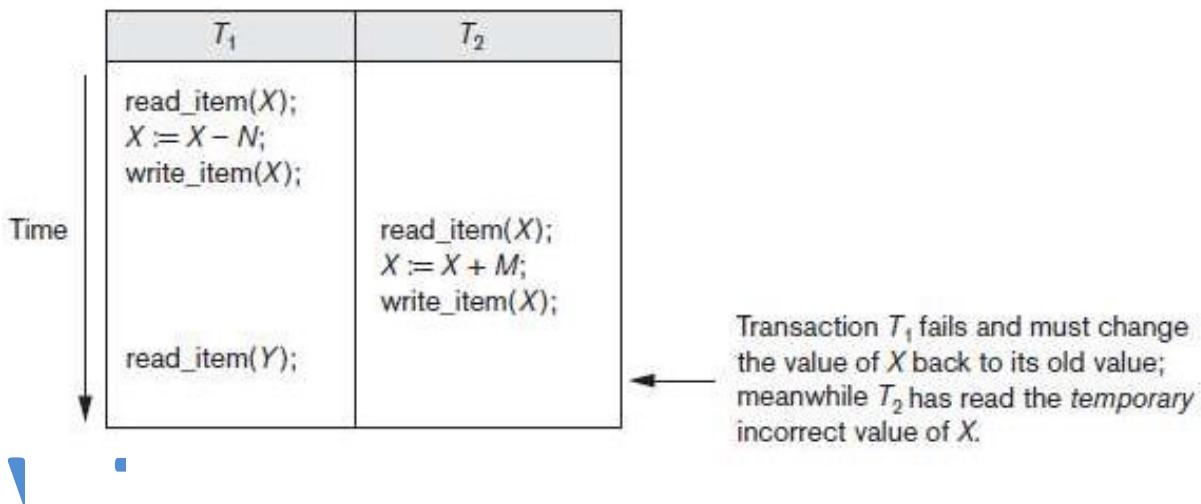
- occurs when two transactions that access the same DB items have their operations interleaved in a way that makes the value of some DB item incorrect
- Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure below



- Final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 is lost.
- For example:
 - $X = 80$ at the start (there were 80 reservations on the flight)
 - $N = 5$ (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y)
 - $M = 4$ (T_2 reserves 4 seats on X)
 - The final result should be $X = 79$.
- The interleaving of operations shown in Figure is $X = 84$ because the update in T_1 that removed the five seats from X was lost.

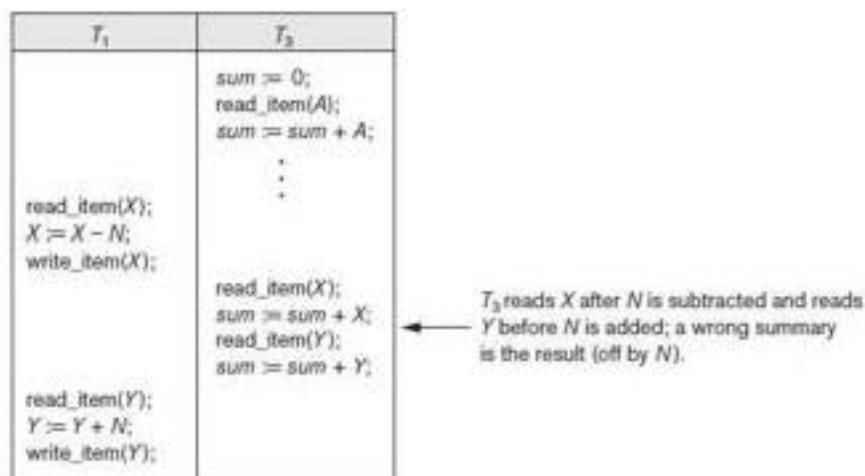
2. The Temporary Update (or Dirty Read) Problem

- occurs when one transaction updates a database item and then the transaction fails for some reason
- Meanwhile the updated item is accessed by another transaction before it is changed back to its original value



3. The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of db items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.



4. The Unrepeatable Read Problem

- Transaction T reads the same item twice and gets different values on each read, since the item was modified by another transaction T' between the two reads.
- for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights
- When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

5.2.4 Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either
 1. All the operations in the transaction are completed successfully and their effect is recorded permanently in the database or
 2. The transaction does not have any effect on the database or any other transactions
- In the first case, the transaction is said to be committed, whereas in the second case, the transaction is aborted
- If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of failures

1. A computer failure (system crash):

- A hardware, software, or network error occurs in the computer system during transaction execution
- Hardware crashes are usually media failures—for example, main memory failure.

2. A transaction or system error:

- Some operation in the transaction may cause it to fail, such as integer overflow or division by zero
- Also occur because of erroneous parameter values

3. Local errors or exception conditions detected by the transaction:

- During transaction execution, certain conditions may occur that necessitate cancellation of the transaction

- For example, data for the transaction may not be found

4. Concurrency control enforcement:

- The concurrency control may decide to abort a transaction because it violates serializability or several transactions are in a state of deadlock

5. Disk failure:

- Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.

6. Physical problems and catastrophes:

- refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake
- Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6.
- Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure.
- Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

5.3 Transaction and System Concepts

5.3.1 Transaction States and Additional Operations

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system keeps track of start of a transaction, termination, commit or aborts.
 - **BEGIN_TRANSACTION:** marks the beginning of transaction execution
 - **READ or WRITE:** specify read or write operations on the database items that are executed as part of a transaction
 - **END_TRANSACTION:** specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution
 - **COMMIT_TRANSACTION:** signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone
 - **ROLLBACK:** signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be **undone**

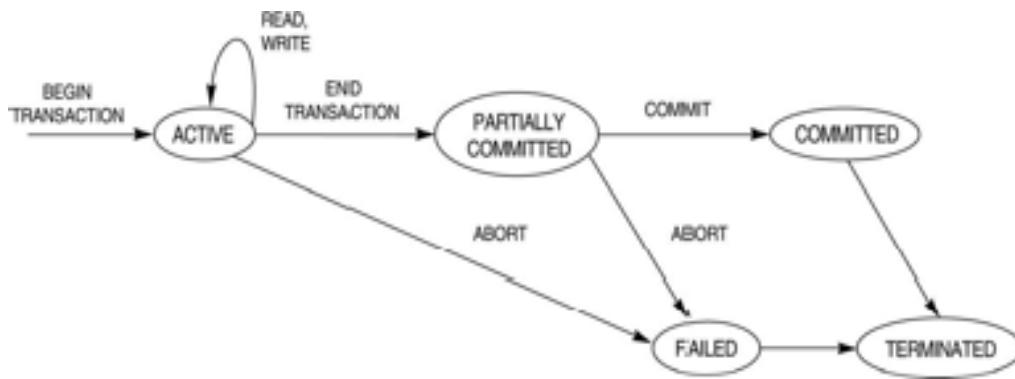


Figure: State transition diagram illustrating the states for transaction execution

- A transaction goes into **active state** immediately after it starts execution and can execute read and write operations.
- When the transaction ends it moves to **partially committed state**.
- At this end additional checks are done to see if the transaction can be committed or not. If these checks are successful the transaction is said to have reached commit point and enters **committed state**. All the changes are recorded permanently in the db.
- A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its write operation.
- Terminated state corresponds to the transaction leaving the system. All the information about the transaction is removed from system tables.
-

5.3.2 The System Log

- **Log or Journal** keeps track of all transaction operations that affect the values of database items
 - This information may be needed to permit recovery from transaction failures.
 - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure
 - One (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer
 - When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*.

- In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures
- The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record.
- In these entries, T refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:
 1. **[start_transaction, T]**. Indicates that transaction T has started execution.
 2. **[write_item, T, X, old_value, new_value]**. Indicates that transaction T has changed the value of database item X from old_value to new_value.
 3. **[read_item, T, X]**. Indicates that transaction T has read the value of database item X.
 4. **[commit, T]**. Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
 5. **[abort, T]**. Indicates that transaction T has been aborted.

5.3.3 Commit Point of a Transaction:

- **Definition a Commit Point:**
 - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
 - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
 - The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:**
 - Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

5.3.4 DBMS specific buffer Replacement policies

Domain Separation(DS) method

- DBMS cache is divided into separate domains, each handles one type of disk pages and replacements within each domain are handled via basic LRU page replacement.
- LRU is a **static** algorithm and does not adopt to dynamically changing loads because the number of available buffers for each domain is predetermined.
- **Group LRU** adds dynamically load balancing feature since it gives each domain a priority and selects pages from lower priority level domain first for replacement.

Hot Set Method:

- This is useful in queries that have to scan a set of pages repeatedly.
- The hot set method determines for each db processing algorithm the set of disk pages that will be accessed repeatedly and it does not replace them until their processing is completed.

The DBMIN method:

- uses a model known as QLSM (Query Locality set model), which predetermines the pattern of page references for each algorithm for a particular db operation
- Depending on the type of access method, the file characteristics, and the algorithm used the QLSM will estimate the number of main memory buffers needed for each file involved in the operation.

5.4 Desirable Properties of Transactions

- Transactions should possess several properties, often called the **ACID** properties
 - A **Atomicity**: a transaction is an atomic unit of processing and it is either performed entirely or not at all.
 - C **Consistency Preservation**: a transaction should be consistency preserving that is it must take the database from one consistent state to another.
 - I **Isolation/Independence**: A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executed concurrently.
 - D **Durability (or Permanency)**: if a transaction changes the database and is committed, the changes must never be lost because of any failure.
- The **atomicity** property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.
- The preservation of **consistency** is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.
- The **isolation** property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks
- **Durability** is the responsibility of recovery subsystem.

5.5 Characterizing Schedules Based on Recoverability

- **schedule** (or **history**): the order of execution of operations from all the various transactions
- **Schedules (Histories) of Transactions:** A schedule S of n transactions T_1, T_2, \dots, T_n is a sequential ordering of the operations of the n transactions.
 - The transactions are interleaved
- Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
 - (1) they belong to *different transactions*;
 - (2) they access the *same item X*; and
 - (3) *at least one* of the operations is a *write_item(X)*
- **Conflicting operations:**
 - $r_1(X)$ conflicts with $w_2(X)$ Read write conflict
 - $r_2(X)$ conflicts with $w_1(X)$
 - $w_1(X)$ conflicts with $w_2(X)$ Write conflict
 - $r_1(X)$ do not conflicts with $r_2(X)$

Schedules classified on recoverability:

- **Recoverable schedule:**
 - One where no transaction needs to be rolled back.
 - A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
 - Example:
 - $S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$
 - $S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
- **Cascadeless schedule:**
 - One where every transaction reads only the items that are written by committed transactions.
- **Schedules requiring cascaded rollback:**
 - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules:**
 - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

5.6 Characterizing Schedules Based on Serializability

- schedules that are always considered to be correct when concurrent transactions are executing are known as **serializable** schedules
- Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:
 1. Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence).
 2. Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

Figure 21.5

Examples of serial and nonserial schedules involving transactions T_1 and T_2 . (a) Serial schedule A: T_1 followed by T_2 . (b) Serial schedule B: T_2 followed by T_1 . (c) Two nonserial schedules C and D with interleaving of operations.

(a)	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 50%; text-align: center;">T_1</th> <th style="width: 50%; text-align: center;">T_2</th> </tr> <tr> <td style="text-align: center;">read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);</td> <td style="text-align: center;">read_item(X); $X := X + M$; write_item(X);</td> </tr> </table>	T_1	T_2	read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);	(b)	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 50%; text-align: center;">T_1</th> <th style="width: 50%; text-align: center;">T_2</th> </tr> <tr> <td style="text-align: center;">read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);</td> <td style="text-align: center;">read_item(X); $X := X + M$; write_item(X);</td> </tr> </table>	T_1	T_2	read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);
T_1	T_2										
read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);										
T_1	T_2										
read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);										
Time ↓	↓	Time ↓	↓								
(c)	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 50%; text-align: center;">T_1</th> <th style="width: 50%; text-align: center;">T_2</th> </tr> <tr> <td style="text-align: center;">read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);</td> <td style="text-align: center;">read_item(X); $X := X + M$; write_item(X);</td> </tr> </table>	T_1	T_2	read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);	(d)	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 50%; text-align: center;">T_1</th> <th style="width: 50%; text-align: center;">T_2</th> </tr> <tr> <td style="text-align: center;">read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);</td> <td style="text-align: center;">read_item(X); $X := X + M$; write_item(X);</td> </tr> </table>	T_1	T_2	read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);
T_1	T_2										
read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);										
T_1	T_2										
read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);	read_item(X); $X := X + M$; write_item(X);										
Time ↓	↓	Time ↓	↓								
Schedule A	↓	Schedule B	↓								
Time ↓	↓	Time ↓	↓								
Schedule C	↓	Schedule D	↓								

- **Serial schedule:**
 - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
 - Otherwise, the schedule is called nonserial schedule.
- **Serializable schedule:**
 - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- **Result equivalent:**
 - Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent:**
 - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable:**
 - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
 - It will leave the database in a consistent state.
 - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

5.6.1 Testing conflict serializability of a Schedule S

For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.

For each case in S where T_j executes a `read_item(X)` after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

For each case in S where T_j executes a `write_item(X)` after T_i executes a `read_item (X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

For each case in S where T_j executes a `write_item(X)` after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

The schedule S is serializable if and only if the precedence graph has no cycles.

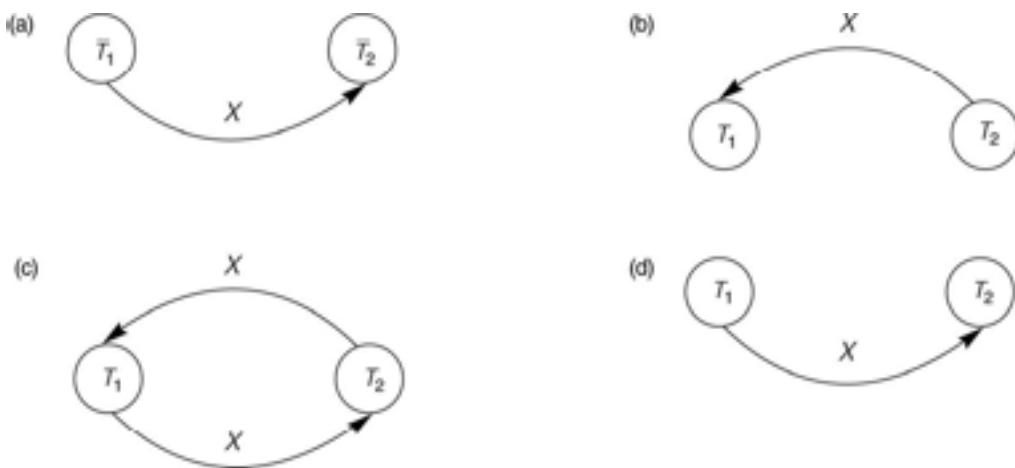


Fig: Constructing the precedence graphs for schedules *A* and *D* from fig 21.5 to test for conflict serializability.

- (a) Precedence graph for serial schedule *A*.
- (b) Precedence graph for serial schedule *B*.
- (c) Precedence graph for schedule *C* (not serializable).
- (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

- Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1 , T_2 , and T_3 .

(a)	transaction T_1	transaction T_2	transaction T_3
	<pre>read_item (X); write_item (X); read_item (Y); write_item (Y);</pre>	<pre>read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);</pre>	<pre>read_item (Y); read_item (Z); write_item (Y); write_item (Z);</pre>

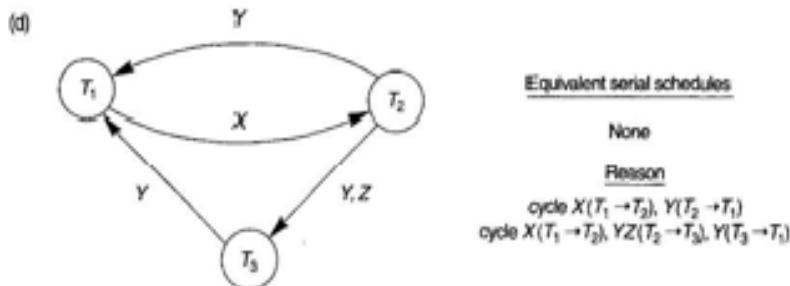
(b)

	transaction T_1	transaction T_2	transaction T_3
<i>Time</i>		read_item (Z); read_item (Y); write_item (Y);	
	read_item (X); write_item (X);		read_item (Y); read_item (Z);
	read_item (Y); write_item (Y);	read_item (X); write_item (X);	write_item (Y); write_item (Z);
			Schedule E

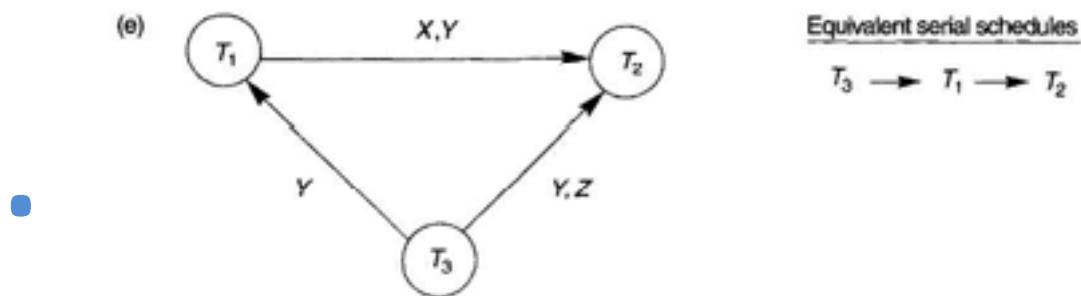
(c)

	transaction T_1	transaction T_2	transaction T_3
<i>Time</i>			read_item (Y); read_item (Z);
	read_item (X); write_item (X);	read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);	write_item (Y); write_item (Z);
	read_item (Y); write_item (Y);		
			Schedule F

- Precedence graph for schedule E



- Precedence graph for schedule F



5.7 Transaction Support in SQL

- The basic definition of an SQL transaction is, it is a logical unit of work and is guaranteed to be atomic
- A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged
- With SQL, there is no explicit `Begin_Transaction` statement. Transaction initiation is done implicitly when particular SQL statements are encountered
- Every transaction must have an explicit end statement, which is either a `COMMIT` or a `ROLLBACK`
- Every transaction has certain characteristics attributed to it and are specified by a `SET TRANSACTION` statement in SQL

- The characteristics are :
 - **The access mode**
 - can be specified as READ ONLY or READ WRITE
 - The default is READ WRITE
 - A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed
 - A mode of READ ONLY, as the name implies, is simply for data retrieval.
 - **The diagnostic area size**
 - DIAGNOSTIC SIZE n, specifies an integer value n, which indicates the number of conditions that can be held simultaneously in the diagnostic area
 - These conditions supply feedback information (errors or exceptions) to the user or program on the n most recently executed SQL statement
 - **The isolation level**
 - specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE
 - - The default isolation level is SERIALIZABLE
 - The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms
 - If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:
 1. **Dirty read.** A transaction T_1 may read the update of a transaction T_2 , which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect.
 2. **Nonrepeatable read.** A transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value.
 3. **Phantoms.** A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T_2 inserts a new row that also satisfies the WHERE-clause condition used in T_1 , into the table used by T_1 . If T_1 is repeated, then T_1 will see a phantom, a row that previously did not exist.

Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

Isolation Level	Type of Violation		
	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

```

EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;

```

- The transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2
- If an error occurs on any of the SQL statements, the entire transaction is rolled back
- This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

Chapter 2: Concurrency Control in Databases

5.8 Introduction to Concurrency Control

- Purpose of Concurrency Control
 - To enforce Isolation (through mutual exclusion) among conflicting transactions.
 - To preserve database consistency through consistency preserving execution of transactions.
 - To resolve read-write and write-write conflicts.
- Example:
 - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

5.9 Two-Phase Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item.
- It is used for synchronizing the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

5.9.1 Types of Locks and System Lock Tables

1. Binary Locks

- A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0).
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item

If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1

We refer to the current value (or state) of the lock associated with item X as **lock(X)**.

- Two operations, **lock_item** and **unlock_item**, are used with binary locking.
- A transaction requests access to an item X by first issuing a **lock_item(X)** operation
 - If $\text{LOCK}(X) = 1$, the transaction is forced to wait.
 - If $\text{LOCK}(X) = 0$, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X
- When the transaction is through using the item, it issues an **unlock_item(X)** operation, which sets $\text{LOCK}(X)$ back to 0 (**unlocks** the item) so that X may be accessed by other transactions
- Hence, a binary lock enforces **mutual exclusion** on the data item.

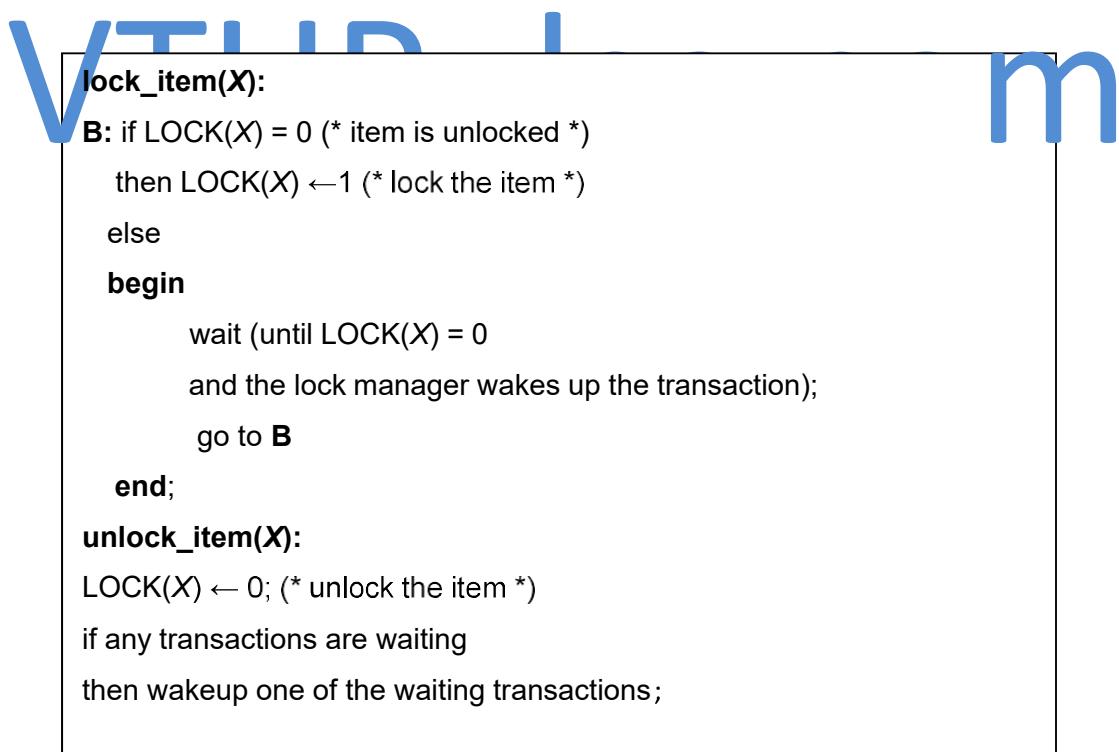


Fig: 2.1.1 Lock and unlock operations for binary locks.

- The lock_item and unlock_item operations must be implemented as indivisible units that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits
- The wait command within the lock_item(X) operation is usually implemented by putting the transaction in a waiting queue for item X until X is unlocked and the transaction can be granted access to it
- Other transactions that also want to access X are placed in the same queue. Hence, the wait command is considered to be outside the lock_item operation.
- It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database
- In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item
- If the simple binary locking scheme described here is used, every transaction must obey the following rules:
 1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T .
 2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T .
 3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X .
 4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X .

2. Shared/Exclusive (or Read/Write) Locks

- binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item
- should allow several transactions to access the same item X if they all access X for reading purposes only
- if a transaction is to write an item X , it must have exclusive access to X
- For this purpose, a different type of lock called a **multiple-mode lock** is used
- In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: **read_lock(X)**, **write_lock(X)**, and **unlock(X)**.

- A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item
- Method to implement read/write lock is to
 - keep track of the number of transactions that hold a shared (read) lock on an item in the lock table
 - Each record in the lock table will have four fields:
 $\langle \text{Data_item_name}, \text{LOCK}, \text{No_of_reads}, \text{Locking_transaction(s)} \rangle$.
- If $\text{LOCK}(X)$ =write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X
- If $\text{LOCK}(X)$ =read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.

read_lock(X):

```
B: if  $\text{LOCK}(X)$  = "unlocked"
    then begin  $\text{LOCK}(X) \leftarrow \text{"read-locked"};$ 
             $\text{no\_of\_reads}(X) \leftarrow 1$ 
        end
    else if  $\text{LOCK}(X)$  = "read-locked"
        then  $\text{no\_of\_reads}(X) \leftarrow \text{no\_of\_reads}(X) + 1$ 
    else begin
        wait (until  $\text{LOCK}(X)$  = "unlocked"
            and the lock manager wakes up the transaction);
        go to B
    end;
```

write_lock(X):

```
B: if  $\text{LOCK}(X)$  = "unlocked"
    then  $\text{LOCK}(X) \leftarrow \text{"write-locked"}$ 
    else begin
        wait (until  $\text{LOCK}(X)$  = "unlocked"
            and the lock manager wakes up the transaction);
        go to B
    end;
```

unlock (X):

```

if LOCK(X) = "write-locked"
    then begin LOCK(X) ← "unlocked";
            wakeup one of the waiting transactions, if any
        end
else if LOCK(X) = "read-locked"
    then begin
        no_of_reads(X) ← no_of_reads(X) -1;
        if no_of_reads(X) = 0
            then begin LOCK(X) = "unlocked";
                    wakeup one of the waiting transactions, if any
                end
        end;
    
```

- When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation `read_lock(X)` or `write_lock(X)` before any `read_item(X)` operation is performed in T.
2. A transaction T must issue the operation `write_lock(X)` before any `write_item(X)` operation is performed in T.
3. A transaction T must issue the operation `unlock(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.
4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

- A transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another
- For example, it is possible for a transaction T to issue a `read_lock(X)` and then later to **upgrade** the lock by issuing a `write_lock(X)` operation
 - If T is the only transaction holding a read lock on X at the time it issues the `write_lock(X)` operation, the lock can be upgraded; otherwise, the transaction must wait

5.9.2 Guaranteeing Serializability by Two-Phase Locking

- A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction
- Such a transaction can be divided into two phases:
 - Expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released
 - Shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired
- If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.
- Transactions T_1 and T_2 in Figure 22.3(a) do not follow the two-phase locking protocol because the $\text{write_lock}(X)$ operation follows the $\text{unlock}(Y)$ operation in T_1 , and similarly the $\text{write_lock}(Y)$ operation follows the $\text{unlock}(X)$ operation in T_2 .

(a)

T_1	T_2
$\text{read_lock}(Y);$ $\text{read_item}(Y);$ $\text{unlock}(Y);$ $\text{write_lock}(X);$ $\text{read_item}(X);$ $X := X + Y;$ $\text{write_item}(X);$ $\text{unlock}(X);$	$\text{read_lock}(X);$ $\text{read_item}(X);$ $\text{unlock}(X);$ $\text{write_lock}(Y);$ $\text{read_item}(Y);$ $Y := X + Y;$ $\text{write_item}(Y);$ $\text{unlock}(Y);$

(b)

Initial values: $X=20, Y=30$

Result serial schedule T_1 , followed by T_2 : $X=50, Y=80$

Result of serial schedule T_2 , followed by T_1 : $X=70, Y=50$

(c)

T_1	T_2
$\text{read_lock}(Y);$ $\text{read_item}(Y);$ $\text{unlock}(Y);$ $\text{write_lock}(X);$ $\text{read_item}(X);$ $X := X + Y;$ $\text{write_item}(X);$ $\text{unlock}(X);$	$\text{read_lock}(X);$ $\text{read_item}(X);$ $\text{unlock}(X);$ $\text{write_lock}(Y);$ $\text{read_item}(Y);$ $Y := X + Y;$ $\text{write_item}(Y);$ $\text{unlock}(Y);$

Time

Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions T_1 and T_2 (b) Results of possible serial schedules of T_1 and T_2 (c) A nonserializable schedule S that uses locks

- If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2' as shown in Figure 22.4.
- Now, the schedule shown in Figure 22.3(c) is not permitted for T_1' and T_2' (with their modified order of locking and unlocking operations) under the rules of locking because T_1' will issue its `write_lock(X)` before it unlocks item Y; consequently, when T_2' issues its `read_lock(X)`, it is forced to wait until T_1' releases the lock by issuing an unlock (X) in the schedule.

Figure 22.4

Transactions T_1' and T_2' , which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

T_1'	T_2'
<code>read_lock(Y);</code> <code>read_item(Y);</code> <code>write_lock(X);</code> <code>unlock(Y);</code> <code>read_item(X);</code> $X := X + Y;$ <code>write_item(X);</code> <code>unlock(X);</code>	<code>read_lock(X);</code> <code>read_item(X);</code> <code>write_lock(Y);</code> <code>unlock(X)</code> <code>read_item(Y);</code> $Y := X + Y;$ <code>write_item(Y);</code> <code>unlock(Y);</code>

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

5.10 Variations of Two-Phase Locking

- **Basic 2PL**
 - Technique described previously
- **Conservative (static) 2PL**
 - Requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring read-set and write-set
 - Its Deadlock-free protocol

- **Strict 2PL**

- guarantees strict schedules
- Transaction does not release exclusive locks until after it commits or aborts
- no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability
- Strict 2PL is not deadlock-free

- **Rigorous 2PL**

- guarantees strict schedules
- Transaction does not release any locks until after it commits or aborts
- easier to implement than strict 2PL

5.11 Dealing with Deadlock and Starvation

- **Deadlock** occurs when each transaction T in a set of two or more transactions is
 - waiting for one or more locks held by other transactions in the set.
- Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.
- A simple example is shown in Figure 22.5(a), where the two transactions T_1' and T_2' are deadlocked in a partial schedule;
- T_1' has a lock on item Y , which is locked by T_1' . Meanwhile,
- T_2' has a lock on item X , which is locked by T_2' .

(a)

T_1'	T_2'
<code>read_lock(Y);</code> <code>read_item(Y);</code> <code>write_lock(X);</code>	<code>read_lock(X);</code> <code>read_item(X);</code> <code>write_lock(Y);</code>

Time ↓

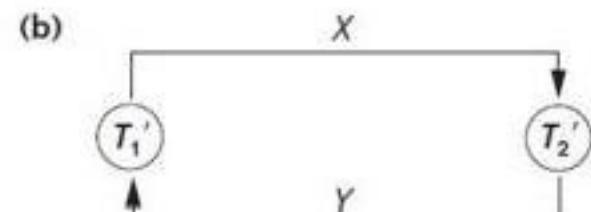


Figure 22.5 Illustrating the deadlock problem (a) A partial schedule of T_1' and T_2' that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

Deadlock prevention protocols

- One way to prevent deadlock is to use a **deadlock prevention protocol**
- One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance*. If any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs.
- A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items
- Both approaches impractical
- Some of these techniques use the concept of **transaction timestamp** $TS(T)$, which is a unique identifier assigned to each transaction
- The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2 , then $TS(T_1) < TS(T_2)$.
- The *older* transaction (which starts first) has the *smaller* timestamp value.
- Protocols based on a timestamp
 - **Wait-die**
 - **Wound-wait**
 - Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:
 - **Wait-die.** If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i dies) and restart it later *with the same timestamp*.
 - **Wound-wait.** If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later *with the same timestamp*; otherwise (T_i younger than T_j) T_i is allowed to wait.
 - In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
 - The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it.

- Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing.
- It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created.
- Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created.
- Another group of protocols that prevent deadlock do not require timestamps. These include the
 - no waiting (NW) and
 - cautious waiting (CW) algorithms
- **No waiting algorithm,**
 - if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.
 - so no deadlock will occur
 - this scheme can cause transactions to abort and restart needlessly
- **cautious waiting**
 - try to reduce ~~unnecessary~~ needless aborts/restarts
 - Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock.
 - The cautious waiting rules are as follows:
 - If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i .
 - It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction.

5.12 Deadlock Detection.

- A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time.

- This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light.
- On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.
- A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**.
- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j , a directed edge $(T_i \rightarrow T_j)$ is created in the wait-for graph.
- When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle.
- One problem with this approach is the matter of determining *when* the system should check for a deadlock.
- One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead.
- Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).
 - If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.
 - Choosing which transactions to abort is known as **victim selection**.
 - The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).
- **Timeouts**
 - Another simple scheme to deal with deadlock is the use of **timeouts**.
 - This method is practical because of its low overhead and simplicity.
 - In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

- **Starvation.** 
 - Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
 - This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others
 - One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
 - Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
 - Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
 - The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.

5.13 Concurrency Control Based on Timestamp Ordering

guarantees serializability using transaction timestamps to order transaction execution for an equivalent serial schedule

5.13.1 Timestamps

- **timestamp** is a unique identifier created by the DBMS to identify a transaction.
- Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*.
- We will refer to the timestamp of transaction T as **TS(T)**.
- Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.
- Timestamps can be generated in several ways.
 - One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3,

... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.

- Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

5.13.2 The Timestamp Ordering Algorithm

- The idea for this scheme is to order the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**.
- The algorithm must ensure that, for each item accessed by *conflicting Operations* in the schedule, the order in which the item is accessed does not violate the timestamp order.
- To do this, the algorithm associates with each database item X two timestamp (**TS**) values:
 - 1. **read_TS(X)**. The **read timestamp** of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X —that is, $\text{read_TS}(X) = \text{TS}(T)$, where T is the *youngest* transaction that has read X successfully.
 - 2. **write_TS(X)**. The **write timestamp** of item X is the largest of all the timestamps of transactions that have successfully written item X —that is, $\text{write_TS}(X) = \text{TS}(T)$, where T is the *youngest* transaction that has written X successfully.

Basic Timestamp Ordering (TO).

- Whenever some transaction T tries to issue a $\text{read_item}(X)$ or a $\text{write_item}(X)$ operation, the **basic TO** algorithm compares the timestamp of T with $\text{read_TS}(X)$ and $\text{write_TS}(X)$ to ensure that the timestamp order of transaction execution is not violated.
- If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*.
- If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back.

- Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.
- An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.
- **The basic TO algorithm :**
 - The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:
 1. Whenever a transaction T issues a `write_item(X)` operation, the following is checked:
 - a. If $\text{read_TS}(X) > \text{TS}(T)$ or if $\text{write_TS}(X) > \text{TS}(T)$, then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than $\text{TS}(T)$ —and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X , thus violating the timestamp ordering.
 - b. If the condition in part (a) does not occur, then execute the `write_item(X)` operation of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.
 2. Whenever a transaction T issues a `read_item(X)` operation, the following is checked:
 - a. If $\text{write_TS}(X) > \text{TS}(T)$, then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than $\text{TS}(T)$ —and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X .
 - b. If $\text{write_TS}(X) \leq \text{TS}(T)$, then execute the `read_item(X)` operation of T and set $\text{read_TS}(X)$ to the *larger* of $\text{TS}(T)$ and the current $\text{read_TS}(X)$.
 - Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*

Strict Timestamp Ordering (TO)

- A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable.

- In this variation, a transaction T that issues a `read_item(X)` or `write_item(X)` such that $TS(T) > write_TS(X)$ has its read or write operation *delayed* until the transaction T that *wrote* the value of X (hence $TS(T) = write_TS(X)$) has committed or aborted.
- To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T until T is either committed or aborted. This algorithm *does not cause deadlock*, since T waits for T' only if $TS(T) > TS(T')$.

Thomas's Write Rule

- A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the `write_item(X)` operation as follows:
 1. If $read_TS(X) > TS(T)$, then abort and roll back T and reject the operation.
 2. If $write_TS(X) > TS(T)$, then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than $TS(T)$ —and hence after T in the timestamp ordering—has already written the value of X . Thus, we must ignore the `write_item(X)` operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by
 - case (1).

If neither the condition in part (1) nor the condition in part (2) occurs, then execute the `write_item(X)` operation of T and set `write_TS(X)` to $TS(T)$.

5.14 Multiversion Concurrency Control Techniques

- Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained
- When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible.
- The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item are retained
- An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items

5.14.1 Multiversion Technique Based on Timestamp Ordering

- In this method, several versions X_1, X_2, \dots, X_k of each data item X are maintained.
- For each version, the value of version X_i and the following two timestamps are kept:
 1. **read_TS(X_i)**. The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i .
 2. **write_TS(X_i)**. The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i .
- Whenever a transaction T is allowed to execute a `write_item(X)` operation, a new version X_{k+1} of item X is created, with both the $\text{write_TS}(X_{k+1})$ and the $\text{read_TS}(X_{k+1})$ set to $\text{TS}(T)$
- Correspondingly, when a transaction T is allowed to read the value of version X_i , the value of $\text{read_TS}(X_i)$ is set to the larger of the current $\text{read_TS}(X_i)$ and $\text{TS}(T)$.
- To ensure serializability, the following rules are used:
 1. If transaction T issues a `write_item(X)` operation, and version i of X has the highest $\text{write_TS}(X_i)$ of all versions of X that is also *less than or equal to* $\text{TS}(T)$, and $\text{read_TS}(X_i) > \text{TS}(T)$, then abort and roll back transaction T ; otherwise, create a new version X_j of X with $\text{read_TS}(X_j) = \text{write_TS}(X_j) = \text{TS}(T)$.
 2. If transaction T issues a `read_item(X)` operation, find the version i of X that has the highest $\text{write_TS}(X_i)$ of all versions of X that is also *less than or equal to* $\text{TS}(T)$; then return the value of X_i to transaction T , and set the value of $\text{read_TS}(X_i)$ to the larger of $\text{TS}(T)$ and the current $\text{read_TS}(X_i)$.

5.14.2 Multiversion Two-Phase Locking Using Certify Locks

- In this multiple-mode locking scheme, there are *three locking modes* for an item: `read`, `write`, and `certify`
- Hence, the state of $\text{LOCK}(X)$ for an item X can be one of `read-locked`, `writelocked`, `certify-locked`, or `unlocked`
- We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a)
- An entry of `Yes` means that if a transaction T holds the type of lock specified in the column header on item X and if transaction T_+ requests the type of lock specified in

the row header on the same item X , then T can obtain the lock because the locking modes are compatible

(a)		Read	Write
	Read	Yes	No
	Write	No	No

(b)		Read	Write	Certify
	Read	Yes	Yes	No
	Write	Yes	No	No
	Certify	No	No	No

Figure 22.6: Lock compatibility tables. (a) A compatibility table for read/write locking scheme.
(b) A compatibility table for read/write/certify locking scheme.

- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so T must wait until T releases the lock
- The idea behind multiversion 2PL is to allow other transactions T' to read an item X while a single transaction T holds a write lock on X
- This is accomplished by allowing *two versions* for each item X ; one version must always have been written by some committed transaction
- The second version X' is created when a transaction T acquires a write lock on the item

5.15 Validation (Optimistic) Concurrency Control Techniques

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing
- In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end

- During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction
- At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability.
- There are three phases for this concurrency control protocol:
 1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
 2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
 3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.
- The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached
- The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.
- The validation phase for T_i checks that, for each such transaction T_j that is either committed or is in its validation phase, *one* of the following conditions holds:
 1. Transaction T_j completes its write phase before T_i starts its read phase.
 2. T_i starts its write phase after T_j completes its write phase, and the *read_set* of T_i has no items in common with the *write_set* of T_j .
 3. Both the *read_set* and *write_set* of T_i have no items in common with the *write_set* of T_j , and T_j completes its read phase before T_i completes its read phase.

5.16 Granularity of Data Items and Multiple Granularity Locking

- All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:
 - A database record
 - A field value of a database record
 - A disk block
 - A whole file

- The whole database
- The granularity can affect the performance of concurrency control and recovery

5.16.1 Granularity Level Considerations for Locking

- The size of data items is often called the **data item granularity**.
- *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes
- The larger the data item size is, the lower the degree of concurrency permitted.
- For example, if the data item size is a disk block, a transaction T that needs to lock a record B must lock the whole disk block X that contains B because a lock is associated with the whole data item (block). Now, if another transaction S wants to lock a different record C that happens to reside in the same block X in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction S would be able to proceed, because it would be locking a different data item (record).
- The smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead
- The best item size *depends on the types of transactions involved*.
- If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record
- On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items

5.16.2 Multiple Granularity Level Locking

- Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be different for various mixes of transactions
- Figure 22.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records.
- This can be used to illustrate a **multiple granularity level 2PL** protocol, where a lock can be requested at any level

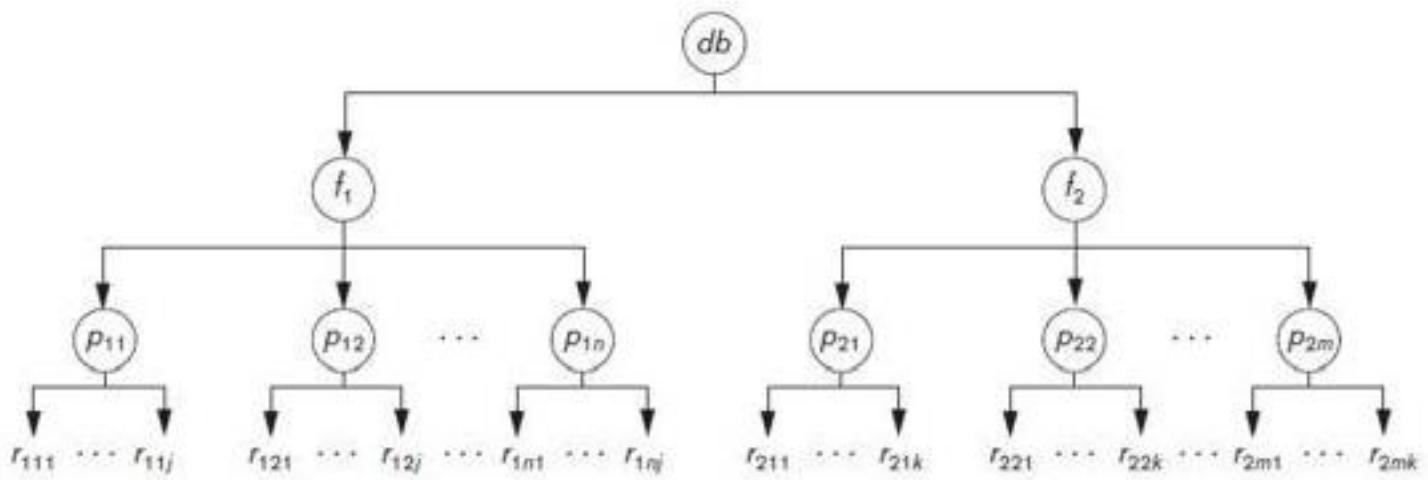


Figure 22.7 A granularity hierarchy for illustrating multiple granularity level locking

- To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed
- The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants.
- There are three types of intention locks:
 1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
 2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
 3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).
- The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8.

	IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	
S	Yes	No	Yes	No	No	
SIX	Yes	No	No	No	No	
X	No	No	No	No	No	

Figure 22.8: Lock compatibility matrix for multiple granularity locking.

- The **multiple granularity locking (MGL)** protocol consists of the following rules:
 1. The lock compatibility (based on Figure 22.8) must be adhered to.
 2. The root of the tree must be locked first, in any mode.
 3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.
 4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
 5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
 6. A transaction T can unlock a node, N , only if none of the children of node N are currently locked by T .

The multiple granularity level protocol is especially suited when processing a mix of transactions that include

- (1) short transactions that access only a few items (records or fields) and
- (2) long transactions that access entire files.

Chapter 3: Introduction to Database Recovery Protocols

5.17 Recovery Concepts

5.17.1 Recovery Outline and Categorization of Recovery Algorithms

- Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure
- To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**.
- Conceptually, we can distinguish two main techniques for recovery from noncatastrophic transaction failures: **deferred update** and **immediate update**.
 - The **deferred update** techniques
 - do not physically update the database on disk until *after* a transaction reaches its commit point; then the updates are recorded in the database
 - Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains
 - - Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk
 - If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed
 - It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk
 - Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**
 - The **immediate update** techniques
 - the database *may be updated* by some operations of a transaction *before* the transaction reaches its commit point.
 - However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible

- If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back
- In the general case of immediate update, both *undo* and *redo* may be required during recovery.
- This technique, known as the **UNDO/REDO algorithm**, requires both operations during recovery, and is used most often in practice.

5.17.2 Caching (Buffering) of Disk Blocks

- It is convenient to consider recovery in terms of the database disk pages (blocks).
- Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers.
- A **directory** for the cache is used to keep track of which database items are in the buffers
- This can be a table of <Disk_page_address, Buffer_location, ... > entries.
- When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache.
- If it is not, the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item.
- The entries in the DBMS cache directory hold additional information relevant to buffer management.
- Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry, to indicate whether or not the buffer has been modified.
- When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero).
- As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one)
- Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer can also be kept in the directory
- When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1*

- Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet.
- Two main strategies can be employed when flushing a modified buffer back to disk.
 - The first strategy, known as **in-place updating**, writes the buffer to the *same original disk location*, thus overwriting the old value of any changed data items on disk. Hence, a single copy of each database disk block is maintained.
 - The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

5.17.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

- When in-place updating is used, it is necessary to use a log for recovery
- In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk.
- This process is generally known as **write-ahead logging**, and is necessary to be able to UNDO the operation if this is required during recovery
- A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM).
- The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM)

In an UNDO/REDO algorithm, both types of log entries are combined. Additionally, when cascading rollback is possible, `read_item` entries in the log are considered to be UNDO-type entries

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern *when* a page from the database can be written to disk from the cache:

1. If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be used to indicate if a page cannot be written back to disk.
- On the other hand, if the recovery protocol allows writing an updated buffer *before* the transaction commits, it is called **steal**. Steal is used when the DBMS

cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *no-steal rule* means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.

2. If all pages updated by a transaction are immediately written to disk *before* the transaction commits, it is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.
- The deferred update (NO-UNDO) recovery scheme follows a *no-steal* approach.
 - However, typical database systems employ a *steal/no-force* strategy.
 - The *advantage of steal* is that it avoids the need for a very large buffer space to store all updated pages in memory.
 - The *advantage of no-force* is that an updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk, and possibly to have to read it again from disk.
 - To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database.
 - For example, consider the following **write-ahead logging (WAL)** protocol for a recovery algorithm that requires both UNDO and REDO:
 1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log records for the updating transaction—up to this point—have been force-written to disk.
 2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force written to disk.

5.17.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.

- A [checkpoint, list of active transactions] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified

- As a consequence of this, all transactions that have their [commit, T] entries in the log before a [checkpoint] entry do not need to have their WRITE operations redone in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing
- As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.
- The recovery manager of a DBMS must decide at what intervals to take a checkpoint.
- The interval may be measured in time—say, every m minutes—or in the number t of committed transactions since the last checkpoint, where the values of m or t are system parameters
- Taking a checkpoint consists of the following actions:
 1. Suspend execution of transactions temporarily.
 2. Force-write all main memory buffers that have been modified to disk.
 3. Write a [checkpoint] record to the log, and force-write the log to disk.
 4. Resume executing transactions.
- The time needed to force-write all modified memory buffers may delay transaction processing because of step 1
- To reduce this delay, it is common to use a technique called **fuzzy checkpointing**.
- In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish.
- When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing.

5.17.5 Transaction Rollback and Cascading Rollback

- If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction
- If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values (BFIMs)
- The undo-type log entries are used to restore the old values of data items that must be rolled back

- If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back
- Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back; and so on.
- This phenomenon is called **cascading rollback**, and can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules
- Figure 23.1 shows an example where cascading rollback is required.
- The read and write operations of three individual transactions are shown in Figure 23.1(a).
- Figure 23.1(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions.
- The values of data items A, B, C , and D , which are used by the transactions, are shown to the right of the system log entries.
- We assume that the original item values, shown in the first line, are $A = 30, B = 15, C = 40$, and $D = 20$.
- At the point of system failure, transaction T_3 has not reached its conclusion and must be rolled back.
- The WRITE operations of T_3 , marked by a single * in Figure 23.1(b), are the T_3 operations that are undone during transaction rollback.
- Figure 23.1(c) graphically shows the operations of the different transactions along the time axis

(a)

T_1	T_2	T_3
read_item(A)	read_item(B)	read_item(C)
read_item(D)	write_item(B)	write_item(B)
write_item(D)	read_item(D)	read_item(A)
		*
		write_item(A)

		A	B	C	D
		30	15	40	20
*	[start_transaction, T_3]				
*	[read_item, T_3,C]				
*	[write_item, $T_3,B,15,12$]			12	
*	[start_transaction, T_2]				
*	[read_item, T_2,B]				
**	[write_item, $T_2,B,12,18$]			18	
**	[start_transaction, T_1]				
**	[read_item, T_1,A]				
**	[read_item, T_1,D]				
**	[write_item, $T_1,D,20,25$]				25
**	[read_item, T_2,D]				
**	[write_item, $T_2,D,25,26$]				26
	[read_item, T_3,A]				

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3 .

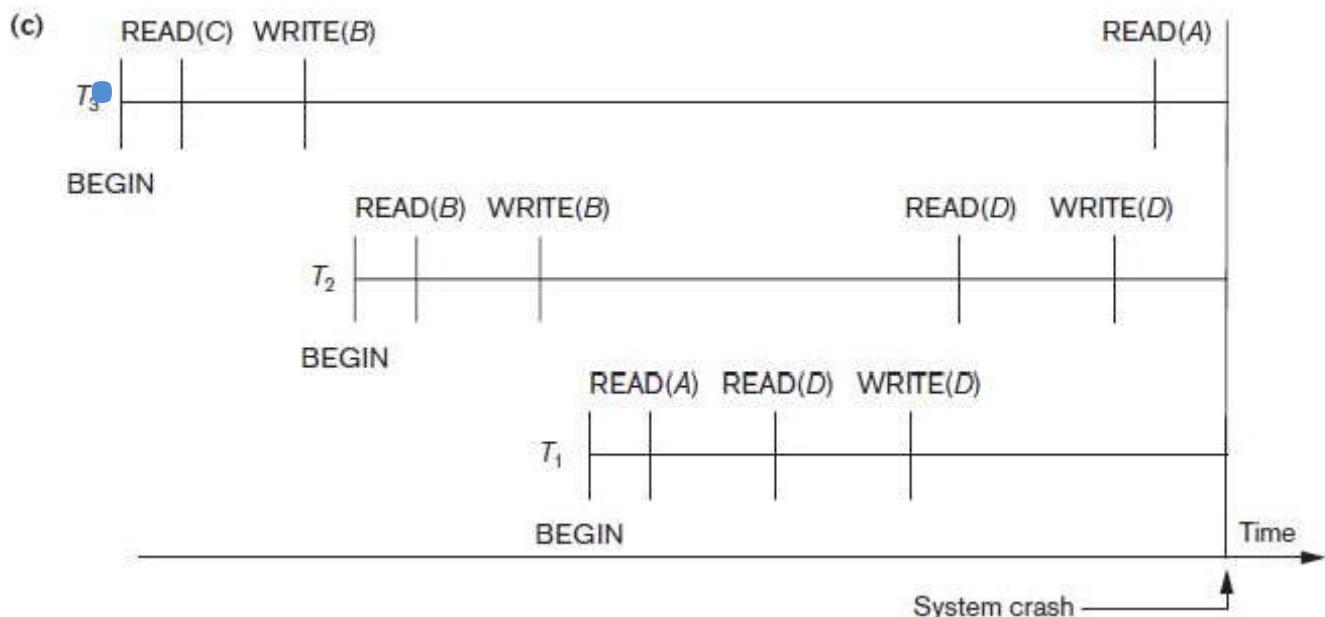


Figure 23.1: Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

5.17.6 Transaction Actions That Do Not Affect the Database

- In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database
- If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete.
- If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action based on these reports that affects the database.
- Hence, such reports should be generated only *after the transaction reaches its commit point*.
- A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

5.18 NO-UNDO/REDO Recovery Based on Deferred Update

- The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.
- During transaction execution, the updates are recorded only in the log and in the cache buffers.
- After the transaction reaches its commit point and the log is forcewritten to disk, the updates are recorded in the database.
- If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way.
- Therefore, only **REDO type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation.
- The **UNDO-type log entries** are not needed since no undoing of operations will be required during recovery.
- We can state a typical deferred update protocol as follows:
 1. A transaction cannot change the database on disk until it reaches its commit point.
 2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

- For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated.
- Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (Recovery using Deferred Update in a Multiuser environment), is as follows:
 - **Procedure RDU_M (NO-UNDO/REDO with checkpoints).** Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint (**commit list**), and the active transactions T_* (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*. The transactions that are active and did not commit are effectively canceled and must be resubmitted.
- The REDO procedure is defined as follows:
 - **Procedure REDO (WRITE_OP).** Redoing a write_item operation WRITE_OP consists of examining its log entry [write_item, T , X , new_value] and setting the value of item X in the database to new_value, which is the after image (AFIM).

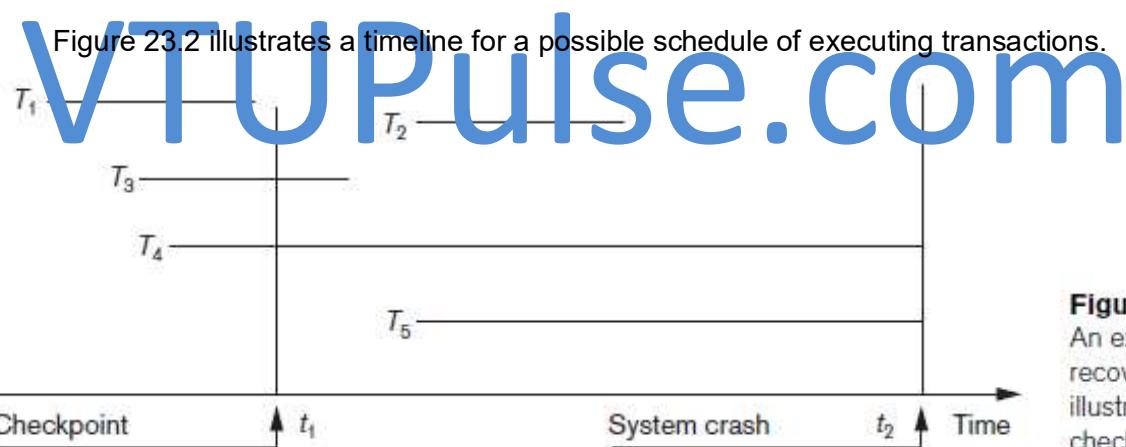


Figure 23.2
An example of a recovery timeline to illustrate the effect of checkpointing.

- Figure 23.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method

(a)

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)

(b)

[start_transaction, T_1]
[write_item, T_1 , D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4 , B, 15]
[write_item, T_4 , A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2 , B, 12]
[start_transaction, T_3]
[write_item, T_3 , A, 30]
[write_item, T_2 , D, 25]

← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Figure 23.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

- The method's main benefit is that transaction operations *never need to be undone*, for two reasons:
 1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
 2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

5.19 Recovery Techniques Based on Immediate Update

- In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point.
- Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction*. This is accomplished by rolling back the transaction and undoing the effect of the transaction's `write_item` operations.
- Therefore, the **UNDO-type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk
- Theoretically, we can distinguish two main categories of immediate update algorithms.
- If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the **UNDO/NO-REDO recovery algorithm**.
- In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **force strategy** for deciding when updated main memory buffers are written back to disk
- If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm**. In this case, the **steal/no-force strategy** is applied.
- When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control.
- The procedure RIU_M (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update (UNDO/REDO recovery).
- **Procedure RIU_M (UNDO/REDO with checkpoints).**
 1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
 2. Undo all the `write_item` operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
 3. Redo all the `write_item` operations of the *committed* transactions from the log, in

the order in which they were written into the log, using the REDO procedure defined earlier.

- The UNDO procedure is defined as follows:

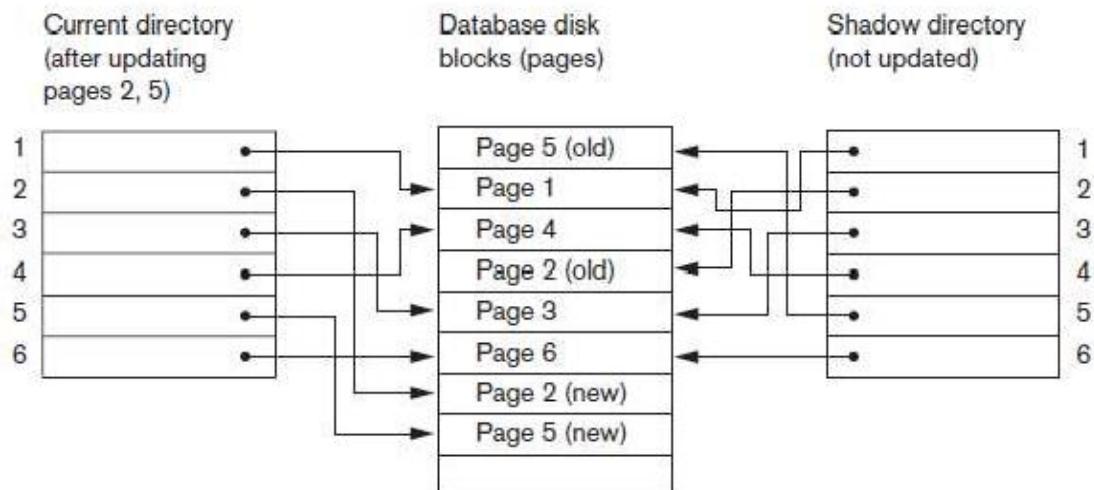
Procedure UNDO (WRITE_OP). Undoing a write_item operation write_op consists of examining its log entry [write_item, T , X , old_value, new_value] and setting the value of item X in the database to old_value, which is the before image (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

5.20 Shadow Paging

- This recovery scheme does not require the use of a log in a single-user environment.
- In a multiuser environment, a log may be needed for the concurrency control method.
- Shadow paging considers the database to be made up of a number of fixedsize disk pages (or disk blocks)—say, n —for recovery purposes
- A **directory** with n entries⁵ is constructed, where the i th entry points to the i th database page on disk.
- The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.
- When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**.
- The shadow directory is then saved on disk while the current directory is used by the transaction.
- During transaction execution, the shadow directory is *never modified*.
- When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten*. Instead, the new page is written elsewhere—on some previously unused disk block.
- The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.
- Figure 23.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept.
- The old version is referenced by the shadow directory and the new version by the current directory.

Figure 23.4

An example of shadow paging.



- To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory.
- The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory.
- Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NOUNDO/ NO-REDO technique for recovery.
- Disadvantage of shadow paging :
 - the updated database pages change location on disk
 - if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant
 - A further complication is how to handle **garbage collection** when a transaction commits
 - Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

5.21 The ARIES Recovery Algorithm

- It is used in many relational database-related products of IBM.
- ARIES uses a steal/no-force approach for writing, and it is based on three concepts:
 - write-ahead logging
 - repeating history during redo, and
 - logging changes during undo.

- **repeating history**, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred*. Transactions that were uncommitted at the time of the crash (active transactions) are undone.
- **logging during undo**, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.
- The ARIES recovery procedure consists of three main steps:
 1. Analysis
 2. REDO
 3. UNDO.
- The **analysis step**
 - identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of the crash
 - The appropriate point in the log where the REDO operation should start is also determined
- The **REDO phase**
 - reapplies updates from the log to the database.
 - Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached
- The **UNDO phase**
 - the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order.
- The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used.
- These tables are maintained by the transaction manager and written to the log during checkpointing.
- In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk.
- Each LSN corresponds to a *specific change* (action) of some transaction.
- Besides the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager.
- When a crash occurs, these tables are rebuilt in the analysis phase of recovery.

- The Transaction Table contains an entry for *each active transaction*, with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction.
- The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.
- **Checkpointing** in ARIES consists of the following: writing a `begin_checkpoint` record to the log, writing an `end_checkpoint` record to the log, and writing *the LSN of the begin_checkpoint record* to a special file.
- This special file is accessed during recovery to locate the last checkpoint information
- After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file.
- The **analysis phase** starts at the `begin_checkpoint` record and proceeds to the end of the log
- The **REDO phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk*.

5.22 Database Backup and Recovery from Catastrophic Failures

- A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure.
- Similarly, the shadow directory must be stored on disk to allow recovery when shadow paging is used.
- The recovery techniques use the entries in the system log or the shadow directory to recover from failure by bringing the database back to a consistent state.
- The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes.
- The main technique used to handle such crashes is a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices.
- In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.
- Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations.

- To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape.

5.23 Assignment Questions

1. Explain properties of a transaction with state transition diagram.
2. Discuss the problems that can occur when concurrent transactions are executed.
3. Discuss the different types of failures. What is meant by catastrophic failure?
4. Discuss the actions taken by the read_item and write_item operations on a database.
5. What is two-phase locking protocol? How does it guarantee serializability?
6. What is a schedule? Explain with example serial, non serial and conflict serializable schedules.
7. Write short notes on
 1. Write ahead log protocol
 2. Time stamp Ordering
 3. Two phase locking protocol
8. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.
9. Describe the wait-die and wound-wait protocols for deadlock prevention.
10. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique?
11. Describe the shadow paging recovery technique.
12. Describe the three phases of the ARIES recovery method.

5.24 Expected Outcome

- ❖ To execute transactions by creating schedules
- ❖ To obtain equivalent and serializable schedules to avoid anomalies.
- ❖ To check whether the given schedule is serializable or not.
- ❖ To study locking protocols
- ❖ To improve resource utilization by applying various forms of locking protocol.

5.25 Further Reading

- ❖ <https://www.smartdraw.com/entity-relationship-diagram/>
- ❖ https://en.wikipedia.org/wiki/Database_normalization
- ❖ www.databasteknik.se/webbkursen/relalg-lecture
- ❖ [https://technet.microsoft.com/en-us/library/bb264565\(v=sql.90\).aspx](https://technet.microsoft.com/en-us/library/bb264565(v=sql.90).aspx)
- ❖ pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/.../Ch16_Overview_Xacts.pdf