DATABASE MANAGEMENT SYSTEM
(Effective from the academic year 2018 -2019)SEMESTER – V

Course Code 18CS53
CIE Marks 40
Number of Contact Hours/Week: 3:2:0
SEE Marks: 40
Total Number of Contact Hours: 50
Exam Hours: 03

CREDITS –4

Course Learning Objectives: This course (18CS53) will enable students to:

- Provide a strong foundation in database concepts, technology, and practice.
- Practice SQL programming through a variety of database problems.
- Demonstrate the use of concurrency and transactions in database
- Design and build database applications for real world problems.

Module 1 **ContactHours:10**

Introduction to Databases: Introduction, Characteristics of database approach, Advantages of using the DBMS approach, History of database applications. **Overview of Database Languages and Architectures:** Data Models, Schemas, and Instances. Three schema architecture and data independence, database languages, and interfaces, The Database System environment. **Conceptual Data Modelling using Entities and Relationships:** Entity types, Entity sets, attributes, roles, and structural constraints, Weak entity types, ER diagrams, examples, Specialization and Generalization.
Textbook 1:Ch 1.1 to 1.8, 2.1 to 2.6, 3.1 to 3.10
RBT: L1,L2, L3

Module 2 **ContactHours:10**

Relational Model: Relational Model Concepts, Relational Model Constraints and relationaldatabase schemas, Update operations, transactions, and dealing with constraint violations.**Relational Algebra:** Unary and Binary relational operations, additional relational operations(aggregate, grouping, etc.) Examples of Queries in relational algebra. **Mapping ConceptualDesign into a Logical Design:** Relational Database Design using ER-to-Relational mapping.**SQL:** SQL data definition and data types, specifying constraints in SQL, retrieval queries inSQL, INSERT, DELETE, and UPDATE statements in SQL, Additional features of SQL.
Textbook 1: Ch4.1 to 4.5, 5.1 to 5.3, 6.1 to 6.5, 8.1; Textbook 2: 3.5
RBT: L1, L2, L3

Module 3 **ContactHours:10**

SQL : Advances Queries: More complex SQL retrieval queries, Specifying constraints as assertions and action triggers, Views in SQL, Schema change statements in SQL. **Database Application Development:** Accessing databases from applications, An introduction to JDBC, JDBC classes and interfaces, SQLJ, Stored procedures, Case study: The internet Bookshop. **Internet Applications:** The three-Tier application architecture, The presentation layer, The Middle Tier
Textbook 1: Ch7.1 to 7.4; Textbook 2: 6.1 to 6.6, 7.5 to 7.7.
RBT: L1, L2, L3

Module 4 **ContactHours:10**

Normalization: Database Design Theory – Introduction to Normalization using Functional and Multivalued Dependencies: Informal design guidelines for relation schema, Functional Dependencies, Normal Forms based on Primary Keys, Second and Third Normal Forms, Boyce-Codd Normal Form, Multivalued Dependency and Fourth Normal Form, Join Dependencies and Fifth Normal Form. **Normalization Algorithms:** Inference Rules, Equivalence, and Minimal Cover, Properties of Relational Decompositions, Algorithms for Relational Database Schema Design, Nulls, Dangling tuples, and alternate Relational Designs, Further discussion of Multivalued dependencies and 4NF, Other dependencies andNormal Forms

10

Textbook 1: Ch14.1 to 14.7, 15.1 to 15.6RBT: L1, L2, L3

Module 5 **ContactHours:10**

Transaction Processing: Introduction to Transaction Processing, Transaction and System concepts, Desirable properties of Transactions, Characterizing schedules based on recoverability, Characterizing schedules based on Serializability, Transaction support in SQL. **Concurrency Control in Databases:** Two-phase locking techniques for Concurrency control, Concurrency control based on Timestamp ordering, Multiversion Concurrency control techniques, Validation Concurrency control techniques, Granularity of Data items and Multiple Granularity Locking. **Introduction to Database Recovery Protocols:** Recovery Concepts, NO-UNDO/REDO recovery based on Deferred update, Recovery techniques based on immediate update, Shadow paging, Database backup and recovery from catastrophicfailures

Textbook 1: 20.1 to 20.6, 21.1 to 21.7, 22.1 to 22.4, 22.7.

RBT: L1, LCourse Outcomes: The student will be able to :

- Identify, analyze and define database objects, enforce integrity constraints on a database using RDBMS.
- Use Structured Query Language (SQL) for database manipulation.
- Design and build simple database systems
- Develop application to interact with databases.

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Fundamentals of Database Systems, Ramez Elmasri and Shamkant B. Navathe, 7th Edition, 2017, Pearson.
2. Database management systems, Ramakrishnan, and Gehrke, 3rd Edition, 2014, McGraw Hill

Reference Books:

1. Silberschatz Korth and Sudharshan, Database System Concepts, 6th Edition, Mc-GrawHill, 2013.
2. Coronel, Morris, and Rob, Database Principles Fundamentals of Design, Implementation and Management, Cengage Learning 2012.

MODULE-1

Introduction to Databases

1.1 Introduction

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

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

- **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database.
- **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS.
- **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.
- **Sharing** a database allows multiple users and programs to access the database simultaneously.
- **Application programs** access the database by sending queries or request for data to the DBMS.A query causes some data to be retrieved

Other important function provided by DBMS include **protecting and maintaining** it over long

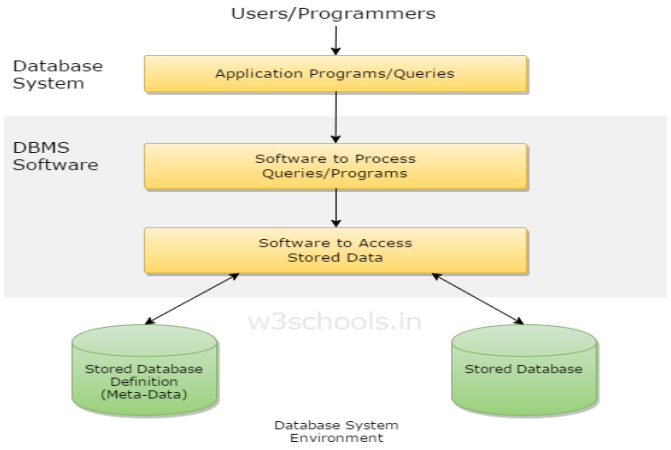


Fig 1.1: A simplified database system environment.

1.3 Characteristics of the Database Approach

1.3.1 Self-Describing Nature of a Database System

1.3.2 Insulation between Programs and Data, and Data Abstraction

1.3.3 Support of Multiple Views of the Data

1.3.4 Sharing of Data and Multiuser Transaction Processing

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

1.3.1 Self-Describing Nature of a Database System

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

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require *changing all programs* that access this file.

By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.**

In object-oriented and object-relational databases users can define operations on data as part of the database definitions.

An **operation** (also called a *function*) is specified in two parts. The ***interface*** (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters).

An **operation** (also called a *function*) is specified in two parts. The ***interface*** (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters).

The **implementation (or method)** of the operation is specified separately and can be changed without affecting the interface.

Application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence.**

The characteristic that allows program-data independence and program-operation independence is called **data abstraction.**

A data model is a type of data abstraction that is used to provide this conceptual representation.

1.3.3 Support of Multiple Views of the Data

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

1.3.4 Sharing of Data and Multiuser Transaction Processing

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

1.4 Actors on the Scene

1.4.1 Database Administrators

1.4.2 Database Designers

1.4.3 End Users

1.4.4 System Analysts and Application Programmers (Software Engineers)

1.4.1 Database Administrators

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

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data.

1.4.3 End Users

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

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

- **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions—**that have been carefully programmed and tested.

  Bank tellers check account balances and post withdrawals and deposits.

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

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

-

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions.

1.5 Workers behind the Scene

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

DBMS system designers and implementers are persons who design and implement the DBMS modules and interfaces as a software package.

Tool developers include persons who design and implement **tools**—the software packages that facilitate database system design and use, and help improve performance. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation .

Operators and maintenance personnel are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.6 Advantages of Using a DBMS

1.6.1 Controlling Redundancy

1.6.2 Restricting Unauthorized Access

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

1.6.4 Permitting Inferencing and Actions Using Rules

1.6.5 Providing Multiple User Interfaces

1.6.6 Representing Complex Relationships Among Data

1.6.7 Enforcing Integrity Constraints

1.6.8 Providing Backup and Recovery

1.6.9 Additional Implications of the Database Approach

1.6.1 Controlling Redundancy

The file based data management systems contained multiple files that were stored in many different locations in a system or even across multiple systems. Because of this, there were sometimes multiple copies of the same file which lead to data redundancy. This is prevented in a database as there is a single database and any change in it is reflected immediately. Because of this, there is no chance of encountering duplicate data.

1.6.2 Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data.

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

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

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

1.6.4 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems.** For example, there may be complex rules in the mini world application for determining when a student is on probation.

1.6.5 Providing Multiple User Interfaces

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

1.6.6 Representing Complex Relationships Among Data

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

1.6.7 Enforcing Integrity Constraints

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

1.6.8 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery.

1.8 When Not to Use a DBMS

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

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

Chapter 2: Database System Concepts and Architecture

A **data model**—a collection of concepts that can be used to describe the structure of a database.

By *structure of a database* we mean the **data types, relationships**, and **constraints** that should hold on the data.
Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

This allows the database designer to specify a set of valid **user-defined operations** that are allowed on the database objects .An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object.

2.1.1 Categories of Data Models

A) High-level or **conceptual data models** provide concepts that are close to the way many users perceive data.
Conceptual data model use concepts such as entities, attributes and relationship

An **entity** represents a real-world object or concept, such as an employee or a project, that is described in the database.

An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary.

A **relationship** among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project. Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used **relational data model.**

B) Low-level or **physical data models** provide concepts that describe the details of how data is stored in the computer.

C) Representational (or **implementation**) **data models,** which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer.
Representational data models represent data by using record structures and hence are sometimes called **record-based data models.**

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used relational data model. As well as in Legacy data models—the **network and hierarchica**l models.

2.1.2 Schemas, Instances, and Database State

In any data model it is important to distinguish between the *description* of the database and the *database itself.*

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

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

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

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

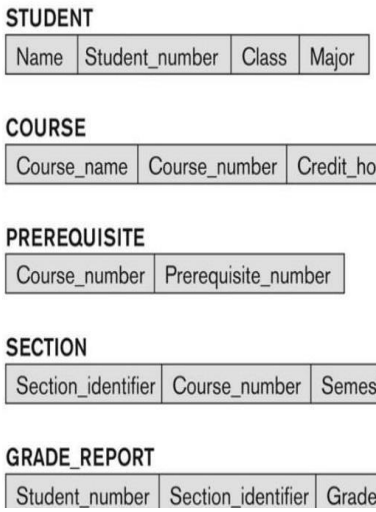


Figure 2.1
Schema diagram for the database in Figure 1.2.

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

SECTION

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

GRADE_REPORT

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320

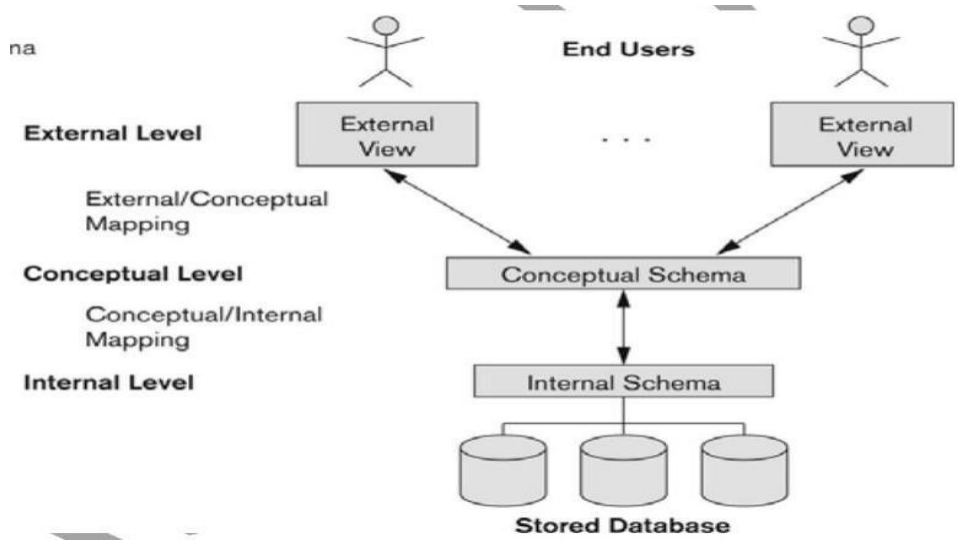
2.2 DBMS Architecture and Data Independence

An architecture for database systems, called the **three-schema architecture** which was proposed to help achieve and visualize the DBMS characteristics.

2.2.1 The Three-Schema Architecture

Proposed to support DBMS characteristics of:

- **Program-data independence.**
- Support of **multiple views** of the data.
- Not explicitly used in commercial DBMS products, but has been useful in explaining database system organization.



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

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types,

relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views.

2.2.2 Data Independence

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

1. **Logical data independence** is the capacity to **change the conceptual schema** without having to change **external schemas or application programs**. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item).

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

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

- **Data definition language (DDL)** is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog. the DDL is used to specify the conceptual schema only.
- **Storage definition language (SDL)** is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

- **View definition language (VDL) is used** to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.
- **Data manipulation language (DML)** The DBMS provides a **data manipulation language (DML)** for these purposes. There are two main types of DMLs. A **high-level** or **nonprocedural DML** can be used on its own to specify complex database operations in a concise manner. A **low-level** or **procedural DML** *must* be embedded in a general-purpose programming language.

2.3.2 DBMS Interfaces

Menu-Based Interfaces for Browsing

These interfaces present the user with lists of options, called **menus,** that lead the user through the formulation of a request. They are often used in **browsing interfaces,** which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces

A forms-based interface displays a **form** to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries.

Graphical User Interfaces

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form.

Natural Language Interfaces

These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema," which is similar to the database conceptual schema. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request.

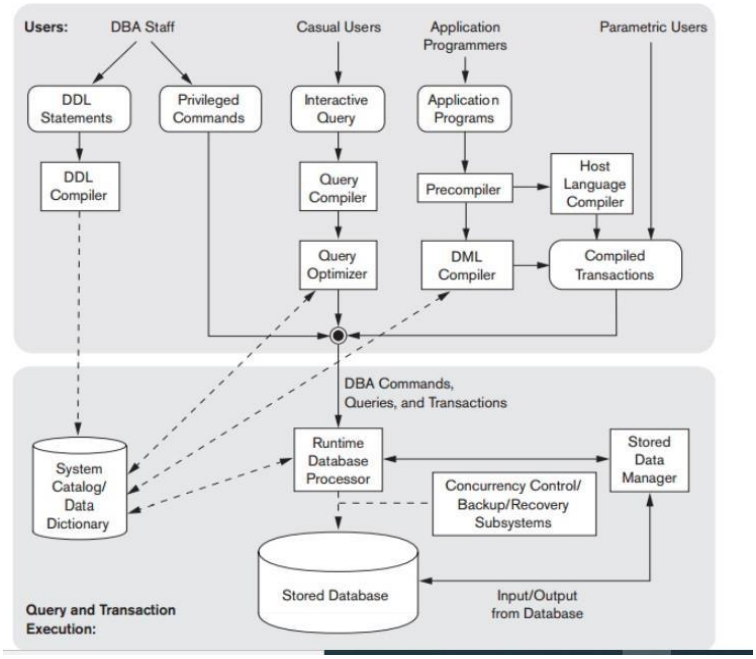
Interfaces for Parametric Users

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

Interfaces for the DBA

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

2.4 The Database System Environment



The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS),** which schedules disk input/output. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file and so on.

The **run-time database processor** handles database accesses at run time; it receives retrieval or update **query compiler** handles high-level queries that are entered interactively. It parses,

analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The **pre-compiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the **DML compiler** for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

2.4.2 Database System Utilities

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

Common utilities have the following types of functions:

1.Loading*:* A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility.

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

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

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

Chapter 3: Data Modeling Using the Entity Relationship Model

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

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

Entity-Relationship (ER) model, which is a popular high-level conceptual data model.

This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications.

2 An Example Database Application

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

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

3 Entity Types, Entity Sets, Attributes, and Keys

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

Entities

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

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

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

Ex:

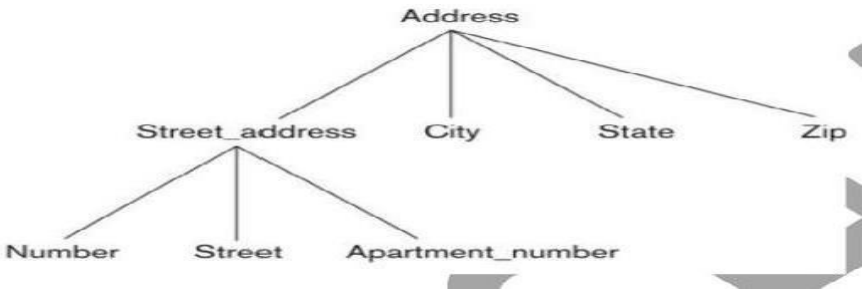
1) The employee entity **E1** has four attributes: **Name, Address, Age, and HomePhone**; their values are **"John Smith," "2311 Kirby, Houston, Texas 77001," "55," and "713-749-2630,"** respectively.

Several types of attributes occur in the ER model:

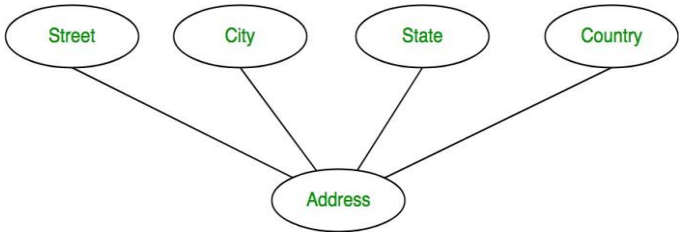
1) ***simple* versus *composite***
2) ***single-valued* versus *multi valued***
3) ***stored* versus *derived.***

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

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



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



2) **Simple** or **Atomic attributes :**
Attributes that are not divisible are called **simple** or **atomic attributes.**
EX: Number, street are simple attributes
Single-valued

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

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

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

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

6) Null Values

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

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

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

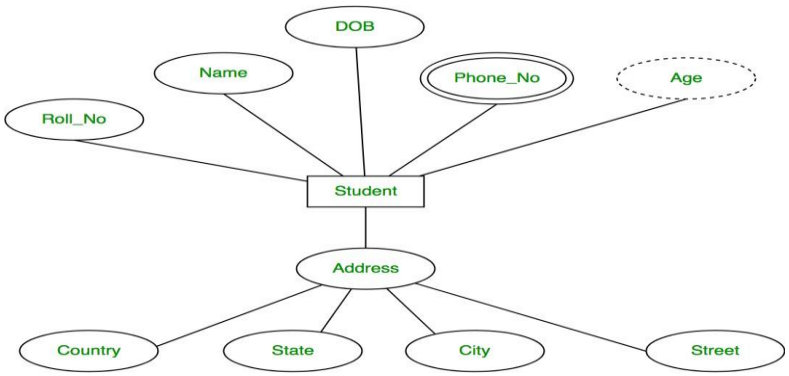
7) Complex Attributes

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

1) For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by {**PreviousDegrees (College, Year, Degree, Field)}**

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

.



Entity Types

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

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

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

The collection of all entities of a particular entity type in the database at any point in time is called an entity set.

The entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current *set of all employee entities* in the database.

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

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

Key Attributes of an Entity Type

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

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

2)For the **Student** entity type, a typical key attribute is **University Student Number**.

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

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

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

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

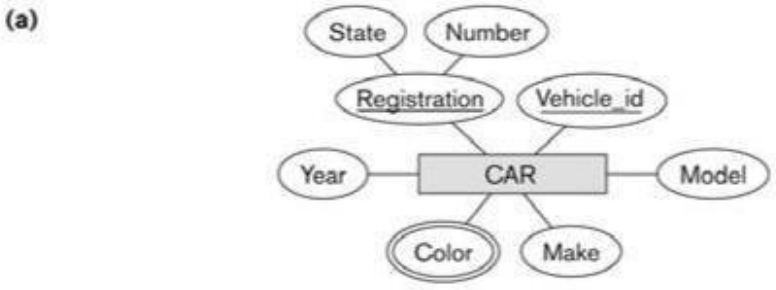


Figure 3.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

(b)

CAR

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

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

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

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

⋮

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

Value Sets (Domains) of Attributes

Each simple attribute of an entity type is associated with a value set **(or** domain **of values), which specifies the set of values that may be assigned to that attribute for each individual entity.** Ex: **The range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70.**

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

Relationships, Relationship Types, Roles, and Structural Constraints

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

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

4.1 Relationship Types, Sets and Instances

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

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

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

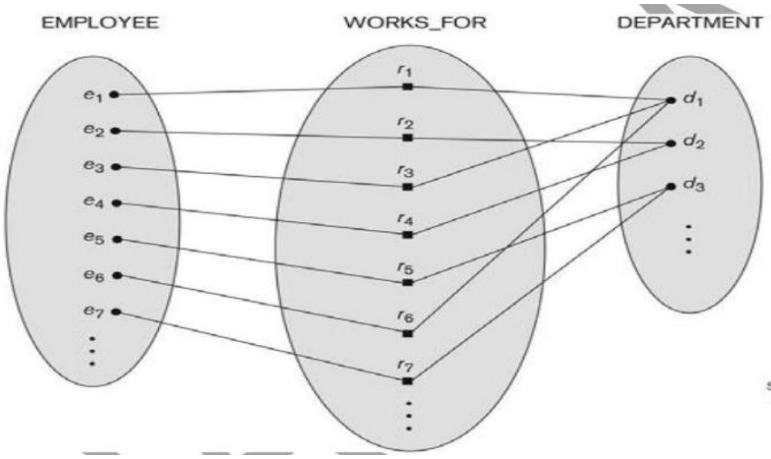


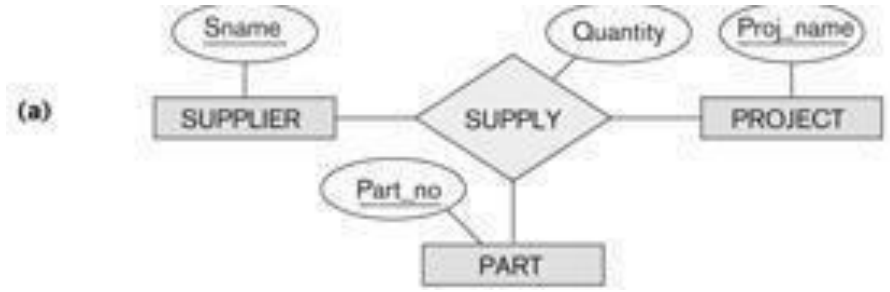
Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

The above figure illustrates this example, where each relationship instance is shown connected to the employee and department entities that participate in . *E*mployees e1, e3, and e6 work for department d1; e2 and e4 work for d2; and e5 and e7 work for d3.

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

4.2 Degree of a Relationship Type

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



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

Relationships as Attributes

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

Role Names and Recursive Relationships

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

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

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

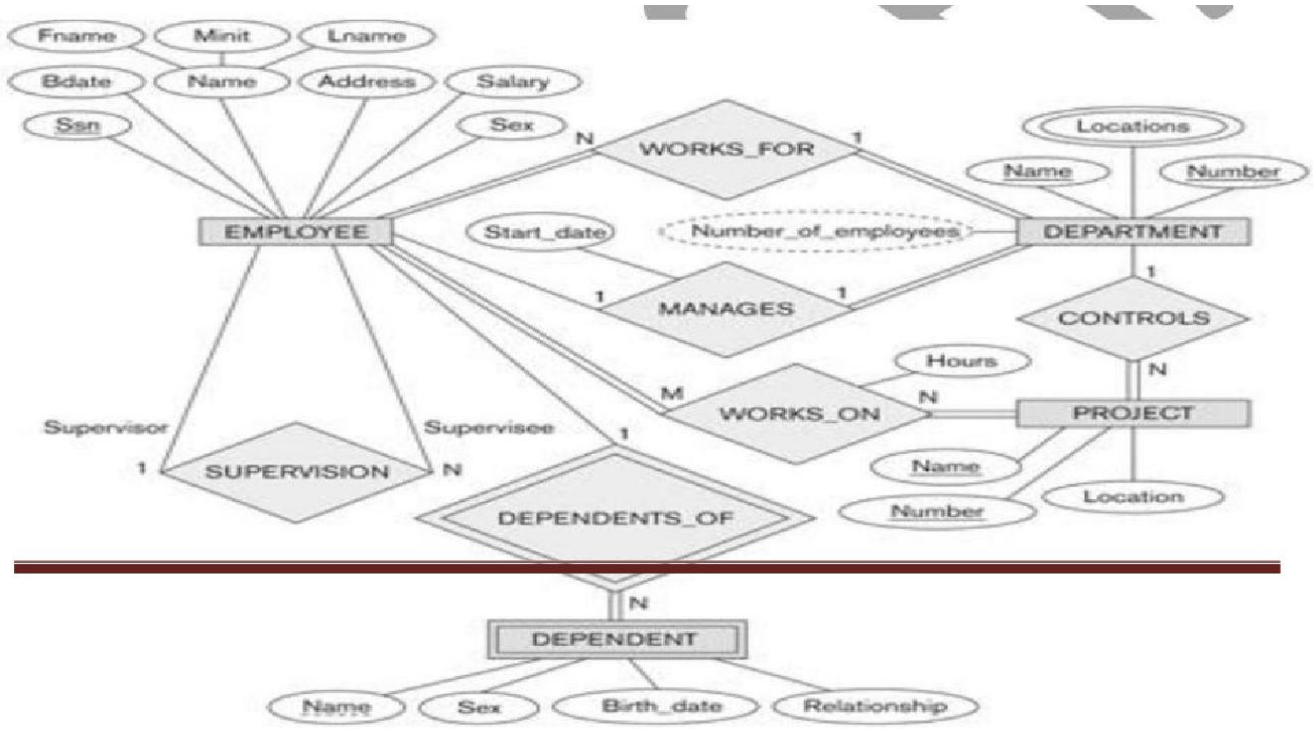


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation

The above figure shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the

EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or boss), and once in the role of *supervisee* (or subordinate).

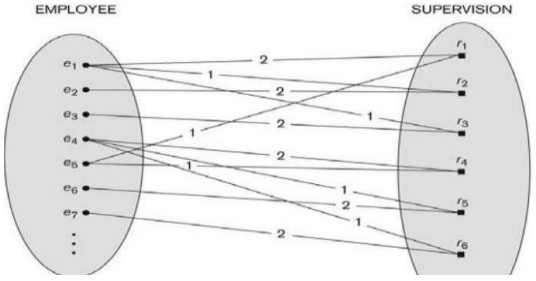


Figure 3.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

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

In the above figure the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e1 supervises e2 and e3; e4 supervises e6 and e7; and e5 supervises e1 and e4.

4.3 Constraints on Relationship Types

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

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

Cardinality Ratios for Binary Relationships

Cardinality ratio :

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

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

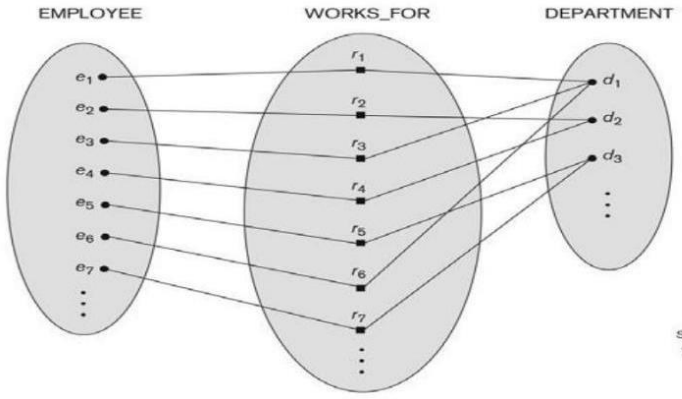
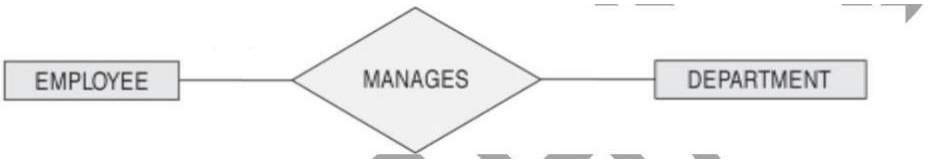


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.



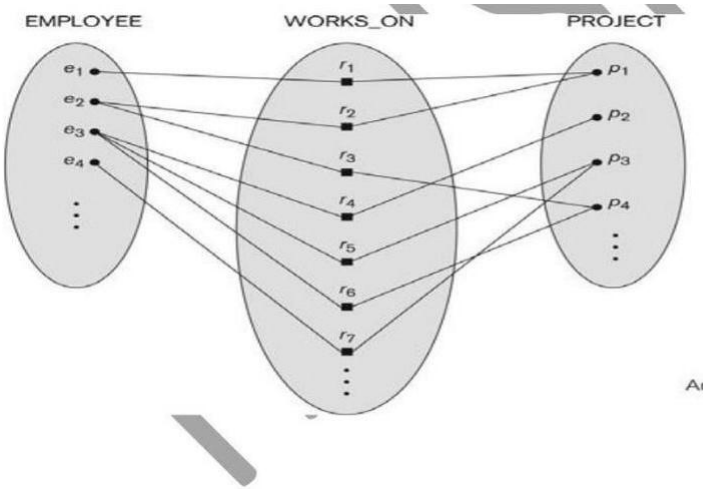


Figure 3.13
An M:N relationship, WORKS_ON.

The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N
An example of a 1:1 binary relationship is MANAGES which relates a department entity to the employee who manages that department. This represents the miniworld constraints that an employee can manage only one department and that a department has only one manager.

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

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

Participation Constraints and Existence Dependencies

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

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

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

structural constraints

cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.In ER diagrams, total participation is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* .

Attributes of Relationship Types

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

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

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the StartDate attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or

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

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

relationship types, the decision as to where a relationship attribute should be placed—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

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

5 Weak Entity Types

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

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

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

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

6 Refining the ER Design for the COMPANY Database

In our example, we specify the following relationship types:

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

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

3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users.

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

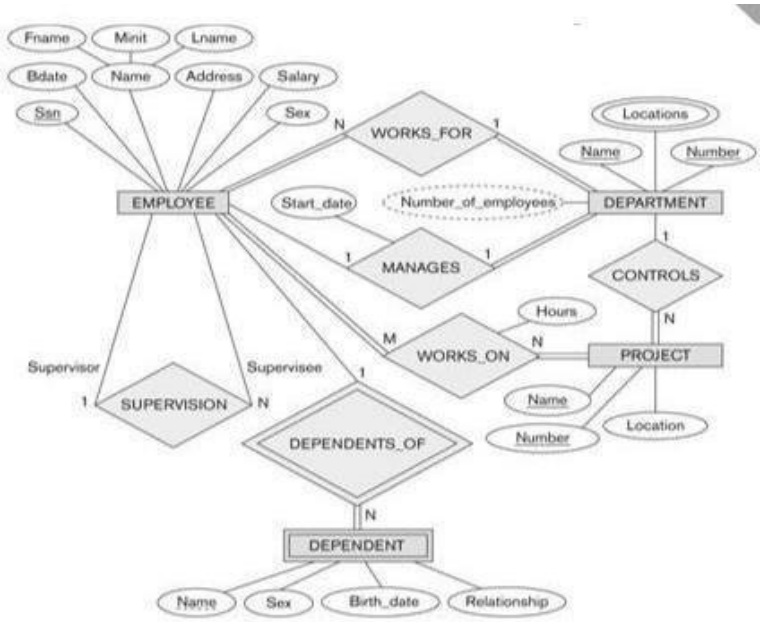
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.

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

Summary of Notation for ER Diagrams

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

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



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

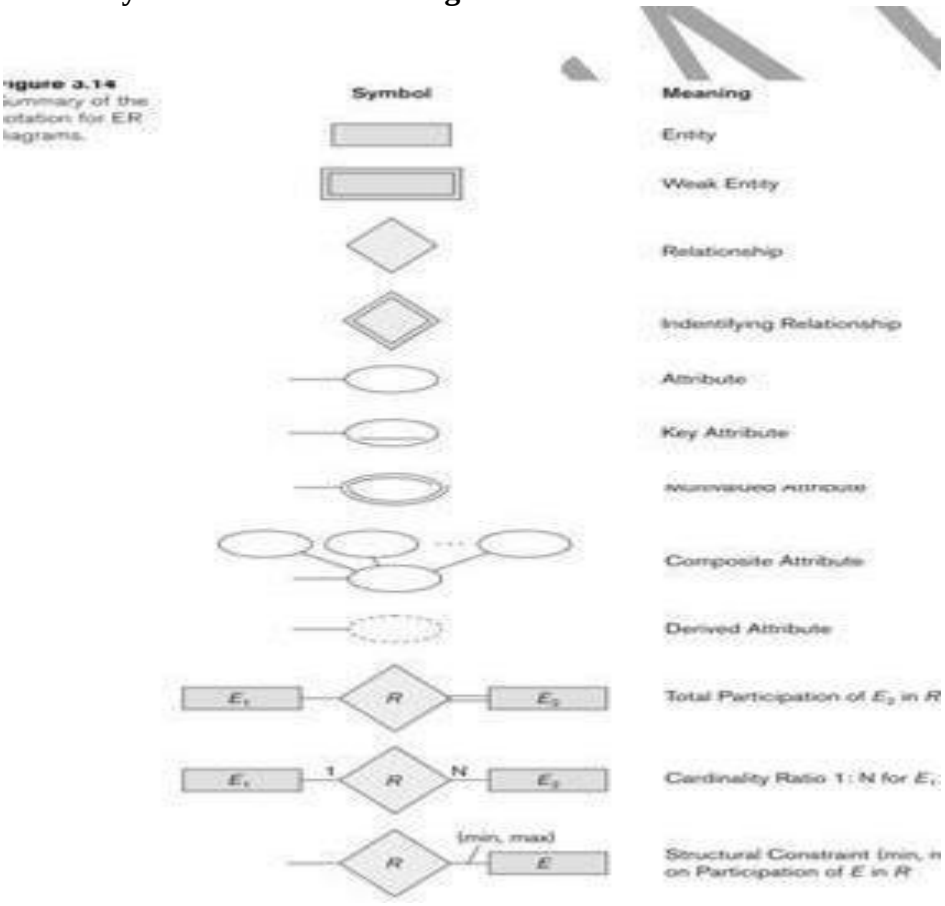
Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with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

In Figure the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT: EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for

DEPARTMENT:EMPLOYEE in WORKS_FOR, and it is M:N for WORKS_ON. The participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

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

Summary of notation for ER diagrams



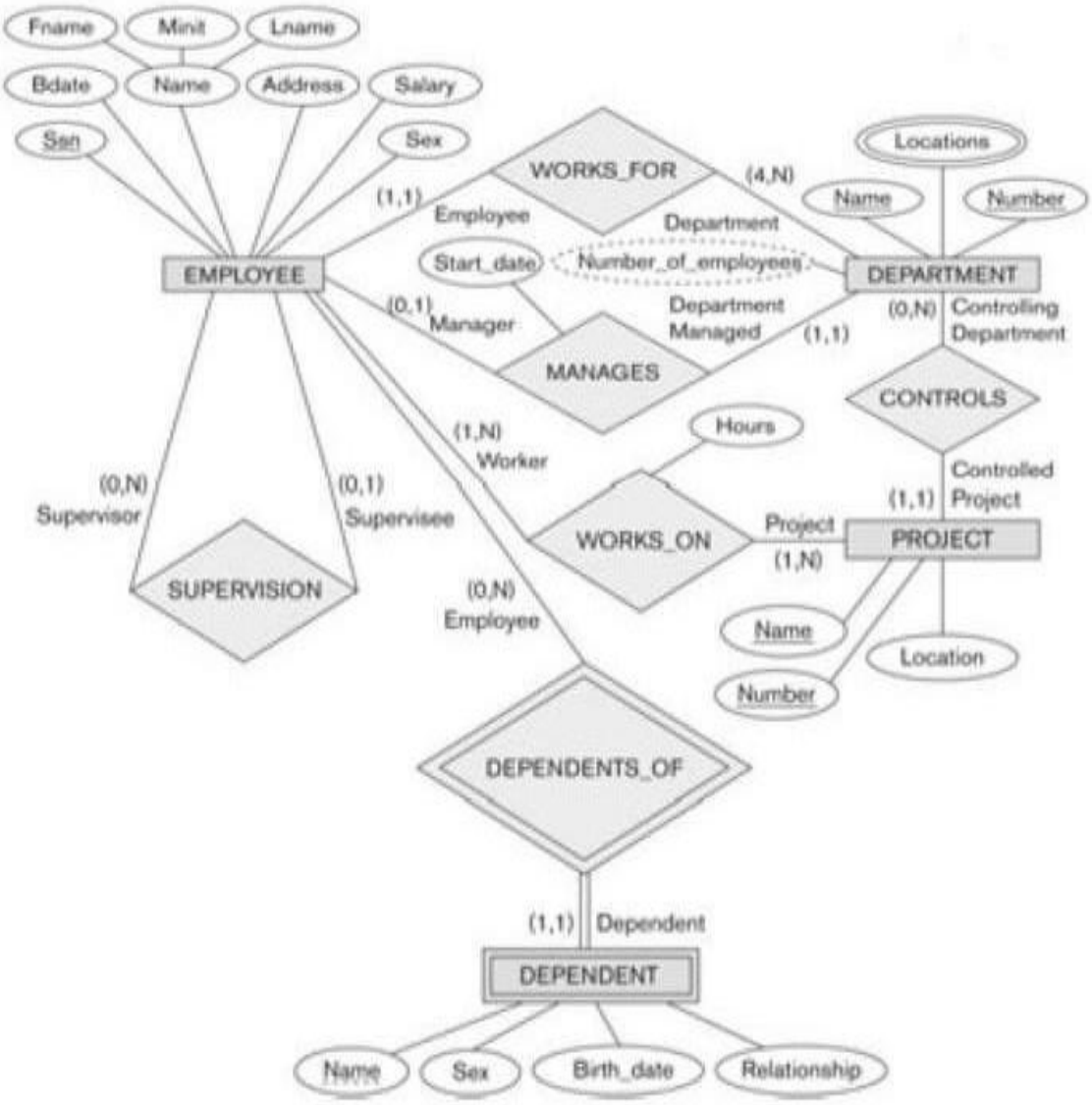


Figure: ER diagram for the company schema, with structural constraints specified using (min,max) notation and role names

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

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

- A **composite attribute** is modeled as a structured domain, as illustrated by the

name attribute of EMPLOYEE.

- A **multivalued attribute** be modeled as a separate class, as illustrated by the LOCATION class in above figure.

2.7 Specialization

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

For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the job type of each employee entity.

We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the method of pay. Figure 8.1shows how we represent a specialization diagrammatically in an EER diagram.The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The subset symbol on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship. Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rectangle representing that subclass. These are called specific attributes (or local attributes) of the subclass.

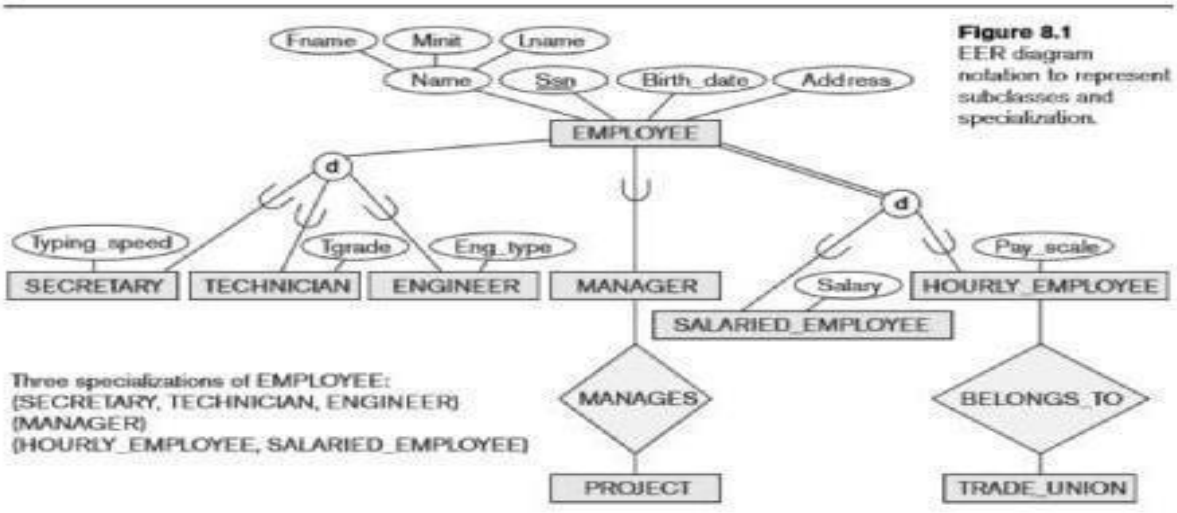


Figure 8.1
EER diagram notation to represent subclasses and specialization.

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

2.8 Generalization

We can think of a reverse process of abstraction in which we suppress the differences among several entity types, identify their common features, and generalize them into a single superclassof which the original entity types are special subclasses.

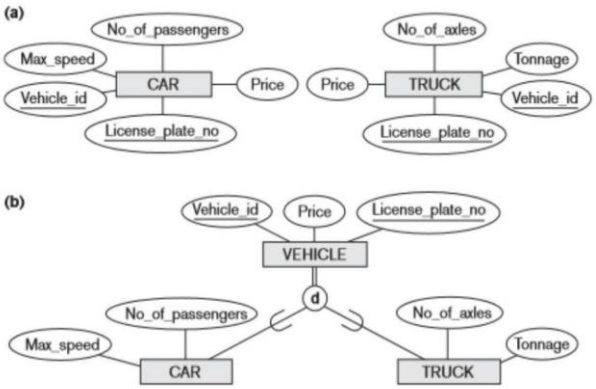


Figure 8.3
Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

For example,consider the entity types CAR and TRUCK shown in Figure 8.3(a).Because they have several common attributes, they can be generalized into the entity type VEHICLE,as shown in Figure 8.3(b).Both CAR and TRUCK are now subclasses of the generalized superclass VEHICLE. We use the term generalization to refer to the process of defining a generalized entity type from the given entity types. Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 8.3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 8.1 we can view EMPLOYEE as a generalization of SECRETARY,TECHNICIAN,and ENGINEER.A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will not use this notation because the decision as to which process is followed in a particular situation is often subjective

MODULE 2

The Relational Data Model and Relational Database Constraints and Relational Algebra

2.1 Relational Model Concepts

- **Domain**: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).

  Examples of domains (some taken from page 147):

  - USA_phone_number: string of digits of length ten
  - SSN: string of digits of length nine
  - Name: string of characters beginning with an upper case letter
  - GPA: a real number between 0.0 and 4.0
  - Sex: a member of the set { female, male }
  - Dept_Code: a member of the set { CMPS, MATH, ENGL, PHYS, PSYC, ... }

  These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, boolean), in a manner analogous to how programming languages have intrinsic data types.

- **Attribute**: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted dom(A).
- **Tuple**: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the miniworld.

  As an example, a tuple for a PERSON entity might be

  { Name --> "Rumpelstiltskin", Sex --> Male, IQ --> 143 }

- **Relation**: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym. (Some database purists would argue that a table is "only" a physical manifestation of a relation.)
- **Relational Schema**: used for describing (the structure of) a relation. E.g., $R(A_1, A_2, ..., A_n)$ says that R is a relation with *attributes* $A_1, ... A_n$. The **degree** of a relation is the number of attributes it has, here *n*.

  Example: STUDENT(Name, SSN, Address)

(See Figure 5.1, page 149, for an example of a STUDENT relation/table having several tuples/rows.)

One would think that a "complete" relational schema would also specify the domain of each attribute.

- **Relational Database**: A collection of **relations**, each one consistent with its specified relational schema.

2.1.2 Characteristics of Relations

Ordering of Tuples: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

Ordering of Attributes: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how E&N define them) make implicit use of the order of the attributes. Hence, E&N view attributes as being arranged as a sequence rather than a set.)

Values of Attributes: For a relation to be in *First Normal Form*, each of its attribute domains must consist of atomic (neither composite nor multi-valued) values. Much of the theory underlying the relational model was based upon this assumption. Chapter 10 addresses the issue of including non-atomic values in domains. (Note that in the latest edition of C.J. Date's book, he explicitly argues against this idea, admitting that he has been mistaken in the past.)

The **Null** value: used for *don't know*, *not applicable*.

Interpretation of a Relation: Each relation can be viewed as a **predicate** and each tuple in that relation can be viewed as an assertion for which that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Example (see Figure 5.1): The first tuple listed means: There exists a student having name Benjamin Bayer, having SSN 305-61-2435, having age 19, etc.

Keep in mind that some relations represent facts about entities (e.g., students) whereas others represent facts about relationships (between entities). (e.g., students and course sections).

The **closed world assumption** states that the only true facts about the miniworld are those represented by whatever tuples currently populate the database.

2.1.3 Relational Model Notation:

- $R(A_1, A_2, ..., A_n)$ is a relational schema of degree n denoting that there is a relation R having as its attributes $A_1, A_2, ..., A_n$.
- By convention, Q, R, and S denote relation names.
- By convention, q, r, and s denote relation states. For example, $r(R)$ denotes one possible state of relation R. If R is understood from context, this could be written, more simply, as r.
- By convention, t, u, and v denote tuples.
- The "dot notation" $R.A$ (e.g., STUDENT.Name) is used to qualify an attribute name, usually for the purpose of distinguishing it from a same-named attribute in a different relation (e.g., DEPARTMENT.Name).

2.2 Relational Model Constraints and Relational Database Schemas

Constraints on databases can be categorized as follows:

inherent model-based: Example: no two tuples in a relation can be duplicates (because a relation is a set of tuples
schema-based: can be expressed using DDL; this kind is the focus of this section.
application-based: are specific to the "business rules" of the miniworld and typically difficult or impossible to express and enforce within the data model. Hence, it is left to application programs to enforce.

Elaborating upon **schema-based constraints**:

2.2.1 Domain Constraints: Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.

2.2.2 Key Constraints: A relation is a *set* of tuples, and each tuple's "identity" is given by the values of its attributes. Hence, it makes no sense for two tuples in a relation to be identical (because then the two tuples are actually one and the same tuple). That is, no two tuples may have the same combination of values in their attributes.

Usually the miniworld dictates that there be (proper) subsets of attributes for which no two tuples may have the same combination of values. Such a set of attributes is called a **superkey** of its relation. From the fact that no two tuples can be identical, it follows that the set of all attributes of a relation constitutes a superkey of that relation.

A **key** is a *minimal superkey*, i.e., a superkey such that, if we were to remove any of its attributes, the resulting set of attributes fails to be a superkey.

Example: Suppose that we stipulate that a faculty member is uniquely identified by *Name* and *Address* and also by *Name* and *Department*, but by no single one of the three attributes mentioned. Then *{ Name, Address, Department }* is a (non-minimal) superkey and each of *{ Name, Address }* and *{ Name, Department }* is a key (i.e., minimal superkey).

Candidate key: any key! (Hence, it is not clear what distinguishes a key from a candidate key.)

Primary key: a key chosen to act as the means by which to identify tuples in a relation.
Typically, one prefers a primary key to be one having as few attributes as possible.

2.2.3 Relational Databases and Relational Database Schemas

A **relational database schema** is a set of schemas for its relations together with a set of **integrity constraints**.

A **relational database state/instance/snapshot** is a set of states of its relations such that no integrity constraint is violated.

2.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

1. Domain constraints

Every domain must contain atomic values(smallest indivisible units) it means composite and multi-valued attributes are not allowed.We perform datatype check here, which means when we assign a data type to a column we limit the values that it can contain. Eg. If we assign the datatype of attribute age as int, we cant give it values other then int datatype.

Domain constraints can be defined as the definition of a valid set of values for an attribute.The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain

Example:

Example:

EID	Name	Phone
01	Bikash Dutta	123456789 234456678

Explanation:

In the above relation, Name is a composite attribute and Phone is a multi-values attribute, so it is violating domain constraint.

2. Key Constraints or Uniqueness Constraints

These are called uniqueness constraints since it ensures that every tuple in the relation should be unique. A relation can have multiple keys or candidate keys(minimal superkey), out of which we choose one of the keys as primary key, we don't have any restriction on choosing the primary key out of candidate keys, but it is suggested to go with the candidate key with less number of

attributes. Null values are not allowed in the primary key, hence Not Null constraint is also a part of key constraint.

Example:

Example:

EID	Name	Phone
01	Bikash	6026526747
02	Paul	7002494274
01	Tuhin	9234567892

3. **Entity integrity constraints**
   The entity integrity constraint states that primary key value can't be null. This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows. A table can contain a null value other than the primary key field.

Example:

EID	Name	Phone
01	Bikash	7002494274
02	Paul	6026526747
NULL	Sony	9234567892

Explanation:
In the above relation, EID is made primary key, and the primary key cant take NULL values but in the third tuple, the primary key is null, so it is a violating Entity Integrity constraints.

4. **Referential Integrity Constraints**

A referential integrity constraint is specified between two tables. In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example:

EID	Name	DNO
01	Divine	12
02	Dino	22
04	Vivian	14

DNO	Place
12	Jaipur
13	Mumbai
14	Delhi

Explanation:
In the above, DNO of the first relation is the foreign key, and DNO in the second relation is the primary key. DNO = 22 in the foreign key of the first table is not allowed since DNO = 22 is not defined in the primary key of the second relation. Therefore Referential integrity constraints is violated here

2.3 Update Operations and Dealing with Constraint Violations.

For each of the *update* operations (Insert, Delete, and Update), we consider what kinds of constraint violations may result from applying it and how we might choose to react.

2.3.1 Insert:

- **Domain constraint violation**: some attribute value is not of correct domain Domain constraint gets violated only when a given value to the attribute does not appear in the corresponding domain or in case it is not of the appropriate datatype.
- **Entity integrity violation**: key of new tuple is **null** On inserting NULL values to any part of the primary key of a new tuple in the relation can cause violation of the Entity integrity constraint.
- **Key constraint violation**: key of new tuple is same as existing one. On inserting a value in the new tuple of a relation which is already existing in another tuple of the same relation, can cause violation of Key Constraints.
- **Referential integrity violation:** On inserting a value in the foreign key of relation 1, for which there is no corresponding value in the Primary key which is referred to in relation 2, in such case Referential integrity is violated.

Ways of dealing with it: reject the attempt to insert! Or give user opportunity to try again with different attribute values.

2.3.2 Delete:

On deleting the tuples in the relation, it may cause only violation of Referential integrity constraints.

Referential Integrity Constraints :
It causes violation only if the tuple in relation 1 is deleted which is referenced by foreign key from other tuples of table 2 in the database, if such deletion takes place then the values in the tuple of the foreign key in table 2 will become empty, which will eventually violate Referential Integrity constraint.
Solutions that are possible to correct the violation to the referential integrity due to deletion are listed below:

1. **Restrict –**
   Here we reject the deletion.
2. **Cascade –**
   Here if a record in the parent table(referencing relation) is deleted, then the corresponding records in the child table(referenced relation) will automatically be deleted.

3. **Set null or set default –**
   Here we modify the referencing attribute values that cause violation and we either set NULL or change to another valid value

2.3.3 Update:

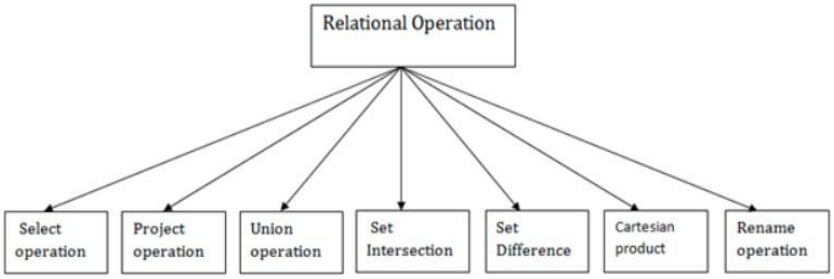
- Key constraint violation: primary key is changed so as to become same as another tuple's□
- Referential integrity violation:
  - foreign key is changed and new one refers to nonexistent tuple
  - primary key is changed and now other tuples that had referred to this one violate the constraint

2.3.4 Transactions: This concept is relevant in the context where multiple users and/or application programs are accessing and updating the database concurrently. A transaction is a logical unit of work that may involve several accesses and/or updates to the database (such as what might be required to reserve several seats on an airplane flight). The point is that, even though several transactions might be processed concurrently, the end result must be as though the transactions were carried out sequentially. (Example of simultaneous withdrawals from same checking account.)

The Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

2.4 Relational Operations:



2.4.1 SELECT (σ)

SELECT operation (**denoted by** σ):

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.
In general, the SELECT operation is denoted by

σ<selection condition>(*R*)

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*.

1) $\sigma_{DNO=4}$**(EMPLOYEE)**

Selects tuples from EMPLOYEE where department no is 4

2 $\sigma_{subject = "database"}$**(Books)**

Selects tuples from books where subject is 'database'.

3)$\sigma_{subject = "database" \text{ and } price = 450}$ **(Books)**

Selects tuples from books where subject is 'database' and 'price' is 450.

4) Selects tuples from Customers where sales is greater than 50000

σ sales>50000(Customers)

5) Selects tuples from employee where dept no is 10 or 20

6) σ**(DNO=4 AND SALARY>25000) OR DNO=5 (EMPLOYEE)**

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 normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain.

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

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

σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000)(EMPLOYEE)

In SQL, the SELECT condition is typically specified in the WHERE clause of a query.

For example, the following operation:

σDno=4 AND Salary>25000 (EMPLOYEE)

would correspond to the following SQL query:

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

PROJECT operation (denoted by π):

The **PROJECT** operation, on the otherhand, selects certain *columns* from the table and discards the other columns.

The general form of the PROJECT operation is

π<attribute list>(*R*)

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

Example:

$$\pi_{\text{NAME, SALARY}}(\text{EMPLOYEE})$$

Select name,salary from employee;

Π salary ,dob,deptno(employee)

The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation.

This is known as **duplicate elimination**. For example, consider the following PROJECT operation:

π Sex, Salary(EMPLOYEE)

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

Π Sex, Salary(EMPLOYEE) would correspond to the following SQL query:

SELECT Sex, Salary **FROM** EMPLOYEE

Figure 7.8 Results of SELECT and PROJECT operations.
(a) $\sigma_{(DNO=4\ AND\ SALARY>25000)\ OR\ (DNO=5\ AND\ SALARY>30000)}(EMPLOYEE)$.
(b) $\pi_{LNAME,\ FNAME,\ SALARY}(EMPLOYEE)$. (c) $\pi_{SEX,\ SALARY}(EMPLOYEE)$

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Duplicate tuples are eliminated by theoperation. **Sequences of operations**: Several operations can be combined to form a *relational algebraexpression* (query)

Example: Retrieve the names and salaries of employees who work in department 4:

Retrieve the names ssn, deptno of employees who salaries are greater than 10000 and less than 30000

Π name,ssn,deptno (**σ(sal>10000 and sal<30000(Employee)))**

π FNAME,LNAME,SALARY **σ** (DNO=4 (EMPLOYEE))

Alternatively, we specify explicit intermediate relations for each step:

$$DEPT4\_EMP \leftarrow \sigma_{DNO=4}(EMPLOYEE)$$

$$P \leftarrow \pi_{FNAME,LNAME,SALARY}(DEPT4\_EMPS)$$

Attributes can optionally be *renamed* in the resulting left-hand-side relation (this may be required for some operations that will be presented later):

$$DEPT4\_EMPS \leftarrow \sigma_{DNO=4}(EMPLOYEE)$$

$$\rho_{(FIRSTNAME,LASTNAME,SALARY)} \leftarrow \pi_{FNAME,LNAME,SALARY}(DEPT4\_EMPS)$$

Figure 7.9 Results of relational algebra expressions. (a) $\pi_{LNAME, FNAME, SALARY}(\sigma_{DNO=5}(EMPLOYEE))$. (b) The same expression using intermediate relations and renaming of attributes.

(a)

FNAME	LNAME	SALARY
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	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

	FIRSTNAME	LASTNAME	SALARY
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

2.5 Relational algebra operation Set theory Operations

Binary operations from mathematical set theory:

1. **UNION**: $R1 \cup R2$,
2. **INTERSECTION**: $R1 \cap R2$,
3. **SET DIFFERENCE**: $R1 - R2$
4. **CARTESIAN PRODUCT**: $R1 \times R2$.

For , , -, the operand relations R1(A1, A2, ..., An) and R2(B1, B2, ..., Bn) must have the same number of attributes, and the domains of corresponding attributes must be compatible; that is, dom(Ai) = dom(Bi) for i=1, 2, ..., n. This condition is called union compatibility. The resulting relation for , , or - has the same attribute names as the first operand relation R1 (by convention).

∪ ∩

∪ ∩

Figure 7.11 Illustrating the set operations union, intersection, and difference. (a) Two union compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∪ INSTRUCTOR. (d) STUDENT − INSTRUCTOR. (e) INSTRUCTOR − STUDENT.

(a)

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNAME	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

(b)

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

(c)

FN	LN
Susan	Yao
Ramesh	Shah

(d)

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

CARTESIAN PRODUCT

$R(A_1, A_2, ..., A_m, B_1, B_2, ..., B_n) \leftarrow R_1(A_1, A_2, ..., A_m) \times R_2(B_1, B_2, ..., B_n)$

A tuple t exists in R for each combination of tuples t1 from R1 and

t_2 from R_2 such that:

$t[A_1, A_2, ..., A_m] = t_1$ and $t[B_1, B_2, ..., B_n] = t_2$
If R1 has n1 tuples and R2 has n2 tuples, then R will have n1* n2 tuples.

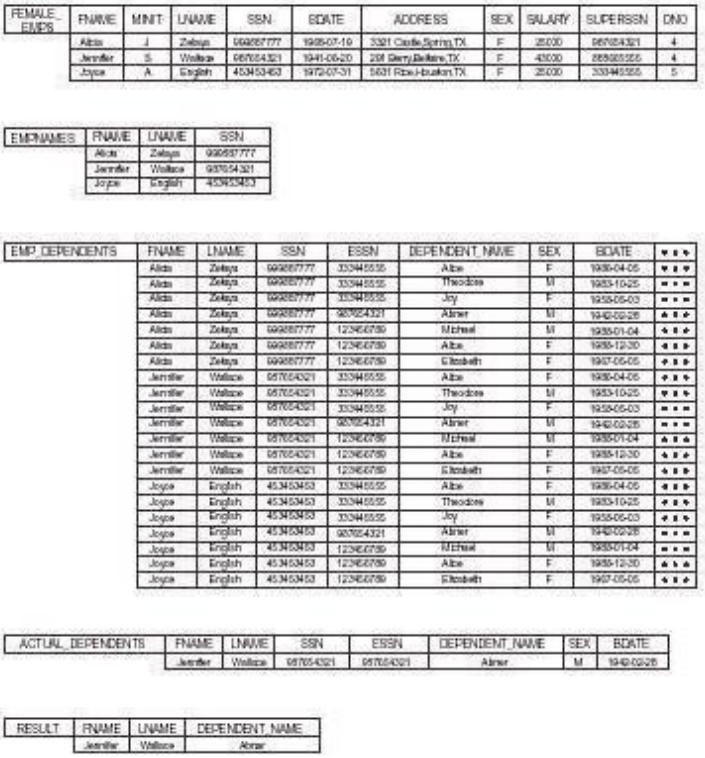
CARTESIAN PRODUCT is a *meaningless operation* on its own. It can *combine related tuples* from two relations *if followed by the appropriate SELECT operation*.

Example: Combine each DEPARTMENT tuple with the EMPLOYEE tuple of the manager.

DEP_EMP ←DEPARTMENT X EMPLOYEE

DEPT_MANAGER ← σ $MGRSSN=SSN$(DEP_EMP)

Figure 7.12 An illustration of the CARTESIAN PRODUCT operation.



FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	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

EMPNAMES	FNAME	LNAME	SSN
	Alicia	Zelaya	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453

EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
	Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05
	Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25
	Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03
	Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28
	Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04
	Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30
	Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05
	Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05
	Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25
	Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28
	Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04
	Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30
	Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05
	Joyce	English	453453453	333445555	Alice	F	1986-04-05
	Joyce	English	453453453	333445555	Theodore	M	1983-10-25
	Joyce	English	453453453	333445555	Joy	F	1958-05-03
	Joyce	English	453453453	987654321	Abner	M	1942-02-28
	Joyce	English	453453453	123456789	Michael	M	1988-01-04
	Joyce	English	453453453	123456789	Alice	F	1988-12-30
	Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05

ACTUAL_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28

RESULT	FNAME	LNAME	DEPENDENT_NAME
	Jennifer	Wallace	Abner

2.6 JOIN Operations

oin is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

THETA JOIN:

Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.

Notation

$R1 \bowtie_\theta R2$

R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don't have anything in common, that is $R1 \cap R2 = \Phi$.

Theta join can use all kinds of comparison operators.

Example 1) $S \leftarrow$ **STUDENT** $\bowtie_{Student.Std = Subject.Class}$ **SUBJECT**

EQUIJOIN

When Theta join uses only **equality** comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

Example of using EQUIJOIN:

Retrieve each DEPARTMENT's name and its manager's name:

$T \leftarrow$ **DEPARTMENT** $\bowtie_{MGRSSN = SSN}$ **EMPLOYEE**

RESULT

π

< - DNAME,FNAME,LNAM (T)

NATURAL JOIN (*):

Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a Natural Join only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.

Natural join acts on those matching attributes where the values of attributes in both the relations are same.

Courses ⋈ HoD

Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

Π Ename, Dept_name (EMPLOYEE ⋈ DEPT)

If the join attributes *have the same names* in both relations, they *need not be specified* and we can write $R \leftarrow R1 * R2$.

Example: Retrieve each EMPLOYEE's name and the name of his/her SUPERVISOR:

SUPERVISOR(SUPERSSN,SFN,SLN) $\leftarrow \pi$ (EMPLOYEE) SSN,FNAME,LNAM

$T \leftarrow$ EMPLOYEE * SUPERVISOR (T)

RESULT $\leftarrow \pi$ FNAME,LNAME,SFN,SLN

Figure 7.14 An illustration of the NATURAL JOIN operation. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

Note: In the *original definition* of NATURAL JOIN, the join attributes were *required* to have the same names in both relations.

(a)

PROJ_DEPT	PNAME	PNUMBER	PLOCATION	DNUM	DNAME	MGRSSN	MGRSTARTDATE
	ProductX	1	Bellaire	5	Research	333445555	1988-05-22
	ProductY	2	Sugarland	5	Research	333445555	1988-05-22
	ProductZ	3	Houston	5	Research	333445555	1988-05-22
	Computerization	10	Stafford	4	Administration	987654321	1995-01-01
	Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
	Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	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

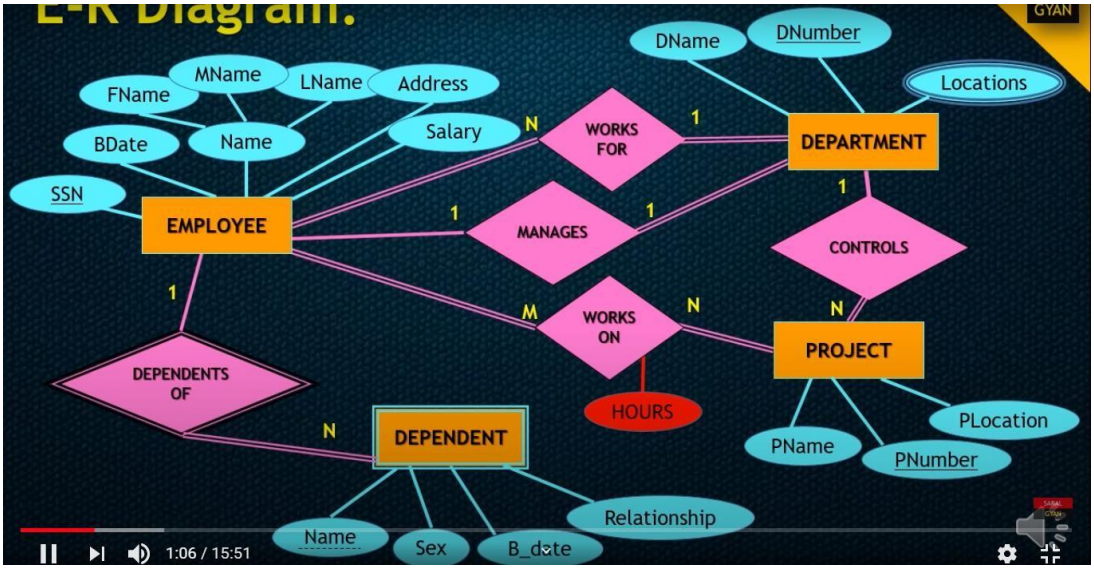
Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

$T \leftarrow EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT$

$RESULT \leftarrow \pi_{FNAME,LNAME,DNAME} (T)$

3.9 Relational Database Design Using ER-to-Relational Mapping

ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram.

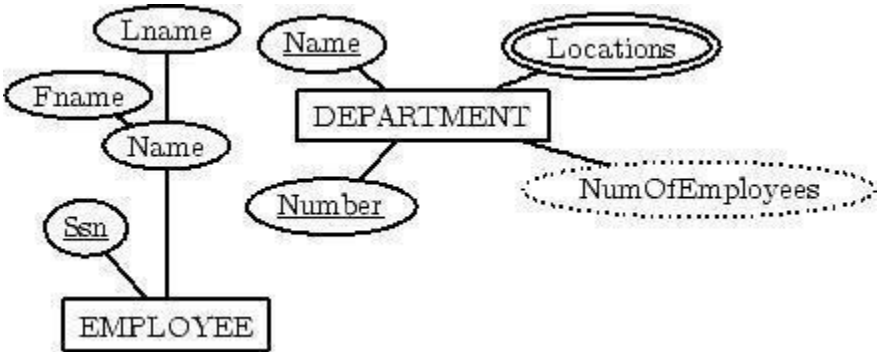


Step 1: For each **regular (strong) entity type** E in the ER schema, create a relation R that includes all the simple attributes of E.

Regular Entity Types

i. For each regular/strong entity type, create a corresponding relation that includes all the simple attributes (includes simple attributes of composite relations)

ii. ii. Choose one of the key attributes as primary § If composite, the simple attributes together form the primary key



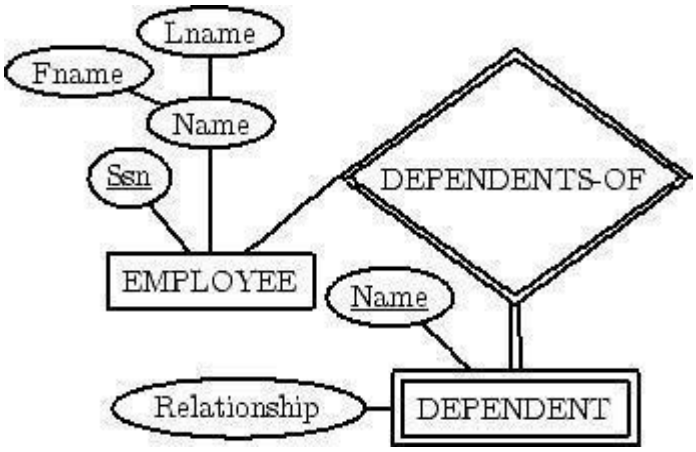
EMPLOYEE

SSN	Lname	Fname

DEPARTMENT

NUMBER	NAME

Step 2: Weak Entity Types i. For each weak entity type, create a corresponding relation that includes all the simple attributes ii. Add as a foreign key all of the primary key attribute(s) in the entity corresponding to the owner entity type iii. The primary key is the combination of all the primary key attributes from the owner and the partial key of the weak entity, if any



DEPENDENT

EMPL-SSN	NAME	Relationship

Step 3: 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. Choose one of the relations, say S, and include the primary key of T as a foreign key in S. Include all the simple attributes of R as attributes of S

DEPARTMENT

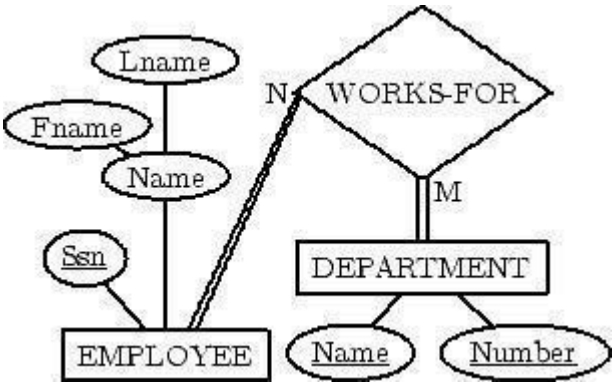
MANAGER-SSN	StartDate

Step 4: For each regular **binary 1:N relationship type** R identify the relation (N) relation S. Include the primary key of T as a foreign key of S. Simple attributes of R map to attributes of S.

EMPLOYEE

SupervisorSSN

Step 5: For each **binary M:N relationship type** R, create a relation S. Include the primary keys of participant relations as foreign keys in S. Their combination will be the primary key for S. Simple attributes of R become attributes of S.



WORKS-FOR

EmployeeSSN	DeptNumber

Step 6: For each **multi-valued attribute A**, create a new relation R. This relation will include an attribute corresponding to A, plus the primary key K of the parent relation (entity type or relationship type) as a foreign key in R. The primary key of R is the combination of A and K.

DEP-LOCATION

Location	DEP-NUMBER

Step 7: For each **n-ary relationship type R**, where n>2, create a new relation S to represent R. Include the primary keys of the relations participating in R as foreign keys in S. Simple attributes of R map to attributes of S. The primary key of S is a combination of all the foreign keys that reference the participants that have cardinality constraint > 1.

Module 3

3.1 MORE COMPLEX SQL QUERIES:

3.1.1 Comparisons Involving NULL and Three-Valued Logic:

- **NULL** ***is used to represent a missing value***, ***but that it usually has one of three different interpretations:***

  (i) value unknown (exists but is not known)

  (ii) value not available (exists but is purposely withheld)

  (iii) attribute not applicable (undefined for this tuple).

- Consider the following examples to illustrate each of the three meanings of NULL.

  - **Unknown value*:*** A particular person has a date of birth but it is not known, so it is represented by NULL in the database.

  - **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.

  - **Not applicable attribute:** An attribute LastCollegeDegree would be NULL for a person who has no college degrees, because it does not apply to that person.

- When a **NULL** is involved in a comparison operation, the result is considered to 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 logic with values ***TRUE or FALSE***.

TABLE 8.1 LOGICAL CONNECTIVES IN THREE-VALUED LOGIC

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

- **Table 8.1** shows the results of three-valued logic.
- ***Rather than using = or <> to compare an attribute value to NULL***, SQL uses **IS** or **IS NOT** key words. **Query 18** illustrates this; its result is shown in **Figure 8.4d**

QUERY 18

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

```
Q18: SELECT FNAME, LNAME
     FROM   EMPLOYEE
     WHERE  SUPERSSN IS NULL;
```

3.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons:

- **Nested Queries** ***are complete select-from-where blocks within the WHERE clause of an outer query***.
- ***SQL has a 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 a 'Smith' involved as manager.

The second selects the project numbers of projects that have a '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.

- The **= ANY** (or **= SOME)** ***operator returns TRUE if the value v is equal to some value in***
- Other operators that can be combined with ANY (or SOME) include **>,>=, <, <=,** and **<**

```
Q16A: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E, DEPENDENT AS D
      WHERE   E.SSN=D.ESSN AND E.SEX=D.SEX AND
              E.FNAME=D.DEPENDENT_NAME;
```

- The keyword **ALL** can also be combined with each of these operators. For example, the comparison condition ***(v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V***.

The following query 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);
```

3.1.3 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***.

- We can understand a correlated query better by considering that the ***nested query is evaluated once for each tuple (or combination of tuples) in the outer query.***

- In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query.

For example,**Q16** may be written as in **Q16A**:

3.1.4 The EXISTS and UNIQUE Functions in SQL:

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

- Query **16B** is an alternative form of query 16 that uses EXISTS.

```
Q16B: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   EXISTS (SELECT *
                      FROM   DEPENDENT
                      WHERE  E.SSN=ESSN AND E.SEX=SEX
                             AND E.FNAME=DEPENDENT_NAME);
```

We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same social security number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple.

- EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query.

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

- On theother hand, ***NOT EXISTS(Q) returns <u>TRUE</u> if there are no tuples in the result of nested queryQ, and it returns <u>FALSE</u> otherwise.***

QUERY 6

Retrieve the names of employees who have no dependents.

```
Q6:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   NOT EXISTS (SELECT  *
                         FROM    DEPENDENT
                         WHERE   SSN=ESSN);
```

- Following query illustrate the use of NOT EXISTS.

We can explain Q6 as follows: 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.

QUERY 7

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

```
Q7:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   EXISTS      (SELECT  *
                          FROM    DEPENDENT
                          WHERE   SSN=ESSN)
             AND
             EXISTS      (SELECT  *
                          FROM    DEPARTMENT
                          WHERE   SSN=MGRSSN);
```

3.1.5 Explicit Sets and Renaming of Attributes in SQL:

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

QUERY 17

Retrieve the social security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17: 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***.
- Hence, the ***AS*** construct can be used to alias both ***attribute*** and ***relation names***, and it can be used in both the ***SELECT*** and ***FROM*** clauses.

3.1.6 Joined Tables in SQL:

- The concept of a ***joined table (or joined relation)*** was incorporated into SQL ***to permit users to specify a table resulting from a join operation in the FROM clause of a query.***

- For example, consider query *Ql,* which retrieves the name and address of every employee who works for the 'Research' department.

```
Q1A: SELECT FNAME, LNAME, ADDRESS
     FROM   (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE  DNAME='Research';
```

second table, DEPARTMENT.

- The concept of a joined table also allows the user to specify different types of join, such as ***NATURAL JOIN*** and various types of ***OUTER JOIN***.

- In a NATURAL JOIN on two relations Rand S, no join condition is specified; an implicit equijoin condition for *each pair of attributes with the same name* from Rand S is created.

- 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.

```
Q1B: SELECT FNAME, LNAME, ADDRESS
     FROM   (EMPLOYEE NATURAL JOIN
             (DEPARTMENT AS DEPT (DNAME, DNO, MSSN, MSDATE)))
     WHERE  DNAME='Research;
```

- ***It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table.***

  This is illustrated by Q2A, which is a different way of specifying query *Q2,* using the concept of a joined table:

```
Q2A: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
     FROM    ((PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)
             JOIN EMPLOYEE ON MGRSSN=SSN)
     WHERE   PLOCATION='Stafford';
```

3.1.7 Aggregate Functions in SQL:

SQL has a number of built-in aggregate functions: COUNT, SUM, MAX, MIN, and AVG.

- ***The COUNT function returns the number of tuples or values as specified in a query.***

- ***The functions SUM, MAX, MIN, and AVG are applied to a set 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***.

- ***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.***

QUERY 19

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

```
Q19: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    EMPLOYEE;
```

QUERY 20

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.

```
Q20: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

QUERIES 21 AND 22

Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21: SELECT  COUNT (*)
     FROM    EMPLOYEE;
Q22: SELECT  COUNT (*)
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DNO=DNUMBER AND DNAME='Research';
```

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

- ***We may also use the COUNT function to count values in a column rather than***

QUERY 23

Count the number of distinct salary values in the database.

```
Q23: SELECT  COUNT (DISTINCT SALARY)
     FROM    EMPLOYEE;
```

- If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated.

- In general, ***NULL values are discarded when aggregate functions are applied to a particular column (attribute).***

- ***We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query***.

Retrieve the names of all employees who have two or more dependents

```
Q5:  SELECT  LNAME, FNAME
     FROM    EMPLOYEE
```

3.1.8 Grouping: The GROUP BY and HAVING Clauses:

- ***In many cases we want to apply the aggregate functions to subgroups of tuples in a relation ,where the subgroups are based on some attribute values.***

  For example, we may want to find the average salary of employees in ***each department*** or the number of employees who work on ***each project***.

- In these cases ***we need to partition the relation into non overlapping subsets (or groups) of tuples***.

- ***Each group (partition) will consist of the tuples that have the same value of some attributes, called the grouping attributets***.

- 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***.

```
Q24:  SELECT    DNO, COUNT (*), AVG (SALARY)
      FROM      EMPLOYEE
      GROUP BY  DNO;
```

In Q24, the EMPLOYEE tuples are partitioned into groups-each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each such group of tuples. **Figure 8.6a** illustrates how grouping works on Q24.It also shows the result of *Q24.*

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 ***satisfy certain conditions.***

- For example, suppose that we want to modify **Query 25** so that only projects with more than two employees appear in the result
- ***SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose***.

QUERY 25

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

```
Q25: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME;
```

- ***HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result ofthe query.***

QUERY 26

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.

```
Q26: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME
```

QUERY 28

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.

```
Q28: SELECT    DNUMBER, COUNT (*)
     FROM      DEPARTMENT, EMPLOYEE
     WHERE     DNUMBER=DNO AND SALARY>40000 AND
               DNO IN    (SELECT    DNO
                          FROM      EMPLOYEE
                          GROUP BY  DNO
                          HAVING    COUNT (*) > 5)
     GROUP BY DNUMBER;
```

Q26.

- A query in SQL can consist of up to six clauses, but only the first two-SELECT and FROM-are mandatory. 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>];
```

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 operator.
- 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. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient.

3.2 SPECIFYING CONSTRAINTS AS ASSERTIONS AND TRIGGERS:

- In SQL, users can specify general constraints - those that do not fall into any of the categories described so far 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.

- 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

```
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.MGRSSN=M.SSN) );
```

for" in SQL, we can write the following assertion:

  - 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.

- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is violated.

- The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition. By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty. Thus, the assertion is violated if the result of the query is not empty.

- Another statement related to CREATE ASSERTION in SQL is CREATE TRIGGER, but triggers are used in a different way.

- Trigger is used to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.

- A typical trigger has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE**, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated.

If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

➢ Rather than offering users only the option of aborting an operation (using ASSERTION) that causes a violation, the DBMS should make the following option available.

- It may be useful to specify a condition that, if violated, causes some user to

  EX: A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.

➢ The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to monitor the database.

➢ Other actions may be specified, such as executing a specific stored procedure or triggering other updates.

Trigger to insert a row in DEPT_LOCATIONS when a row is added to DEPARTMENT

```
create trigger t after insert on DEPARTMENT
for each row
        begin
                insert into DEPT_LOCATIONS values (6, 'TEXAS');
The Trigger 't' is activated after we do the following
        end;
:
```

```
insert into DEPARTMENT values ('accounts', 6,453453453 '1995-04-23');
```

3.3 VIEWS (VIRTUAL TABLES) IN SQL:

3.3.1 Concept of a View in SQL:

➢ A view in SQL terminology is a single table that is derived from other tables.

➢ These other tables could be base tables or previously defined views.

➢ A view does not necessarily exist in physical form; it is considered a virtual table, in contrast to base tables, whose tuples are actually stored in the database.

➢ For example, we may frequently issue the following query that retrieve the employee name and the project names that the employee works on.

```
SELECT FNAME, PNAME
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE (SSN = ESSN AND PNO =
PNUMBER);
```

- Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins.
- We can then 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.

3.3.2 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
- a query to specify the contents of the view.

➢ If none of the view attributes results from applying Aggregate functions (arithmetic operations), we do not have to specify attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

➢ The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 5.2 when applied to the COMPANY database schema of Figure 5.1.

```
V1: CREATE VIEW    WORKS_ON1
    AS SELECT      FNAME, LNAME, PNAME, HOURS
       FROM        EMPLOYEE, PROJECT, WORKS_ON
       WHERE       SSN=ESSN AND PNO=PNUMBER;

V2: CREATE VIEW    DEPT_INFO(DEPT_NAME,NO_OF_EMPS,TOTAL_SAL)
    AS SELECT      DNAME, COUNT (*), SUM (SALARY)
       FROM        DEPARTMENT, EMPLOYEE
       WHERE       DNUMBER=DNO
       GROUP BY    DNAME;
```

Figure 5.2 Two views V1 and V2 specified on the database schema of Figure 5.1

- 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.
- We 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.

We can now specify SQL queries on views V1 and V2 in the same way we specify queries involving base tables. relations.

Retrieve the last name and first name of all employees who work on 'ProjectX'

QV1: SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME =
'ProjectX' ;

- 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.
  - It is the responsibility of the DBMS and not the user to make sure that the view is up to date.
  - If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it.

```
V1A: DROP VIEW WORKS_ON1;
```

3.4 EMBEDDED SQL and DYNAMIC SQL:

- For example, the ***practical relational model has three main constructs***:
  - Attributes and their data types
  - Tuples (rows) and
  - Tables (sets or multisets of records).
- The first problem that may occur is that the "data types of the programming language differ from the attribute data types in the data model".

Hence, it is necessary to have a binding for each host programming language that specifies for each attribute type the compatible programming language types. It is necessary to have a binding for each programming language because different languages have different data types; for example, the data types available in C and JAVA are different, and both differ from the SQL data types.

- Another problem occurs because the "results of most queries are set (distinct elements) or multisets (duplicated elements) of tuples, and each tuple is formed of a sequence of attribute values.
  - In the program, it is often necessary to access the individual data values within individual tuples for printing or processing.
  - Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language.
  - A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple.
  - A cursor or iterator variable is used to loop over the tuples in a query result.
  - Individual values within each tuple are typically extracted into distinct program variables of the appropriate type.

3.4.1 Retrieving Single Tuples with Embedded SQL:

- The programming language in which we embed SQL statements language is called the host language such as C, ADA, COBOL, or PASCAL.
- An embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a preprocessor (or precompiler) can separate embedded SQL statements from the host language code.
- The SQL statements can be terminated by a semicolon (;) or a matching END-EXEC.
- Within an embedded SQL command, we may refer to shared variables which are used in both the C program and the embedded SQL statements.
- Shared variables are prefixed by a colon (:) when they appear in an SQL statement.
  Names of attributes and relations-can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the ":"
- Shared variables are declared within a declare section in the program, as shown in Figure 3.4.1 (lines 1 through 7)

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

Figure 3.4.1: c program variables used in the embedded SQL examples E1 and E2

- A few of the common bindings of C types to SQL types are as follows:

i) The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types long, short, float,and double, respectively.

ii) Fixed-length and varying-length strings (CHAR[i], VARCHAR[i]) in SQL can be mapped to arrays of characters (char [i+ 1], varchar [i+ 1]) in C.

Notice that the only embedded SQL commands in Figure 5.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE. The variables declared in line 6 - SQLCODE and SQLSTATE are used to communicate errors and exception conditions between the database system and the program

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE:

- SQLCODE and SQLSTATE are used by the DBMS to communicate exception or error conditions to the application program.
- The SQLCODE variable is an integer variable.
- After each database command is executed, the DBMS returns a value in SQLCODE:
  - **a)** A value of 0 indicates that the statement was executed successfully by the DBMS.
  - **b)** If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result.
  - **c)** If SQLCODE < 0, this indicates some error has occurred.
- In later versions of the SQL standard, a communication variable called SQLSTATE was added, which is a string of five characters:
  - **a)** A value of "00000" in SQLSTATE indicates no error or exception.
  - **b)** Other values indicate various errors or exceptions. For example, "02000" indicates "no more data" when using SQLSTATE.

```
//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)   }
```

Example of Embedded SQL Programming:

Figure 3.4.2: Program segment E1, a c program segment with embedded SQL

- The program reads (inputs) a social security number value and then retrieves the EMPLOYEE tuple with that social security number from the database via the embedded SQL command.
- The INTO clause (line 5) specifies the program variables into which attribute values from the database are retrieved.
- C program variables in the INTO clause are prefixed with a colon (:).
- Line 7 in El illustrates the communication between the database and the program through the special variable SQLCODE.
- If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions.
- In El a single tuple is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to C program variables in the INTO clause in line 5.

- In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. A ***cursor*** is used to allow tuple-at-a-time processing by the host language program.

3.6 Specifying Queries at Runtime Using Dynamic SQL:

- In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must write a new program, and go through all the steps involved (compiling, debugging, testing, and so on).

- In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) dynamically at runtime

  - For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs.

a) Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations.

```
//Program Segment E3:
0)   EXEC SQL BEGIN DECLARE SECTION ;
1)   varchar sqlupdatestring [256] ;
2)   EXEC SQL END DECLARE SECTION ;
     …
3)   prompt("Enter the Update Command: ", sqlupdatestring) ;
4)   EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)   EXEC SQL EXECUTE sqlcommand ;
     …
```

Dynamic SQL example:

Program segment E3 reads a string that is input by the user (that string should be an SQL update command) into the string variable **sqlupdatestring** in line3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable **sqlcommand.** Line 5 then executes the command.

It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5

EXEC SQL EXECUTE IMMEDIATE: sqlupdatestring ;

in E3) into a single statement by writing:

3.7 DATABASE STORED PROCEDURES AND SQL/PSM(Persistent Stored Modules):

- It is sometimes useful to create database program modules (procedures or functions)-that are stored and executed by the DBMS at the database server. These are historically known as database stored procedures, although they can be functions or procedures.
- Stored procedures are useful in the following circumstances:

a) If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.

b) Executing a program at the server can reduce data transfer and hence communication cost between the client and server in certain situations.

c) These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users.

SELECT C.cid, C.cname, COUNT(*) FROM Customers C,
Orders a WHERE C.cid = O.cid GROUP BY C.cid, C.cname

THE THREE-TIER APPLICATION ARCHITECTURE

3.8.1 Single-Tier and Client-Server Architectures

Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through *dumb terminals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator. The commoditization of the PC and the availability of cheap client computers led to the development of the two-tier

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. In the traditional client server architecture, the client implements just the graphical user interface, and the server. implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustrated in Figure 7.6.

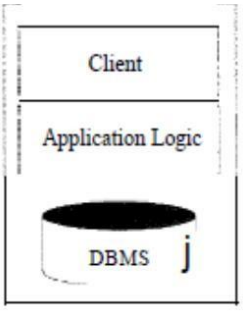


Figure 7.5 A Single-Tier Architecture

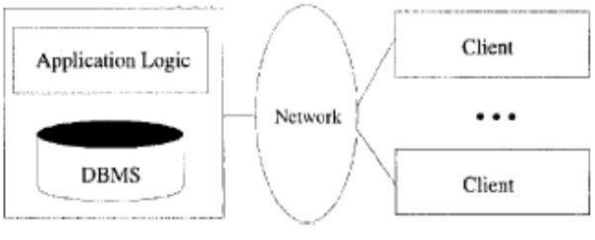


Figure 7.6 A Two-Server Architecture: Thin Clients

3.8.2 THREE TIER ARCHITECTURES

Presentation Tier:

- Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates. Ex: Using The hypertext markup language (HTML)

Middle Tier:

- The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as *C++* or Java. It controls what data needs to be input before an action can be executed,

database query results.

Data Management Tier:

Data-intensive Web applications involve DBMSs, which are the subject of this book.

3.8.3 Advantages of the Three-Tier Architecture

The ***three-tier architecture*** has the following ***advantages***:

a) **Heterogeneous Systems:**
Applications can utilize the strengths of different platforms 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.

b) **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.

c) **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.

d) 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.

Module 4 : NORMALIZATION: DATABASE DESIGN THEORY

Informal design guidelines for relation schemas

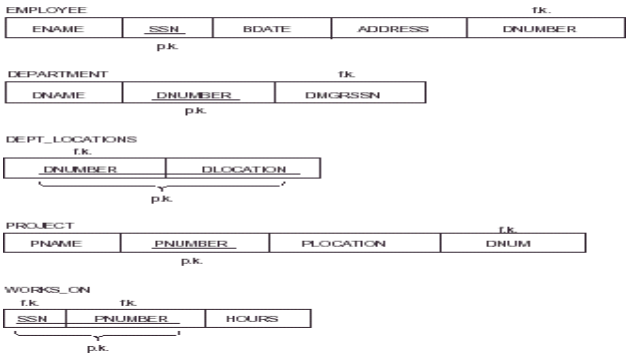
The four informal measures of quality for relation schema

1. Semantics of the attributes
2. Reducing the redundant values in tuples
3. Reducing the null values in tuples
4. Disallowing the possibility of generating spurious tuples

- **Semantics of relations attributes**

Specifies how to interpret the attributes values stored in a tuple of the relation. In other words, how the attribute value in a tuple relate to one another.

Figure 14.1 Simplified version of the COMPANY relational database schema.



Guideline 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Reducing redundant values in tuples. Save storage space and avoid update anomalies.

- Insertion anomalies.
- Deletion anomalies.
- Modification anomalies

Insertion Anomalies

To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for that department that the employee works for, or nulls. It's difficult to insert a new department that has no employee as yet in the EMP_DEPT relation. The only way to do this is to place null values in the attributes for employee. This causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity - not a department entity.

Deletion Anomalies

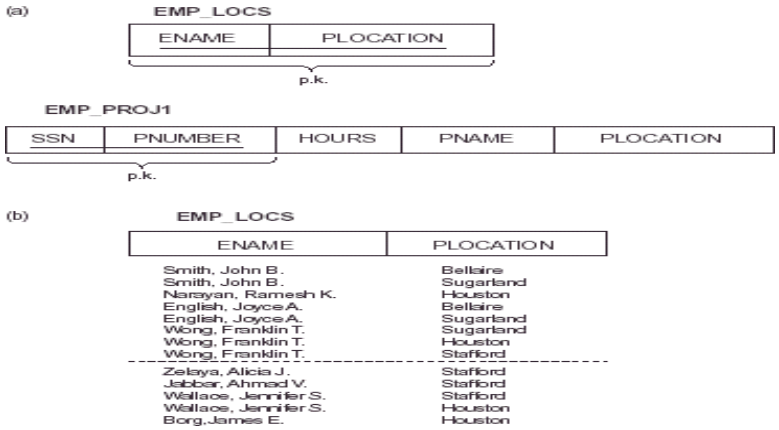
If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

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.

Guideline 2: Design the base relation schemas so that no insertion, deletion, or modification better to have a separate relation, EMP_OFFICE, rather than an attribute OFFICE_NUMBER in EMPLOYEE.

Guideline 3: Avoid placing attributes in a base relation whose values are mostly null. Disallowing spurious tuples. Spurious tuples - tuples that are not in the original relation but generated by natural join of decomposed subrelations. Example: decompose EMP_PROJ into EMP_LOCS and EMP_PROJ1.



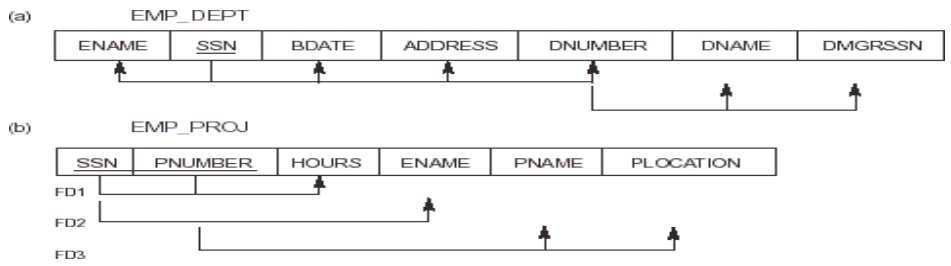
Guideline 4: Design relation schemas so that they can be naturally JOINed on primary keys or foreign keys in a way that guarantees no spurious tuples are generated.

Q Explain Functional Dependencies.

A functional dependency (FD) is a constraint between two sets of attributes from the database.
It is denoted by X →Y We say that "*Y* is functionally dependent on *X*". Also, X is called the left-hand side of the FD. Y is called the right-hand side of the FD. A functional dependency is a property of the semantics or meaning of the attributes, i.e., a property of the relation schema.

Ex Vehicle → State

They must hold on all relation states (extensions) of R. Relation extensions r(R).
A FD X →*Y* is a **full *functional dependency*** if removal of any attribute from X means that the dependency does not hold any more; otherwise, it is **a *partial functional dependency***.
Examples:
1. SSN→ENAME
2. PNUMBER →{PNAME, PLOCATION}
3. {SSN, PNUMBER}→HOURS



FD is property of the relation schema R, not of a particular relation state/instance Let R be a relation schema, where X→R and Y R t1, t2 r, t1[X] = t2[X] t1[Y] = t2[Y]

The FD X⟶Y holds on R if and only if for all possible relations r(R), whenever two tuples of r agree on the attributes of X, they also agree on the attributes of Y.

- the single arrow denotes "functional dependency"
- $X \rightarrow Y$ can also be read as "X determines Y"
- the double arrow denotes "logical implication" .

Q. Explain the Inference Rule (IR) for Functional dependencies

- The Armstrong's axioms are the basic inference rule.
- Armstrong's axioms are used to conclude functional dependencies on a relational database.
- The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.
- Using the inference rule, we can derive additional functional dependency from the initial set.

The Functional dependency has 6 types of inference rule:

1. Reflexive Rule (IR_1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

If $X \supseteq Y$ then $X \rightarrow Y$

Example:

X = {a, b, c, d, e}

Y = {a, b, c}

Example:
Lastname ⊆ { Firstname, Lastname }
then, { **Firstname, Lastname}** → **Lastname**

2. Augmentation Rule (IR_2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

If $X \rightarrow Y$ then $XZ \rightarrow YZ$

Example:

For R(ABCD), **if** $A \rightarrow B$ then $AC \rightarrow BC$

Example:
Regno → { Firstname, Lastname}
then, **Regno, address → { Firstname, Lastname, address }**

3. Transitive Rule (IR_3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Example:
Rollno → **address** and **address** → **Pincode**
then **Rollno** → **Pincode**

4. Union Rule (IR_4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

If X → Y and X → Z then X → YZ

Proof:

1. X → Y (given)
2. X → Z (given)
3. X → XY (using IR_2 on 1 by augmentation with X. Where XX = X)
4. XY → YZ (using IR_2 on 2 by augmentation with Y)
5. X → YZ (using IR_3 on 3 and 4)

Example:
Rollno → **name** and **Rollno** → **address**
then **Rollno** → **name, address**

5. Decomposition Rule (IR_5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If X → YZ then X → Y and X → Z

Proof:

1. X → YZ (given)
2. YZ → Y (using IR_1 Rule)
3. X → Y (using IR_3 on 1 and 2)

Example:
Rollno → **Firstname, Lastname**
then, **Rollno** → **Firstname** and **Rollno** → **Lastname**

6. Pseudo transitive Rule (IR_6)

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

If X → Y and YZ → W then XZ → W

Proof:

1. X → Y (given)
2. WY → Z (given)
3. WX → WY (using IR_2 on 1 by augmenting with W)
4. WX → Z (using IR_3 on 3 and 2)

Example:
Rollno → **name** and **name, marks** →**percentage**
then, **Rollno,marks** → **percentage**

Q What is the need for normalization? Explain the first, second and third normal forms with examples

The purpose of normalization:

- The problems associated with redundant data.
- The identification of various types of update anomalies such as insertion, deletion, and modification anomalies.
- How to recognize the appropriateness or quality of the design of relations.
- The concept of functional dependency, the main tool for measuring the appropriateness of attribute groupings in relations.
- How functional dependencies can be used to group attributes into relations that are in a known normal form.
- How to define normal forms for relations
- How to undertake the process of normalization.
- How to identify the most commonly used normal forms, namely first (1NF), second (2NF), and third (3NF) normal forms, and Boyce-Codd normal form (BCNF).
- How to identify fourth (4NF), and fifth (5NF) normal forms.

The most commonly used normal forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form

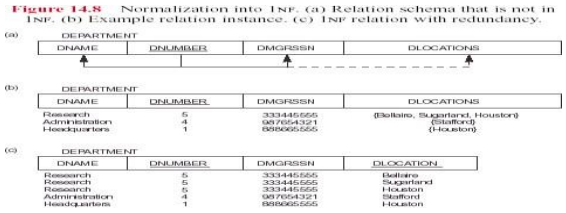
Other Normal Forms

- Fourth Normal Form
- Fifth Normal Form
- Domain Key Normal Form

First Normal Form (1NF)

It states that the domains of attributes 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. Practical Rule: "Eliminate Repeating Groups," i.e., make a separate table for each set of related attributes, and give each table a primary key.

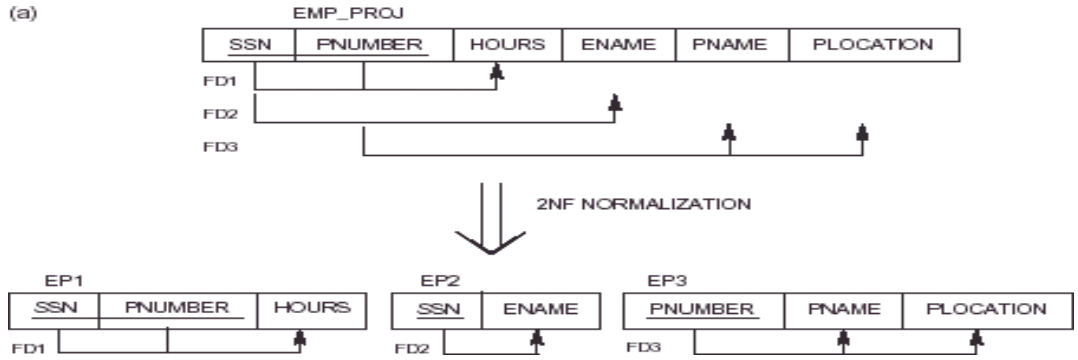
Formal Definition: A relation is in first normal form (1NF) if and only if all underlying simple domains contain atomic values only.



Second Normal Form (2NF)

Second normal form is based on the concept of fully functional dependency. A functional X→Y is a fully functional dependency is removal of any attribute A from X means that the dependency does not hold any more. A relation schema is in 2NF if every nonprime attribute in relation is fully functionally dependent on the primary key of the relation. It also can be restated as: a relation schema is in 2NF if every nonprime attribute in relation is not partially dependent on any key of the relation.Practical Rule: "Eliminate Redundant Data," i.e., if an attribute depends on only part of a multivalued key, remove it to a separate table.

Formal Definition: A relation is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key.

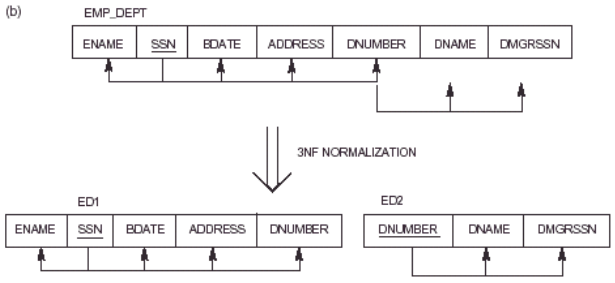


Third Normal Form (3NF)

Third normal form is based on the concept of transitive dependency. A functional dependency X→Y in a relation is a transitive dependency if there is a set of attributes Z that is not a subset of any key of the relation, and both X → Z and ~~Z→~~ Y hold. In other words, a relation is in 3NF if, whenever a functional dependency X→ A holds in the relation, either (a) X is a superkey of the relation, or (b) A is a prime attribute of the relation.

Practical Rule: "Eliminate Columns not Dependent on Key," i.e., if attributes do not contribute to a description of a key, remove them to a separate table.

Formal Definition: A relation is in third normal form (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.



1NF: R is in 1NF iff all domain values are atomic.

2NF: R is in 2 NF iff R is in 1NF and every nonkey attribute is fully dependent on the key.

3NF: R is in 3NF iff R is 2NF and every nonkey attribute is non-transitively dependent on the key.

Boyce Codd Normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X→Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design technical	D134	100
1002	American	Purchasing	D134	600

Functional dependencies in the table above:
emp_id →emp_nationality
emp_dept →{dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:
emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

Emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:
emp_id →emp_nationality
emp_dept →{dept_type, dept_no_of_emp}

Candidate keys:For first table: emp_id For second table: emp_dept For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

Q. Explain multivalued dependency and fourth normal form 4NF with examples.

- **Rules for 4th Normal Form**

For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:

- It should be in the **Boyce-Codd Normal Form**.
- And, the table should not have any **Multi-valued Dependency**.

A table is said to have multi-valued dependency, if the following conditions are true,.

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation (table), it is said to have multi-valued dependency.

Example: Suppose that employees can be assigned to multiple projects. Also suppose that employees can have multiple job skills as shown in database. Try to normalize the" database.

p_no	Proj_no	Skill
1211	1 7	Analysis Design Programming

Solution: In order to normalize it we flatten the database with first normal form as shown below:

Emp_no	Proj_no	Skill
1211	1	Analysis
1211	1	Design
1211	1	Programming
1211	7	Analysis
1211	7	Design
1211	7	Programming

This database shows that Project_No and Skill are independent multi-valued facts about Emp_No that is it contains "a multi-valued dependency. here is a high degree of redundancy that will lead to update problems. Since the database contains MVDs, so it should be decomposed with the help of rule of fourth normal form. Here, the database contain the following MVDs:

Emp_No → Proj_No

Emp_No→ Skill

Here, Proj_No and Skill are independent to each other, so it should be decomposed in to following database according to forth normal form.

EMP _PROJECT (Emp_No, Proj_No)

EMP _SKILL (Emp_No, Skill)

EMP _PROJECT

Emp_no	Proj_no
1211	1
1211	7

EMP _SKILL

Emp_no	Skill
1211	Analysis
1211	Design
1211	Programming

Q. Explain Join dependencies and Fifth normal form/ Projected Normal Form (5NF)

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.

Consider an example of different Subjects taught by different lecturers and the lecturers taking classes for different semesters.

COURSE
SUBJECT
LECTURER
CLASS

SUBJECT	LECTURER	CLASS
Mathematics	Alex	SEMESTER 1
Mathematics	Rose	SEMESTER 1
Physics	Rose	SEMESTER 1
Physics	Joseph	SEMESTER 2
Chemistry	Adam	SEMESTER 1

In above table, Rose takes both Mathematics and Physics class for Semester 1, but she does not take Physics class for Semester 2. In this case, combination of all these 3 fields is required to identify a valid data. Imagine we want to add a new class - Semester3 but do not know which Subject and who will be taking that subject. We would be simply inserting a new entry with Class as Semester3 and leaving Lecturer and subject as NULL. As we discussed above, it's not a good to have such entries. Moreover, all the three columns together act as a primary key, we cannot leave other two columns blank!

Hence we have to decompose the table in such a way that it satisfies all the rules till 4NF and when join them by using keys, it should yield correct record. Here, we can represent each lecturer's Subject area and their classes in a better way. We can divide above table into three - (SUBJECT, LECTURER), (LECTURER, CLASS), (SUBJECT, CLASS)

SUBJECT	LECTURER
Mathematics	Alex
Mathematics	Rose
Physics	Rose
Physics	Joseph
Chemistry	Adam

5NF

CLASS	LECTURER
SEMESTER 1	Alex
SEMESTER 1	Rose
SEMESTER 1	Rose
SEMESTER 2	Joseph
SEMESTER 1	Adam

CLASS	SUBJECT
SEMESTER 1	Mathematics
SEMESTER 1	Physics
SEMESTER 1	Chemistry
SEMESTER 2	Physics

Now, each of combinations is in three different tables. If we need to identify who is teaching which subject to which semester, we need join the keys of each table and get the result.

For example, who teaches Physics to Semester 1, we would be selecting Physics and Semester1 from table 3 above, join with table1 using Subject to filter out the lecturer names. Then join with table2 using Lecturer to get correct lecturer name. That is we joined key columns of each table to get the correct data. Hence there is no lose or new data - satisfying 5NF condition.

Q.What is Functional dependency? Write an algorithm to find minimal cover for a set of Functional dependencies.

DATABASE MANAGEMENT SYSTEM (18CS53)

A functional dependency (FD) is a constraint between two sets of attributes from the database. It is denoted by X →Y. We say that "*Y* is functionally dependent on *X*". Also, X is called the left-hand side of the FD. Y is called the right-hand side of the FD. A functional dependency is a property of the semantics or meaning of the attributes i.e., a property of the relation schema. They must hold on all relation states (extensions) of R. Relation extensions r(R). A FD X *Y* is a full *functional dependency* if removal of any attribute from X means that the dependency does not hold any more; otherwise, it is a *partial functional dependency.*

Examples:

1. SSN→ENAME

2. PNUMBER→{PNAME, PLOCATION}

3. {SSN, PNUMBER}→HOURS

FD is property of the relation schema R, not of a particular relation state/instance The FD X→Y holds on R if and only if for all possible relations r(R), whenever two tuples of r agree on the attributes of X, they also agree on the attributes of Y.

• the single arrow denotes "functional dependency"

• *X*→*Y* can also be read as "*X* determines *Y*".

Finding the Minimal Cover or Canonical cover

- A canonical cover is a simplified and reduced version of the given set of functional dependencies.
- Since it is a reduced version, it is also called as **Irreducible set**.

Given a set of functional dependencies F :

1. Start with F

2. Remove all trivial functional dependencies

3. Repeatedly apply (in whatever order you like), until no changes are possible

• Union Simplification (it is better to do it as soon as possible, whenever possible)

• RHS Simplification

• LHS Simplification

4. Result is the minimal cover

Canonical Cover Algorithm Basic Algorithm

ALGORITHM CanonicalCover (X: FD set)

BEGIN

REPEAT UNTIL STABLE

(1) Where possible, apply Additivity rule (A's axioms) (e.g., A→BC, A→CD becomes A→BCD)

(2) remove "extraneous attributes" from each FD

(e.g., AB→ C, A→ B becomes A→B, B→ C i.e., A is extraneous in AB →C

C) Extraneous Attributes

(1) Extraneous is RHS? e.g.: can we replace A→ BC with A→C? (i.e. Is B extraneous in A→BC?)

(2) Extraneous in LHS ? e.g.: can we replace AB →C with A→ C ? (i.e. Is B extraneous in AB→C?) Simple but expensive test:

1. Replace A →BC (or AB→C) with A→C in F F2 = F - {A→BC} U {A→C} or F - {AB→C} U {A→C}

2. Test if F2+ = F+ ? if yes, then B extraneous

Problem: <u>Find minimal cover for</u>

a) **FD { B→A, D→A, AB→D}**
   STEP 1 : All the above FD's are in Canonical form step1 complete(only one attribute in right hand side)
   STEP2: if AB→D can be replaced by B→D & A→D
   Since B→A by augmenting B (IR2) - BB→AB or B→AB
   However AB→D is given
   Hence by transitive rule(IR3) - B→AB , AB→D we get B→D
   So we can replace AB→D to B→D
   Now we have set equivalent
   E+ ={ B→A, D→A, B→D}
   No reduction possible since all FD's have single attribute on the left hand side. Hence E+ is the minimal cover for the given FD's

b) **G: {A→BCDE, CD→E}**
   STEP 1: Not canonical form A→BCDE (i)
   A→B, A→C, A→D, A→E------(i)

   STEP 2: For CD→E from (i) A→C , A→D

   We can get A→CD and CD→E as A→E (IR3)

   As A→E is redundant we can remove A→E in the set

   Now we have {A→B, A→C, A→D, CD→E } the minimal cover

<u>Closure Of Functional Dependency</u>

The Closure Of Functional Dependency means the complete set of all possible attributes that can be functionally derived from given functional dependency
● If "F" is a functional dependency then closure of functional dependency can be denoted using "{F} + ".
● There are three steps to calculate closure of functional dependency. These are:
Step-1 : Add the attributes which are present on Left Hand Side in the original functional dependency.

Step-2 : Now, add the attributes present on the Right Hand Side of the functional dependency.
Step-3 : With the help of attributes present on Right Hand Side, check the other attributes that can be derived from the other given functional dependencies. Repeat this process until all the possible attributes which can be derived are added in the closure.

Example 1

Consider a relation R(A,B,C,D,E) having below mentioned functional dependencies.
FD1 : A -> BC
FD2 : C -> B
FD3 : D -> E
FD4 : E -> D
Now, calculate the closure of attributes of the relation R. The closures will be:
{A} + = {A, B, C}
{B} + = {B}
{C} + = {B, C}
{D} + = {D, E}
{E} + = {E, D

Example 2

Consider a relation R (A , B , C , D , E , F , G) with the functional dependencies-

$A \rightarrow BC$
$BC \rightarrow DE$
$D \rightarrow F$
$CF \rightarrow G$

Now, let us find the closure of some attributes and attribute sets

Closure of attribute A-

$A^+ = \{ A \}$

$= \{ A , B , C \}$ (Using $A \rightarrow BC$)

$= \{ A , B , C , D , E \}$ (Using $BC \rightarrow DE$)

$= \{ A , B , C , D , E , F \}$ (Using $D \rightarrow F$)

$= \{ A , B , C , D , E , F , G \}$ (Using $CF \rightarrow G$)

Thus,

$$\mathbf{A^+ = \{ A , B , C , D , E , F , G \}}$$

Closure of attribute D-

$D^+ = \{ D \}$

$= \{ D , F \}$ (Using $D \rightarrow F$)

We can not determine any other attribute using attributes D and F contained in the result set.

Thus,

$$\mathbf{D^+ = \{ D , F \}}$$

DATABASE MANAGEMENT SYSTEM (18CS53)

Closure of attribute set {B, C}-

$\{ B , C \}^+ = \{ B , C \}$

$= \{ B , C , D , E \}$ (Using $BC \rightarrow DE$)

$= \{ B , C , D , E , F \}$ (Using $D \rightarrow F$)

$= \{ B , C , D , E , F , G \}$ (Using $CF \rightarrow G$)

Thus,

$$\{ B , C \}^+ = \{ B , C , D , E , F , G \}$$

Closure of attribute set { C,F}-

$\{ C ,F \}^+ = \{ C,F\}$
$= \{ C , F , G\}$

Closure Of Functional Dependency : Calculating Candidate Key

A Candidate Key of a relation is an attribute or set of attributes that can determine the whole relation or contains all the attributes in its closure.

Example-1 : Consider the relation R(A,B,C) with given functional dependencies :
FD1 : A-> B
FD2 : B ->C
{A} + = {A, B, C}
{B} + = {B, C}
{C} + = {C}
Clearly, “A” is the candidate key as, its closure contains all the attributes present in the relation “R”.

Example-2 : Consider another relation R(A, B, C, D, E) having the Functional dependencies :
FD1 : A-> BC
FD2 : C-> B
FD3 : D ->E
FD4 : E ->D
{A} + = {A, B, C}
{B} + = {B}
{C} + = {C, B}
{D} + = {E, D}
{E} + = {E, D}
In this case, a single attribute does is unable to determine all the attribute on its own like in previous example. Here, we need to club two or more attributes to determine the candidate keys.
{A, D} + = {A, B, C, D, E}
{A, E} + = {A, B, C, D, E}
Hence, "AD" and "AE" are the two possible keys of the given relation “R”. Any other combination other than these two would have acted as extraneous attributes

Closure Of Functional Dependency : Key Definitions

1. **Prime Attributes** : Attributes which are indispensable part of candidate keys. For example : "A, D, E" attributes are prime attributes in above example
2. **Non-Prime Attributes** : Attributes other than prime attributes which does not take part in formation of candidate keys.
3. **Extraneous Attributes** : Attributes which does not make any effect on removal from candidate key.

For example : Consider the relation R(A, B, C, D) with functional dependencies :
FD1 : A-> BC
FD2 : B ->C
FD3 : D ->C
Here, Candidate key can be "AD" only. Hence,
Prime Attributes : A, D.
Non-Prime Attributes : B, C
Extraneous Attributes : B, C(As if we add any of the to the candidate key, it will remain unaffected). Those attributes, which if removed does not affect closure of that set.

Equivalence of Functional Dependencies

Two different sets of functional dependencies for a given relation may or may not be equivalent. If FD1 and FD2 are the two sets of functional dependencies following with below 3 cases are possible, then FD's are equivalent.

- If FD1 can be derived from FD2, we can say that FD2 ⊃ FD1.
- If FD2 can be derived from FD1, we can say that FD1 ⊃ FD2.
- If above two cases are true, FD1=FD2.

Q. Let us take an example to show the relationship between two FD sets.
A relation R(A,B,C,D) having two FD sets FD1 = {A->B, B->C, AB->D} and FD2 = {A->B, B->C, A->C, A->D}

Step 1. Checking whether all FDs of FD1 are present in FD2

- A->B in set FD1 is present in set FD2.
- B->C in set FD1 is also present in set FD2.
- AB->D in present in set FD1 but not directly in FD2 but we will check whether we can derive it or not. For set FD2, (AB) + = {A,B,C,D}. It means that AB can functionally determine A, B, C and D. So AB->D will also hold in set FD2. As all FDs in set FD1 also hold in set FD2, FD2 ⊃ FD1 is true.

Step 2. Checking whether all FDs of FD2 are present in FD1

- A->B in set FD2 is present in set FD1.
- B->C in set FD2 is also present in set FD1.
- A->C is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, (A) + = {A,B,C,D}. It means that A can functionally determine A, B, C and D. SO A->C will also hold in set FD1.
- A->D is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, (A) + = {A,B,C,D}. It means that A can functionally determine A, B, C and D. SO A->D will also hold in set FD1. As all FDs in set FD2 also hold in set FD1, FD1 ⊃ FD2 is true.

Step 3. As FD2 ⊃ FD1 and FD1 ⊃ FD2 both are true FD2 =FD1 is true.
These two FD sets are semantically equivalent.

Let us take another example to show the relationship between two FD sets. A relation R2(A,B,C,D) having two FD sets FD1 = {A->B, B->C,A->C} and FD2 = {A->B,

B->C, A->D}

Step 1. Checking whether all FDs of FD1 are present in FD2
● A->B in set FD1 is present in set FD2.
● B->C in set FD1 is also present in set FD2.
● A->C is present in FD1 but not directly in FD2 but we will check whether we can derive it or not. For set FD2, (A) + = {A,B,C,D}. It means that A can functionally determine A, B, C and D. SO A->C will also hold in set FD2.
As all FDs in set FD1 also hold in set FD2, FD2 ⊃ FD1 is true.

Step 2. Checking whether all FDs of FD2 are present in FD1
● A->B in set FD2 is present in set FD1.
● B->C in set FD2 is also present in set FD1.
● A->D is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, (A) + = {A,B,C}. It means that A can't functionally determine D. SO A->D will not hold in FD1.
As all FDs in set FD2 do not hold in set FD1, FD2 ⊄ FD1.

Step 3. In this case, FD2 ⊃ FD1 and FD2 ⊄ FD1, these two FD sets are not semantically equivalent.

Q . Properties of Relational Decompositions

Decomposition of a relation is done when a relation in relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is **lossless join** as well as **dependency preserving.**

Algorithm 11.1 Testing for the lossless (nonadditive) join property

Input: A universal relation R, a decomposition DECOM P = f R_1 ; R_2; : : : ; R_mg of R, and a set F of functional dependencies.

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.
3. For each row i{ for each column j { if R_i includes attribute A_j Then set S(i; j) := a_j
4. Repeat the following loop until a complete loop execution results in no changes to S

{For each X →Y in F { for all rows in S which has 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 "α" symbol for the column, set the other rows to the same "α" symbol in the column. If no "α" symbol exists for the attribute in any of the rows, choose one of the β symbols that appear in one of the rows for the attribute and set the other rows to that same symbol β in the column

4. If a row is made up entirely of "α" symbols, then the decomposition has the lossless join property; otherwise it does not.

Problem

DATABASE MANAGEMENT SYSTEM (18CS53)

Consider R(A,B,C,D,E) which is decomposed into R1=(A,B,C) AND R2=(C,D,E)with the FD A→B C , CD→E ,B→D, E→A show that the above decomposition of schema R is not a loseless

	A	B	C	D	E
R1	α	α	α		
R2			α	α	α

A→B C no change

CD→E no change

E→A no change The above table does not contain any of the rows completely filled with symbol α so it is not loseless.

Problem

Example:

R = {SSN; ENAME; PNUM BER; PN AM E; PLOCATION; HOURS}

FD {SSN →ENAM E; PNUM BER →{PN AM E; PLOCATION } ,{SSN; PNUMBER}→HOURS}

DECOMP= { R_1 ; R_2; R_3 }

R_1 = {SSN; EN AME}

R_2 = {PNUMBER; PNAME; PLOCATION}

R_3 = {SSN; PNUMBER; HOURS}

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	**α**	**α**				
R_2			**α**	**α**	**α**	
R_3	**α**		**α**			**α**

a) initial matrix

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	**α**	**α**				
R_2			**α**	**α**	**α**	

R_3	**α**	**α**	**α**		**α**

b) after applying FD - SSN →EN AM E

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	**α**	**α**				
R_2			**α**	**α**	**α**	
R_3	**α**	**α**	**α**	**α**	**α**	**α**

c)after applying FD PNUM BER →{PN AM E; PLOCATION}

The last row is all α symbols hence it is loseless join decomposition.

Note :Practice all other Solved problems in Class

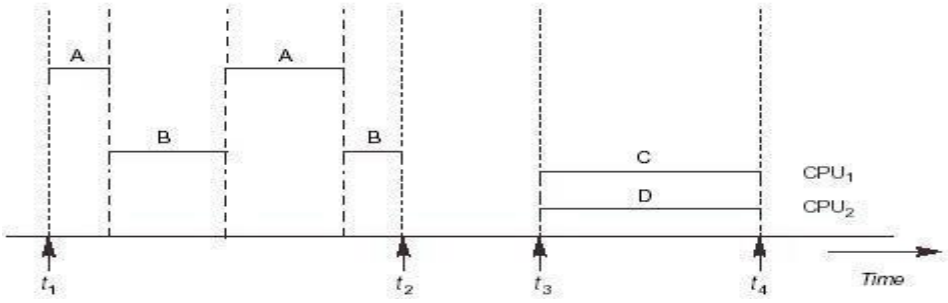
Module 5

5.1 Introduction to Transaction Processing

Single-User Versus Multiuser Systems

- A DBMS is **single-user** id at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.
- Most DBMS are multiuser (e.g., airline reservation system).
- *Multiprogramming operating systems* allow the computer to execute multiple programs (or processes) at the same time (having one CPU, concurrent execution of processes is actually interleaved).
- If the computer has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

Figure 19.1 Interleaved processing versus parallel processing of concurrent transactions.



5.2 Transactions, Read and Write Operations

- A *transaction* is a logical unit of database processing that includes one or more database access operations (e.g., insertion, deletion, modification, or retrieval operations). The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**

- *Read-only transaction* - do not changes the state of a database, only retrieves data.
- The basic database access operations that a transaction can include are as follows:

read_item(X): reads a database item *X* into a program variable *X*.

- *write_item(X):* writes the value of program variable *X* into the database item named *X*.

Executing a read_item(*X*) command includes the following steps:

3. Find the address of the disk block that contains item *X*.

4. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
5. Copy item *X* from the buffer to the program variable named *X*.

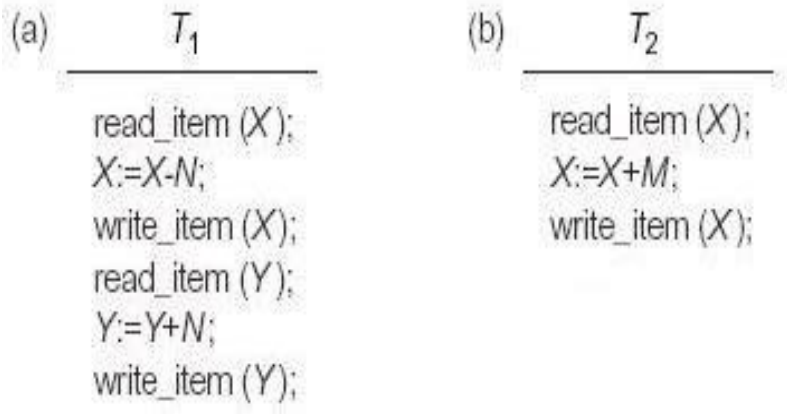
Executing a write_item(*X*) command includes the following steps:
Find the address of the disk block that contains item *X*.

6. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
7. Copy item *X* from the buffer to the program variable named *X*.

Executing a write_item(*X*) command includes the following steps:

6. Find the address of the disk block that contains item *X*.

7. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
8. Copy item *X* from the program variable named *X* into its correct location in the buffer.
9. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

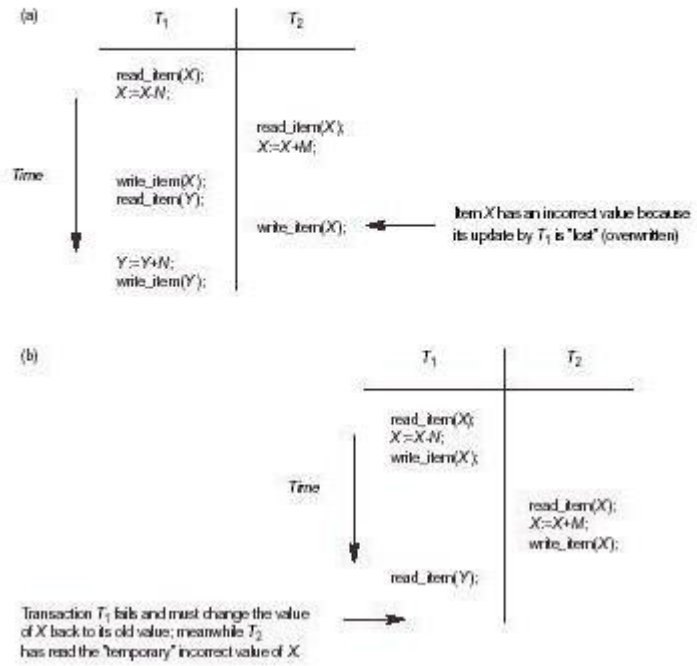


5.3 Why Concurrency Control Is Needed

- The Lost Update Problem.

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved then the final value of item *X* is incorrect, because T2 reads the value of *X before* T1 changes it in the database, and hence the updated value resulting from T1 is lost. For example, if *X* = 80 at the start (originally there were 80 reservations on the flight), *N* = 5 (T1 transfers 5 seat reservations from the flight corresponding to *X* to the flight corresponding to *Y*), and *M* = 4 (T2 reserves 4 seats on *X*), the final result should be *X* = 79; but in the interleaving of operations, it is *X* = 84 because the update in T1 that removed the five seats from *X* was *lost*

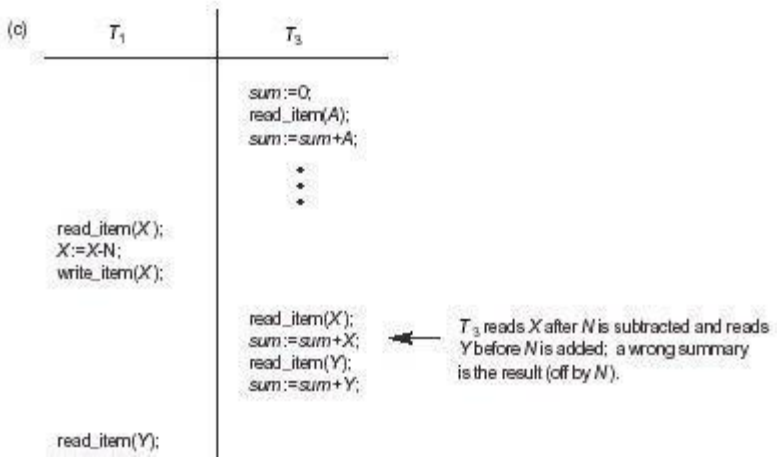
Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.



The Temporary Update (or Dirty Read) Problem.

This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value. Figure 19.03(b) shows an example where T1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction T2 reads the "temporary" value of *X*, which will not be recorded permanently in the database because of the failure of T1. The value of item *X* that is read by T2 is called *dirty data*, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem.*

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



- **The Incorrect Summary Problem.**

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure 19.03(c) occurs, the result of T3 will be off by an amount *N* because T3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it.

Another problem that may occur is called **unrepeatable read,** where a transaction *T* reads an item twice and the item is changed by another transaction *T'* between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer is inquiring 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.

5.4 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 has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not. This may happen if a transaction **fails** after executing some of its operations but before executing all of them.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

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. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
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. Notice that an exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be cancelled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
4. *Concurrency control enforcement:* The concurrency control method (see Chapter 20) may decide to abort the transaction, to be restarted later, because it violates serializability (see Section 19.5) or because 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. This may happen during a read or a write operation of the transaction.

6. *Physical problems and catastrophes:* This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

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 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.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

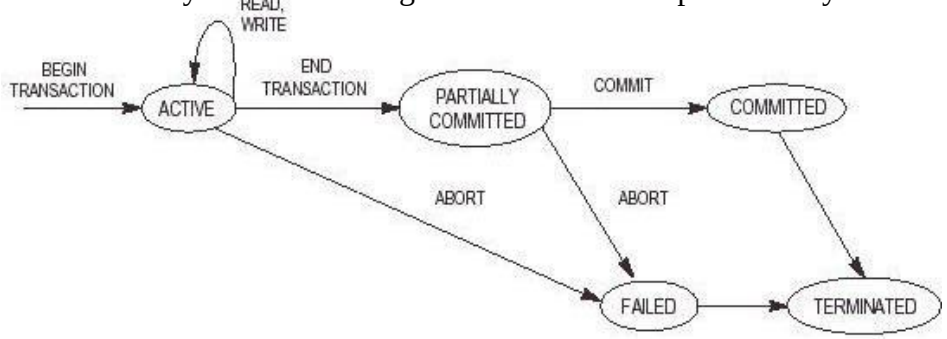
5.5 Transaction and System Concepts

Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see below). Hence, the recovery manager keeps track of the following operations:

- BEGIN_TRANSACTION: This marks the beginning of transaction execution.
- READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.
- END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 19.5) or for some other reason.
- COMMIT_TRANSACTION: This 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 (or ABORT): This 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.*

Shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log). Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state.** Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.



5.6 The System Log

- To be able to recover from failures that affect transactions, the system maintains a *log* to keep track of all transactions that affect the values of database items.
- Log records consists of the following information (*T* refers to a unique *transaction_id*):
    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.

5.7 Desirable Properties of Transactions

Transactions should posses the following (ACID) properties:

Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not 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. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

5.8 Schedules and Recoverability

A **schedule** (or **history**) *S* of *n* transactions T1, T2, ..., Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in *S*, the operations of Ti in *S* must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in *S*. For now, consider the order of operations in *S* to be a *total ordering*, although it is possible theoretically to deal with schedules whose operations form *partial orders*.

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Similarly, the schedule for Figure 19.03(b), which we call Sb, can be written as follows, if we assume that transaction T1 aborted after its read_item(*Y*) operation:

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

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*).

For example, in schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations w1(*X*) and w2(*X*). However, the operations r1(*X*) and r2(*X*) do not conflict, since they are both read operations; the operations w2(*X*) and w1(*Y*) do not conflict, because they operate on distinct data items *X* and *Y;* and the operations r1(*X*) and w1(*X*) do not conflict, because they belong to the same transaction.

A schedule *S* of *n* transactions T1, T2, ..., Tn, is said to be a **complete schedule** if the following conditions hold:

1. The operations in *S* are exactly those operations in T1, T2, ..., Tn, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

5.10 Characterizing Schedules Based on Recoverability

once a transaction *T* is committed, it should *never* be necessary to roll back *T*. The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called **nonrecoverable,** and hence should not be permitted.

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. A transaction *T* **reads** from transaction *T* in a schedule *S* if some item *X* is first written by T' and later read by *T*. In addition, T' should not have been aborted before *T* reads item *X*, and there should be no transactions that write *X* after T' writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *X).*

Consider the schedule S'_a given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

$$S'_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$$

S'_a is recoverable, even though it suffers from the lost update problem. However, consider the two (partial) schedules S_c and S_d that follow:

$$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$$
$$S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$$
$$S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$$

)

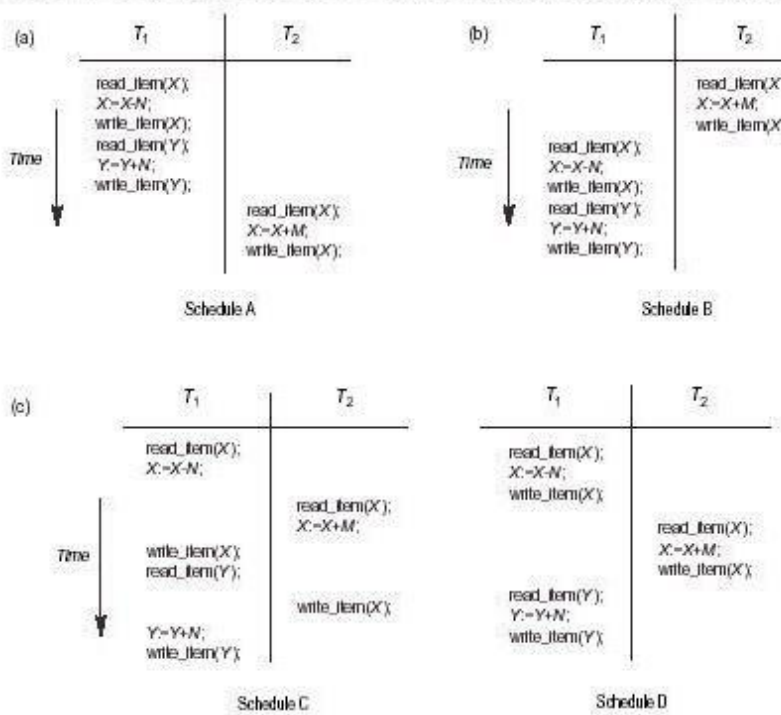
S_e is not recoverable, because T2 reads item *X* from T1, and then T2 commits before T1 commits. If T1 aborts after the c2 operation in S_e, then the value of *X* that T2 read is no longer valid and T2 must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c2 operation in S_e must be postponed until
after T1 commits. If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of *X* it read is no longer valid.

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

Serializability of Schedules

- If no interleaving of operations is permitted, there are only two possible arrangement for transactions T1 and T2.
    1. Execute all the operations of T1 (in sequence) followed by all the operations of T2 (in sequence).
    2. Execute all the operations of T2 (in sequence) followed by all the operations of T1
- A schedule *S* is *serial* if, for every transaction *T* all the operations of *T* are executed consecutively in the schedule.
- A schedule *S* of *n* transactions is *serializable* if it is equivalent to some serial schedule of the same *n* transactions.

Figure 19.5 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.



5.11 Transaction Support in SQL

- An SQL transaction is a logical unit of work (i.e., a single SQL statement).
- The *access mode* can be specified as *READ ONLY* or *READ WRITE*. The default is *READ WRITE*, which allows update, insert, delete, and create commands to be executed.
- The *diagnostic area size* option specifies an integer value *n*, indicating the number of conditions that can be held simultaneously in the diagnostic area.
- The *isolation level* option is specified using the statement *ISOLATION LEVEL*.
- the default isolation level is *SERIALIZABLE*.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
  READ WRITE
  DIAGNOSTICS SIZE 5
  ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
  VALUES ('Jabbar', 'Ahmad', '998877665', 2, 44000);
EXEC SQL UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
 GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: . . . ;
```

Concurrency Control in Databases

Two-Phase Locking Techniques for Concurrency Control:

The main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Types of Locks and System Lock Tables:

Locks are of two kinds −

- **Binary Locks** −
- **Shared/exclusive**

- **Binary Locks:** A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. 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).

lock_item(X):

```
B:
 if LOCK(X) = 0 (*item is unlocked*)
      then LOCK(X) ←1 (*lock the item*)
 else
      begin
      wait (until LOCK(X) = 0
           and the lock manager wakes up the transaction);
      go to B
      end;
```

unlock_item(X):

```
      LOCK(X) ← 0; (* unlock the item *)
      if any transactions are waiting then wakeup one of the waiting transactions;
```

- 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 LOCK(X) = 1, the transaction is forced to wait. If 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 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.

When we use the binary locking scheme, the system must enforce 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.

These rules can be enforced by the lock manager module of the DBMS.

- **Shared/Exclusive (or Read/Write) Locks:** This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

read_lock(X):

```
B:
if LOCK(X) = –unlocked‖
    then begin LOCK(X) ← –read-locked‖;

     no_of_reads(X) ← 1

    end
else if LOCK(X) = –read-locked‖

      then no_of_reads(X) ← no_of_reads(X) + 1 else
begin
      wait (until LOCK(X) = –unlocked‖
      and the lock manager wakes up the transaction);

      go to B

     end;
```

write_lock(X):

```
 B:
if LOCK(X) = –unlocked‖

       then LOCK(X) ← –write-locked‖
else begin
      wait (until 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 it 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;
```

Above algorithm shows Locking and unlocking operations for two mode (read/write, or shared/exclusive) locks.

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. This rule may be relaxed for downgrading of locks, as we discuss shortly.

5. A transaction T will not issue a write_lock (X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed for upgrading of locks, as we discuss shortly.

6. A transaction T will not issue an unlock (X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

- **Conversion (Upgrading, Downgrading) of Locks:** It is desirable to relax conditions 4 and 5 in the preceding list in order to allow lock conversion; that is, 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.

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: an **expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released; and **a 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 downrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking:

There are a number of variations of two-phase locking (2PL).
The technique just described is known as **basic 2PL**.

- A variation known as **conservative 2PL** (**or static 2PL)** requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set.

- In practice, the most popular variation of 2PL is strict 2PL, which guarantees strict schedules.

- In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts.

Hence, no other transaction can read or write an item that is written by Tunless T hascommitted, leading to a strict schedule for recoverability.

- Strict 2PL is not deadlock-free.

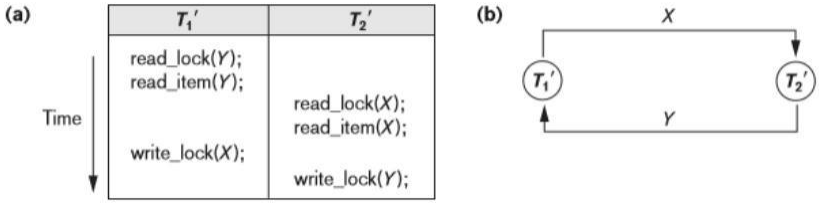
- A more restrictive variation of strict 2PL is rigorous 2PL, which also guarantees strict schedules.

- In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Dealing with Deadlock and Starvation:

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T′ 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 21.5(a), where the two transactions T1′ and T2′ are deadlocked in a partial schedule; T1′ is in the waiting queue for X, which is locked by T2′, whereas T2′ is in the waiting queue for Y, which is locked by T1′. Meanwhile, neither T1′ nor T2′ nor any other transaction can access items X and Y.



- Above Figure Illustrating the deadlock problem. (a) A partial schedule of T1′ and T2′ 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 (which is generally not a practical assumption)—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. Obviously, this solution further limits concurrency.

- 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, which is also not practical in the database context.

- Two schemes that prevent deadlock are called wait-die and wound-wait.

  - **Wait-die:** 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.

  - **Wound-wait:** 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.

- 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:** In the 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. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly.

    - **Cautious waiting algorithm:** The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts.

- **Deadlock Detection:**

    - An alternative 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 heavy, it may be advantageous to use a deadlock prevention scheme.

- **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.

- **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 in that it gives 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.
- The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

Concurrency Control Based on Timestamp Ordering

✓ **Timestamp**

- A timestamp is a unique identifier created by the DBMS to identify a transaction. Timestamp ordering do not use locks; hence, deadlocks cannot occur.

- Whenever a transaction begins, it receives a timestamp. This timestamp indicates the order in which the transaction must occur, relative to the other transactions.

- So, given two transactions that affect the same object, the operation of the transaction with the earlier timestamp must execute before the operation of the transaction with the later timestamp.

- However, if the operation of the wrong transaction is executed first, then it is aborted and the transaction must be restarted.

- Every object in the database has a **read timestamp**, which is updated whenever the object's data is read, and a **write timestamp**, which is updated whenever the object's data is changed.

Generation of time stamp

Timestamps can be generated in several ways.

1. 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.

2. Another way is 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.

2.2 Time Stamp Ordering Algorithm

- The idea for this scheme is to order the transactions based on their timestamps.

- A schedule in which the transactions participate is 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 allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must not violate the serializability order.

- To do this, the algorithm associates with each database item X to timestamp (TS)

  VALUES:

  1. Read –TS (X): The read timestamp of item X; this is the largest time stamp among all the timestamps of transactions that have successfully read item – X i.e. Read – TS (X) = TS (T). Where T is the youngest transaction that has read X successfully.

  2. Write – TS (X): The write timestamp of item X; this is the largest of all the timestamps of transactions that have successfully written item X – i.e., write –TS (X) = TS(T), where T is the youngest transaction that has written X successfully.

  Basic Timestamp Ordering (TO)

- Wherever some transaction T tries to issue a read- item (X) or write-item (X) operation, the basic TO (Time Ordering) compares the timestamp of T with read – TS

  (X) and write – TS (X) to ensure that the timestamp order of transaction is not violate.

- If this order is violated, then transaction is aborted resubmitted to the system as a new transaction with a new timestamp.

- If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back similarly, any transaction T2 that may have used a value written by T1 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 are not guaranteed to be recoverable.

- The concurrency control algorithm must check whether conflicting operations violate the time stamp ordering in the following two cases:

1. When a transaction T issues a write_item(X) operation, the following check is performed:

a. If read_TS(X) > TS(T) or if write_TS(X) > 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 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 write_TS(X) to TS(T).

2. When a transaction T issues a read_item(X) operation, the following check is performed:

a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than 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 write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Strict Timestamp Ordering (TO)

- A variation of basic TO is called Strict TO which 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 T1 that wrote the value of X (hence TS(T1) =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 T1 until T1 is either committed or aborted. This algorithm does not cause deadlock, since T waits T1 only if TS(T)>TS(T1).

Thomas Write Rule

A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability but it rejects some write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X)>TS(T) (read timestamp is greater than timestamp transaction), then abort and rollback T and reject the operation.

2. If Write_TS(X) > TS(T) (write timestamp is greater than timestamp transaction), then do not execute the write operation but continue processing. Because some transaction with a timestamp greater than TS(T) would have already written the value of X.

3. 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).

4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

In all concurrency control techniques we have discussed so far, a certain degree of checking is done before a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked.
In optimistic concurrency control techniques, also known as validation or certification techniques, no checking is done while the transaction is executing.

4.1 Validation-Based (Optimistic) Concurrency Control

In this schema, updates in the transaction are not applied directly to the database items on the disk until the transaction reaches its end and is validated. 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. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise the transaction is aborted and restarted later.

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.

5 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

5.1 Granularity Level Considerations for Locking

- The size of data items is often called the data item granularity. Fine granularity => Small item sizes
  Coarse granularity => Large item sizes

- 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).
  - 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).

- Smaller the data item size is, the more the number of items in the database.
  - More lock and unlock operations will be performed, causing a higher overhead.
  - Hence, more storage space will be required for the lock table.

- The best item size depends on the *types of transactions* involved.

5.2 Multiple Granularity Level Locking

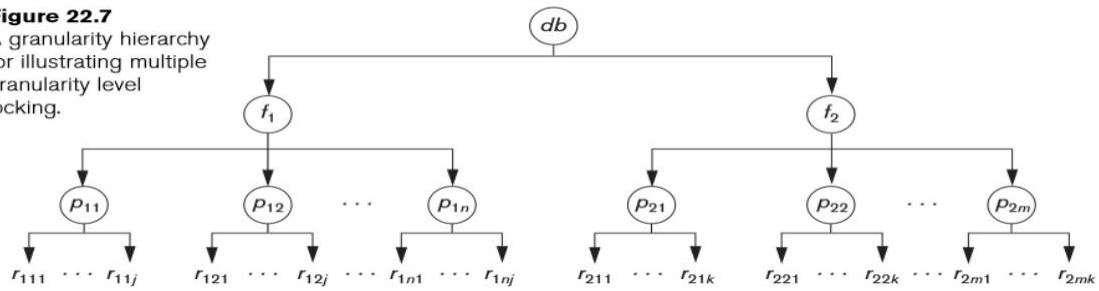


Figure 22.7
A granularity hierarchy for illustrating multiple granularity level locking.

The above figure shows a simple granularity hierarchy with a database containing two files (f1, f2), 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. However, additional types of locks will be needed to support such a protocol efficiently.

- Suppose transaction T1 wants to update all the records in file f1,and T1 requests and is granted an exclusive lock for f1.Then all of f1's pages (p11 through p1n)—and the records contained on those pages—are locked in exclusive mode.

- This is beneficial for T1 because setting a single file-level lock is more efficient than setting n page-level locks or having to lock each individual record.

- Now suppose another transaction T2 only wants to read record r1nj from page p1n of file f1; then T2 would request a shared record-level lock on r1nj.

- However, the database system (that is, the transaction manager or more specifically the lock manager) must verify the compatibility of the requested lock with already held locks.

- One way to verify this is to traverse the tree from the leaf r1nj to p1n to f1 to db. If at any time a conflicting lock is held on any of those items, then the lock request for r1nj is denied and T2 is blocked and must wait.

- This traversal would be fairly efficient.

- If transaction T2's request came before transaction T1's request, the shared record lock is granted to T2 for r1nj, but when T1's file-level lock is requested, it is quite difficult for the lock manager to check all nodes (pages and records) that are descendants of node f1 for a lock conflict.

- This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The 3 types are:

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 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 Nis already locked by transaction Tin 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.

	IS	IX	S	SIX	X
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.

Rule 1 simply states that conflicting locks cannot be granted.
Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes.

Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules.

To illustrate the MGL protocol with the database hierarchy in Figure 22.7, consider the following three transactions:

1. T1 wants to update record r111 and record r211.

2. T2 wants to update all records on page p12.

3. T3 wants to read record r11j and the entire f2 file.

Recovery Concepts

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 before image (BFIM) is overwritten with the after image(AFIM) in the database on disk.
- This process is generally known as **write-ahead logging** and is necessary so we can 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).
- With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk from its main memory buffer.

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 cache can be written to disk:

- 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 set to 1 (pin) to indicate that a cache buffer 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**.

- 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.

- If all pages updated by a transaction are immediately written to disk before the transaction commits, the recovery approach 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 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 having to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

22.1.4 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.

- The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time.

- 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.

- As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction.

- **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. However, until step 2 is completed, the previous checkpoint record should remain valid.

22.1.5 Transaction Rollback and Cascading Rollback

1. 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.

2. If any data item values have been changed by the transaction and written to the database on disk, they must be restored to their previous values (BFIMs).

3. The undo-type log entries are used to restore the old values of data items that must be rolled back.

4. 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.

5. 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. This phenomenon is called **cascading rollback**, and it can occur when the recovery protocol ensures **recoverable schedules** but does not ensure **cascadeless schedules.**

6. Cascading rollback can be complex and time-consuming. So almost all recovery mechanisms are designed then cascading rollback is never required.

7. The below figure shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown fig(a) and fig (b), the system log at the point of a system crash for a particular execution schedule of these transactions.

8. 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 T3 has not reached its conclusion and must be rolled back.

9. The WRITE operations of T3, marked by a single * in fig(b), are the T3 operations that are undone during transaction rollback. The fig(c) graphically shows the operations of the different transactions along the time axis.

(a) The read and write operations of three transactions

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

(b) System log at point crashes

	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]				

← System crash

* 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.

(c) Operations before the crash

We must now check for cascading rollback. From fig(c), we see that transaction T2 reads the value of item B that was written by transaction T3; this can also be determined by examining the log. Because T3 is rolled back, T2 must now be rolled back, too. The WRITE operations of T2, marked by ** in the log, are the ones that are undone. **Note:**

- Only write_item operations need to be undone during transaction rollback.
- read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.
- In practice, cascading rollback of transactions is never required because practical recovery methods guarantee cascadeless or strict schedules.
- Hence, there is also no need to record any read_item operations in the log because these are needed only for determining cascading rollback.

22.1.6 Transaction Actions That Do Not Affect the Database

1. 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.

2. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete.

3. 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 that is based on these reports and that affects the database.

4. Hence, such reports should be generated only after the transaction reaches its commit point.

5. 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 cancelled.

22.2 NO-UNDO/REDO Recovery Based on Deferred Update

- 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 force written 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.
- 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; hence all buffers that have been changed by the transaction must be pinned until the transaction commits

    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.

- Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations.

- REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk.

- For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated.

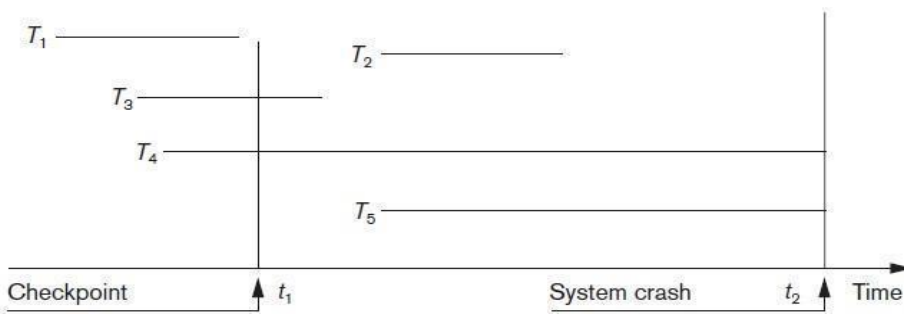
Recovery Algorithm RDU_M (Recovery using Deferred Update in a Multiuser environment) :

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 cancelled 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).

The below figure illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time *t*1, transaction *T*1 had committed, whereas transactions *T*3 and *T*4 had not. Before the system crash at time *t*2, *T*3 and *T*2 were committed but not *T*4 and *T*5. According to the RDU_M method, there is no need to redo the write_item operations of transaction *T*1—or any transactions committed before the last checkpoint time *t*1. The write_item operations of *T*2 and *T*3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions *T*4 and *T*5 are ignored: They are effectively cancelled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol (no-steal policy).



Drawbacks

1. The drawbacks of this method here is that it limits the concurrent execution of transactions because *all* write-locked items remain locked until the transaction reaches its commit point.

2. It require excessive buffer space to hold all updated items until the transactions commit.

Benefits:

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. 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.