ANNAI WOMEN'S COLLEGE

(Arts & Science)
(Affiliated to Bharathidasan University, Tiruchirappalli - 620 024)
TNPL Road, Punnam Chatram, Karur - 639 136.

DEPARTMENT OF COMPUTER SCIENCE, BCA & IT

Faculty Name : Mrs. M. Priya MCA., M.Phil.,

Major : B.Sc(CS)	**Semester** : IV
Subject Code : 16SCCCS4	**Subject** : Database Systems

DATABASE SYSTEMS

Unit I

    Introduction: Database-System Applications- Purpose of Database Systems - View of Data -- Database Languages - Relational Databases - Database Design -Data Storage and Querying Transaction Management -Data Mining and Analysis - Database Architecture - Database Users and Administrators - History of Database Systems.

Unit II

    Relational Model: Structure of Relational Databases -Database Schema - Keys - Schema Diagrams - Relational Query Languages - Relational Operations Fundamental Relational- Algebra Operations Additional Relational-Algebra Operations- Extended Relational-Algebra Operations - Null Values - Modification of the Database.

Unit III

    SQL: Overview of the SQL Query - Language - SQL Data Definition - Basic Structure of SQL Queries - Additional Basic Operations - Set Operations - Null Values Aggregate Functions - Nested Subqueries - Modification of the Database -Join Expressions - Views - Transactions - Integrity Constraints - SQL Data Types and Schemas - Authorization .

Unit IV

    Relational Languages: The Tuple Relational Calculus - The Domain Relational Calculus Database Design and the E-R Model: Overview of the Design Process - The Entity-Relationship Model - Reduction to Relational Schemas - Entity-Relationship Design Issues - Extended E-R Features - Alternative Notations for Modeling Data - Other Aspects of Database Design .

Unit V

    Relational Database Design: Features of Good Relational Designs - Atomic Domains and First Normal Form - Decomposition Using Functional Dependencies - Functional-Dependency Theory - Decomposition Using Functional Dependencies - Decomposition Using Multivalued Dependencies-More Normal Forms - Database-Design Process

Text Book:

    1. Database System Concepts, Sixth edition, Abraham Silberschatz, Henry F. Korth, S.Sudarshan, McGraw-Hill-2010.

INDEX

UNIT-I
INTRODUCTION

1. What is DBMS? (Part-A)

   What is Database? (Part-A)

❖ A **Database Management System (DBMS)** consists of a collection of interrelated data and a set of programs to access those data. The collection of data is called as database, contains information about one particular enterprise.

❖ The primary goal of a DBMS is to provide an environment that is both convenient and efficient for people to use in retrieving and storing database information.

2. Why Database Systems are designed? (Part-A)

   What is the Use of Database Systems? (Part-A)

✓ The Database Systems are designed to store large amount of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information.

✓ It must provide for the safety of information stored, despite system crashes or attempts at unauthorized access. The data are shared among several users the system must avoid possible anomalous results.

✓

DATABASE SYSTEM APPLICATIONS

1. List out the Database System Applications. (Part-B)

✓ The widely used database applications are:

   Banking:
   - For customer information, accounts, loans and banking transactions.

   Airlines:
   - For reservations and schedule information. Airlines were the first to use databases in a geographically distributed manner.

   Universities:
   - For student information, course registrations, and grades.

   Credit and Transactions:
   - For purchase on credit cards and generation of monthly statements.

   Telecommunication:
   - For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

   Sales:
   - For customer, product, and purchase information.

   Manufacturing:
   - For management of supply chain and for tracking production of items in factories, inventories of items in warehouses, and orders for items.

   Finance:
   - For storing information about holdings, Sales, and purchases of financial instruments such as stocks and bonds.

   Human Resources:
   - For information about employees, salaries, payroll taxes and benefits and for generation of paychecks.

PURPOSE OF DATABASE SYSTEMS

1. Explain about Database Systems. (Part-C)

   Discuss about File-Processing Systems. (Part-C)

   List out and explain about the Disadvantages of File-Processing System. (Part-C)

- To keep the information on a computer is to store it in permanent system files. The user to manipulate the information, the system has a number of application programs that manipulate the files, including

  - ✓ A program to debit or credit an account
  - ✓ A program to add a new account
  - ✓ A program to find the balance of an account
  - ✓ A program to generate monthly statements

- The new application programs are added to the system as the need arises. The file-processing system is supported by a conventional operating system.
- To keep the organizational information in a file processing system has a number of major disadvantages. They are:
  - **Data Redundancy and Inconsistency**
  - **Difficulty in Accessing Data**
  - **Data Isolation**
  - **Integrity Problems**
  - **Atomicity Problems**
  - **Concurrent-Access Anomalies**
  - **Security Problems**

Data Redundancy and Inconsistency:

- The same information duplicated in several places (files) is called as **Redundancy**. This redundancy leads to higher storage and access cost.
- The redundancy may lead to data inconsistency; that is, the various copies of the same data may no longer agree.

Example:
- A changed student address may be reflected in one record but not elsewhere in the system.

Difficulty in Accessing Data:

- The file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner.

Example:
- The bank officer needs to find out the names of all customers who live within the particular city. The officers extract the information manually, or ask data-processing department to a have system programmer write the necessary application program.
- Both alternatives are unsatisfactory because if the officer needs some other information it is very difficult to extract the information with the existing application program.

Data Isolation:

- The data are scattered in various files, and files may be in different formats, it is difficult to write new application programs to retrieve the appropriate data.

Integrity Problems:

- The data values stored in the database must satisfy certain types of consistency constraints.

Example:
- The balance of bank account may never fall below a prescribed amount ($250). The developers enforce these constraints in the system by adding appropriate code in the various application programs. When new constraints are added, it is difficult to change the programs to enforce them.

Atomicity Problems:

- In computer system any mechanical or electrical device, is subject to failure. Once the failure has occurred and has been detected, the data are restored to the consistent state that existed prior to the failure.
- The transaction must be atomic- it must happen in it's entirely or not at all. It is difficult to ensure this property in a file-processing system.

Concurrent-Access Anomalies:
- The overall performance of the system is improved and a faster response time is possible, many systems allow multiple users to update the data simultaneously.
- The interaction of concurrent updates may result in inconsistent data.

Security Problems:
- Not every user of the database system should be able to access all the data. The application programs are added to the system in an ad hoc manner, it is difficult to enforce such security constraints.

VIEW OF DATA

1. What is the Major Purpose of Database System? (Part-A)
   What is Data Abstraction? (Part-A)
   Explain about the Various Levels of Data Abstraction. (Part-B)
 a) Data Abstraction:
- A major purpose of a database system is to provide users with an abstract view of the data. The system hides certain detail of how data are stored and maintained is called as data abstraction.
- To hide the information through several levels of abstraction, to simply users interactions with the system:
  - **Physical Level**
  - **Logical Level**
  - **View Level**

Physical Level:
- The lowest level of abstraction describes how the data are actually stored. The complex low-level data structures are records are described as a block of consecutive storage locations (words, bytes).
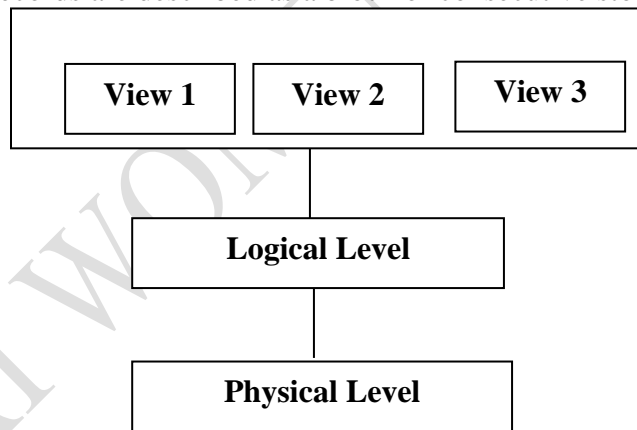

Figure: The three levels of Data Abstraction

Logical Level:
- This level describes what data are stored in the database, and what relationships exist among those data. It is used by database administrators, who must decide what information is to kept in the database.

View Level:
- The highest level of abstraction describes only part of the entire database. It is defined for simplify the interaction with the system. The system may provide many views for the same database.
- To hide the details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing parts of the database.

b) Instances and Schema:
2. What is the Instance of the Database? (Part-A)
   What is Schema? (Part-A)
3. What is Data Independence? (Part-A)
   Explain about the Types of Data Independence. (Part-B)

❖ The collection of information stored in the database at a particular moment is called **an instance** of the database.
❖ The overall design of the database is called the **database schema**. A database schema corresponds to the programming language type-definition.
❖ A value of a variable in programming languages corresponds to an instance of a database schema.
❖ The database systems have several schemas, partitioned according to the levels of abstraction. At the lowest level is the **physical schema**; at the intermediate level is the **logical schema**; and at the highest level is a **subschema**.
❖ The database systems support one physical schema, one logical schema, and several subschema's.
❖ The ability to modify a schema definition in one level without affecting a schema definition in the next higher level is called data independence. There are two levels of data independence:
   ▪ **Physical Data Independence**
   ▪ **Logical Data Independence**

Physical Data Independence:
❖ It is the ability to modify the physical schema without causing application programs to be rewritten.
❖ The modifications at the physical level are necessary to improve performance.

Logical Data Independence:
❖ It is the ability to modify the logical schema without causing application programs to be rewritten.
❖ The modifications at the logical level are necessary whenever the logical structure of the database is altered. It is more difficult to achieve than in physical data independence, since application programs are heavily dependent on the logical structure of the data.

c) Data Models

1. What is Data Model? (Part-A)
➤ The structure of a database is called data model. It consists of collection tool for describing data, data relationships, data semantics, and consistency constraints.
➤ The data models can be classified into four different categories:
   ▪ **Relational model**
   ▪ **Entity-Relationship Model**
   ▪ **Object-Based Data Model**
   ▪ **Semantic Data Model**

Relational Model:
➤ It consists of collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.
➤ Tables are also known as **relations**.
➤ The relational model is an example of a **record-based model**. Record-based models are so named because the database is structured in fixed-format records of several types.
➤ Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

Entity-Relationship Model:
➤ It consists of collection of basic objects, called entities, and relationships among those object is called Entity-Relationship model.
➤ An entity is a "thing" or "object" in the real world that is distinguishable from other objects.
➤ The overall logical structure of a database can be expressed graphically by an E-R diagram.

E-R Diagram

6
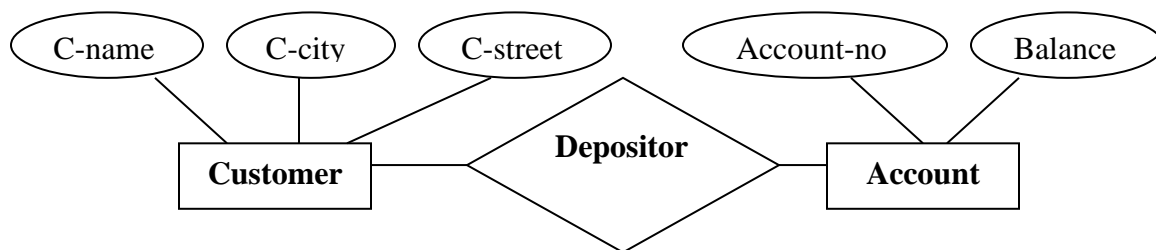
Components of E-R Diagram:
        Rectangles represent entity sets, **Ellipses** represent attributes, **Diamonds** represent relationships among entity sets, **Lines** link attributes to entity sets and entity sets to relationships.

Object-Based Data Model

➢ Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology.

➢ This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity.

➢ The object-relational data model combines features of the object-oriented data model and relational data model.

Semistructured Data Model:

➢ The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes.

➢ This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes.

➢ The **Extensible Markup Language (XML)** is widely used to represent semistructured data.

DATABASE LANGUAGES

1. Explain about the Database Languages. (Part-B)
   Expand DDL and DML. (Part-A)
   What is Query? (Part-A)
   What is Data Dictionary? (Part-A)

❖ A database system provides two different types of languages. They are:
- **Data-Definition Language (DDL)**
- **Data-Manipulation Language (DML)**

Data-Definition Language:

❖ A database schema is specified by a set of definitions expressed by a special language called a DDL.

❖ The result of compilation of DDL statements is a set of tables that is stored in a special file called **data dictionary or data directory.**

❖ A data dictionary is a file that contains **metadata** (data about data). The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a data storage and definition language.

❖ Thus, database systems implement integrity constraints that can be tested with minimal overhead:

• **Domain Constraints:**
   ✓ A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types).
   ✓ Domain constraints are the most elementary form of integrity constraint.

• **Referential Integrity**:
   ✓ There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity).

• **Assertions:**
   ✓ An assertion is any condition that the database must always satisfy.
   ✓ Domain constraints and referential-integrity constraints are special forms of assertions.

• **Authorization**.
   ✓ We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being:

- o **read authorization**, which allows reading, but not modification, of data;
  - o **insert authorization**, which allows insertion of new data, but not modify existing data;
  - o **update authorization**, which allows modification, but not deletion, of data; and
  - o **delete authorization**, which allows deletion of data.
- ❖ The result of compilation of DDL statements is a set of tables that is stored in a special file called **data dictionary or data directory.**
- ❖ A data dictionary is a file that contains **metadata** (data about data). The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a data storage and definition language.

Data-Manipulation Language:

- ❖ It is a language that enables user to access or manipulate data as organized by the appropriate data model. Using this language perform the following operations:
  - ▪ **Retrieval of information from database**
  - ▪ **Insertion of information in to database**
  - ▪ **Deletion of information from database**
  - ▪ **Modification of information stored in the database**
- ❖ A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language.
- ❖ The **DML** consists of the following types:
  - ▪ **Procedural DML**
  - ▪ **Nonprocedural DML**

Procedural DML:

- ❖ It requires a user to specify what data are needed and how to get those data.

Nonprocedural DML:

- ❖ It requires a user to specify what data are needed without specifying how to get those data. It is usually easier to learn and use than are procedural DMLs.

RELATIONAL DATABASES

1. Define Data-Manipulation Language. (Part-A)
Define DDL (Part-A)

- ➢ A primary goal of a database system is to retrieve information from and store new information in the database. It also includes a DML and DDL.

a) Tables:

- ➢ Each table has multiple columns and each column has a unique name. **Figure 1.0** presents a sample relational database comprising two tables: one shows details of university instructors and the other shows details of the various university departments.
- ➢ The **first table,** the **instructor** table, shows, for example, that an instructor named Einstein with **ID** 22222 is a member of the Physics department and has an annual salary of $95,000.
- ➢ The **second table**, **department** table**,** shows, for example, that the Biology department is located in the Watson building and has a budget of $90,000. Of course, a real-world university would have many more departments and instructors.
- ➢ The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types.
- ➢ Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El said	History	60000
45565	Katz	Comp. Sci	75000
98345	Kim	Elec. Eng	8000
76766	Cruck	Biology	72000
10101	Srinivasan	Comp.Sci	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	finance	80000

(a) The instructor table

dept_name	building	budget
Comp. Sci	Taylor	100000
Biology	Watson	90000
Elec. Eng	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The department table
Figure-1.0 A sample relational database

b) **Data-Manipulation Language**
  ➢ The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table.
  ➢ **Example :**   SQL query that finds the names of all instructors in the History department:

> **select instructor.name**
> **from instructor**
> **where instructor.dept name = 'History';**

  ➢ The query specifies that those rows from the table **instructor** where the **dept _name** is History must be retrieved, and the name attribute of these rows must be displayed.

c) **Data-Definition Language**
  ➢ SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc.
  ➢ For instance, the following SQL DDL statement defines the department table:

> **create table** department
> (dept name **char** (20),
> building **char** (15),
> budget **numeric** (12,2));

  ➢ Execution of the above DDL statement creates the department table with three columns: dept name, building, and budget, each of which has a specific data type associated with it.

d) **Database Access from Application Programs**
  ➢ SQL is not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL.
  ➢ Such computations and actions must be written in a *host* language, such as C, C++, or Java with embedded SQL queries that access the data in the database.
  ➢ **Application programs** are programs that are used to interact with the database in this fashion.

> ➢ To access the database, DML statements need to be executed from the host language.
> ➢ There are two ways to do this:

• By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results. The Open Database Connectivity (ODBC) standard for use with the language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

• By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the **DML precompiler**, converts the DML statements to normal procedure calls in the host language.

DATABASE DESIGN

> ➢ Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation.
> ➢ They are part of the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supporting role.

a) Design Process

> ➢ A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements.
> ➢ The initial phase of database design, then, is to characterize fully the data needs of the prospective database users.
> ➢ The designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise.
> ➢ A fully developed conceptual schema indicates the functional requirement of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data.
> ➢ The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.
>   - Logical-design phase
>   - Physical- design phase

Logical-Design Phase:
> ❖ The designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used.

Physical Design Phase:
> ❖ The designer uses the resulting system-specific database schema in the subsequent physical-design phase, in which the physical features of the database are specified.

b) Database Design for a University Organization
> ❖ The design process, let us examine how a database for a university could be designed.
> ❖ The initial specification of user requirements may be based on interviews with the database users, and on the designer's own analysis of the organization.
> ❖ Here are the major characteristics of the university.

• The university is organized into departments. Each department is identified by a unique name (**dept_ name**), is located in a particular **building**, and has a **budget.**

• Each department has a list of courses it offers. Each course has associated with it a **course_id, title, dept_ name,** and **credits**, and may also have associated **prerequisites.**

• Instructors are identified by their unique ID. Each instructor has **name**, associated department (**dept_ name**), and **salary**.

• Students are identified by their **unique ID**. Each student has a **name**, an associated major department (**dept name**), and **tot_cred** (total credit hours the student earned thus far).

• The university maintains a list of classrooms, specifying the name of the **building, room number**, and **room capacity.**

• The university maintains a list of all classes (sections) taught. Each section is identified by a **course_id, sec_id, year,** and **semester,** and has associated with it a **semester, year, building, room_number,** and **time_slot_id** (the time slot when the class meets).

• The department has a list of teaching assignments specifying, for each instructor, the sections the instructor is teaching.

• The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).

c) **The Entity-Relationship Model**

❖ The **entity-relationship (E-R)** data model uses a collection of basic objects, called entities, and relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects.

❖ Entities are described in a database by a set of **attributes**.

Example:

• The attributes **dept_name, building**, and **budget** may describe one particular department in a university, and they form attributes of the **department** entity set. Similarly, attributes ID, name, and salary may describe an instructor entity.

❖ A **relationship** is an association among several entities.

Example:

• A member relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

❖ The overall logical structure (schema) of a database can be expressed graphically by an entity-relationship (E-R) diagram. There are several ways in which to draw these diagrams. One of the most popular is to use the **Unified Modeling Language (UML)**.

❖ In the notation we use, which is based on UML, an E-R diagram is represented as follows:

Instructor		department
ID name salary	member	dept_name building budget
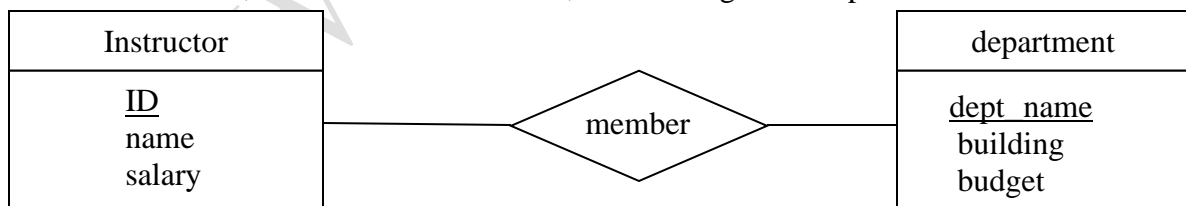
A sample E-R diagram.

• Entity sets are represented by a rectangular box with the entity set name in the header and the attributes listed below it.

• Relationship sets are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

❖ The E-R diagram indicates that there are two entity sets, **instructor and department**, with attributes as outlined earlier.

❖ The diagram also shows a relationship **member** between **instructor and department**.

❖ In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set.

d) Normalization

- ❖ Another method for designing a relational database is to use a process commonly known as normalization.
- ❖ The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate **normal form.**
- ❖ To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:
    - • Repetition of information
    - • Inability to represent certain information

ID	name	salary	dept_name	building	budget
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci	Taylor	100000
98345	Kim	8000	Elec. Eng	Taylor	85000
76766	Cruck	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp.Sci	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	finance	Painter	120000

The Faculty table

- ❖ The two separate tables **instructor** and **department**, we have a single table, **faculty**, that combines the information from the two tables.
- ❖ Notice that there are two rows in faculty that contain repeated information about the History department, specifically, that department's building and budget.
- ❖ The repetition of information in our alternative design is undesirable.
- ❖ Repeating information wastes space.
- ❖ One solution to this problem is to introduce **null** values.
- ❖ The null value indicates that the value does not exist (or is not known).
- ❖ An unknown value may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists).

DATABASE SYSTEM STRUCTURE

1. Explain about the Database System Structure. (Part-C)

Explain about the Components of Database System. (Part-C)

- ❖ A database system is partitioned into modules that deal the responsibilities of the overall system. The functions of database system may be provided by the computer's operating system. The design of a database system must include consideration of the interface between the database system and the operating system.
- ❖ The functional components of a database system can be broadly divided into the following components:
    - ▪ **Storage Manager**
    - ▪ **Query Processor**

a) Storage Manager:

- ❖ It is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

❖ It is also responsible for the interaction with the file manager. It also translates the various DML statements into low-level file-system commands.
❖ It also responsible for the interaction, storing, retrieving, and updating of data in the database. The following are components in this manager:

- **Authorization and Integrity Manager**
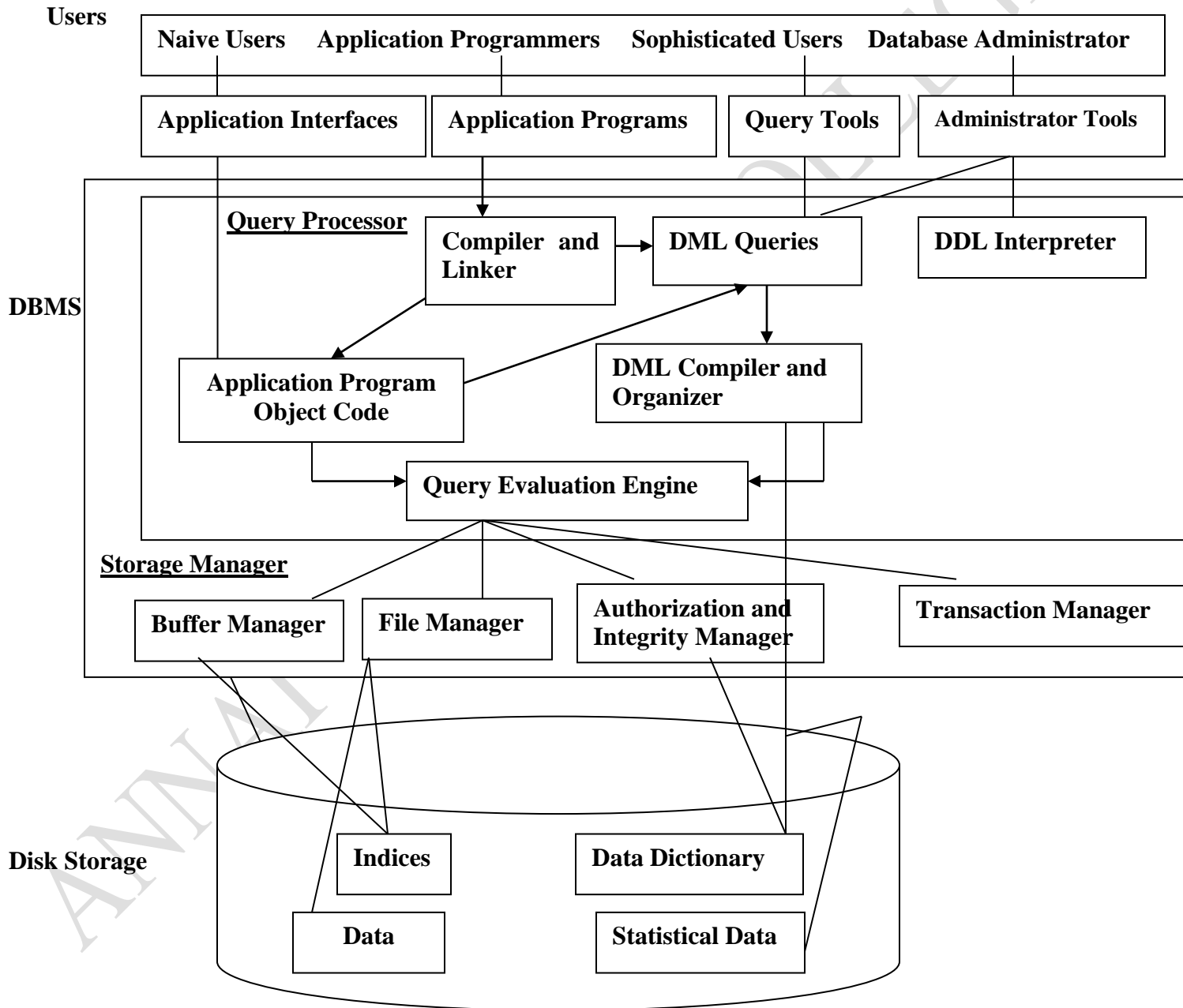- **Transaction Manager**
- **File Manager**
- **Buffer Manager**

Users

| Naive Users | Application Programmers | Sophisticated Users | Database Administrator |

| Application Interfaces | Application Programs | Query Tools | Administrator Tools |

DBMS

Query Processor

| Compiler and Linker | → | DML Queries | | DDL Interpreter |

| Application Program Object Code | | DML Compiler and Organizer |

| Query Evaluation Engine |

Storage Manager

| Buffer Manager | File Manager | Authorization and Integrity Manager | | Transaction Manager |

Disk Storage

| Indices | | Data Dictionary |

| Data | | Statistical Data |

Figure: System Structure

- **Authorization and Integrity Manager:**
  ✓ It tests the satisfaction of integrity constraints and checks the authority of users to access data.

13

- ▪ **Transaction Manager**:
  - ✓ The manager ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- ▪ **File Manager**:
  - ✓ This type of manager manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

- ▪ **Buffer Manager**:
  - ✓ This type of manager is responsible for fetching data from disk storage into main memory, and deciding what data to cache in memory.

- ❖ The storage manager implements several data structures as part of the physical system implementation:
  - **Data Files:**
    - ✓ It stores the database itself.
  - **Data Dictionary**:
    - ✓ It stores metadata about the structure of the database.
  - **Indices:**
    - ✓ It provides fast access to data items that hold particular values.

b) The Query Processor:

- ❖ The query processor consists of the following components:
  - ▪ **DML Compiler**
  - ▪ **DDL Interpreter**
  - ▪ **Query Evaluation Engine**

  DML Compiler:
  - ✓ It translates DML statements in a query language into the low-level instructions that the query evaluation engine understands. It also translates a user's request into an equivalent more efficient form.

  DDL Interpreter:
  - ✓ It interprets DDL statements and records them in a set of tables containing metadata.

  Query Evaluation Engine:
  - ✓ It executes low-level instructions generated by the DML compiler.

TRANSACTION MANAGEMENT

1. Explain about the Transaction Management. (Part-B)
What is Transaction? (Part-A)
Explain about the ACID Properties. (Part-B)
What is the Role of Transaction Manager? (Part-A)

- ❖ A transaction is a collection of operations that performs a single logical function in a database application. The transaction manager is responsible for ensuring that the database remains in a consistent (correct) state despite system failures.
- ❖ Each transaction is a unit of both atomicity and consistency. Each and every transaction does not violate any database consistency constraints.

ACID Properties:

- ❖ To ensure the integrity of the data the database system maintains the ACID properties of the transaction.

Atomicity:

- ❖ Either all operations of the transaction are reflected properly in the database, or none are.

Consistency:

- ❖ Execution of a transaction in isolation (that is, no other transaction executing concurrently) preserves the consistency of the database.

14

Isolation:
- ❖ Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, the T_j finished execution before T_i started or T_j started execution after T_j finished.

Durability:
- ❖ After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

DATA MINNING & ANALYSIS

1. Define data minning. (Part-A)
- ❖ The term **data mining** refers loosely to the process of semiautomatically analyzing large databases to find useful patterns.
- ❖ Like knowledge discovery in artificial intelligence (also called **machine learning**) or statistical analysis, data mining attempts to discover rules and patterns from data.
- ❖ However, data mining differs from machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases."
- ❖ Some types of knowledge discovered from a database can be represented by a set of **rules**.
  Example : "Young women with annual incomes greater than $50,000 are the most likely people to buy small sports cars." Of course such rules are not universally true, but rather have degrees of "support" and "confidence."
- ❖ There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns.
- ❖ **Data warehouses:**
  - ▪ To execute queries efficiently on such diverse data, companies have built *data warehouses*.
  - ▪ Data warehouses gather data from multiple sources under a unified schema, at a single site.
- ❖ **Textual data**:
  - ▪ It has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases.
- ❖ **Information Retrival:**
  - ▪ Querying of unstructured textual data is referred to as *information retrieval*.
  - ▪ Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage.

DATABASE ARCHITECTURE

- ❖ The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs.
- ❖ Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines.
- ❖ Database systems can also be designed to exploit parallel computer architectures.
- ❖ Most users of a database system today are not present at the site of the database system, but connect to it through a network.
- ❖ We can therefore differentiate between **client** machines, on which remote database users work, and **server** machines, on which the database system runs.
- ❖ Database applications are usually partitioned into **two or three parts:**
  - • **Two-tier architecture**
  - • **Three- tier architecture**

Two-tier architecture:
     ✓ The application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.
     ✓ **Example:** Client programs using ODBC/JDBC to communicate with a database

Three-tier architecture:
     ✓ The client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an **application server**, usually through a forms interface.
     ✓ **Example:** Web-based applications, and applications built using "middleware"

❖ The application server in turn communicates with a database system to access data.
❖ The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.
❖ Three-tier applications are more appropriate for large applications, and for applications that run on the WorldWideWeb.
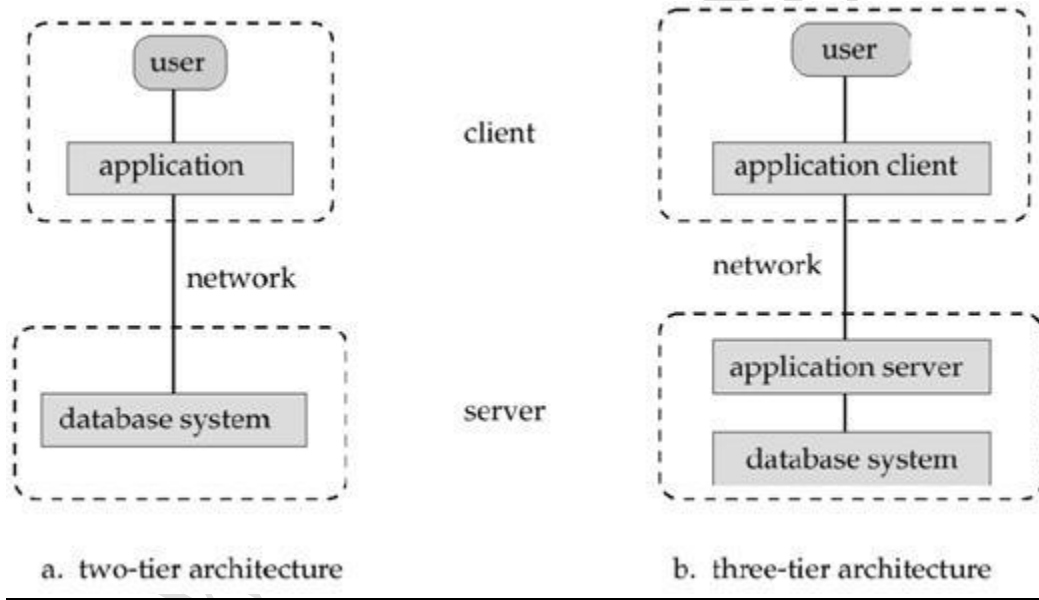

a. two-tier architecture   b. three-tier architecture

Fig. Two-Tier and Three-Tier architectures.

DATABASE USERS AND ADMINISTRATORS

1. What is the Primary Goal of Database System? (Part-A)
  List out the Different Types of Database Users? (Part-A)
  Explain about the Types of Database Users. (Part-B)
  Explain about the Database Administrator. (Part-B)
  What is DBA? (Part-A)
  Discuss abut the Functions of DBA. (Part-A)
    ❖ A primary goal of a database system is to retrieve information from and store new information in the database.

a) Database Users and User Interface:
    ❖ There are four different types of database-system users, differentiated by the way that they expect to interact with the system.
        ▪ **Application Programmers**
        ▪ **Sophisticated Users**
        ▪ **Specialized Users**
        ▪ **Naive Users**

16

- **Application Programmers**:
  - ❖ The application programmers are computer professionals who interact with the system through DML, which are embedded in a program written in a host language. These programs are commonly referred to as application programs.
  - ❖ The DML precompiler is a special preprocessor converts the DML statements to normal procedure calls in the host language. The resulting program is then run through the host-language compiler, which generates appropriate object code.

- **Sophisticated Users:**
  - ❖ The sophisticated users interact with the system without writing programs. They form their requests by database query language. Each such query is submitted to a query processor whose function is to break down DML statement into instructions that the storage manager understands.

- **Specialized Users:**
  - ❖ These users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.
  - ❖ These applications are computer-aided design systems, knowledge-base and expert systems, systems that store data with complex data types and environment-modeling systems.

- **Naive Users:**
  - ❖ These users are unsophisticated users who interact with the system by invoking one of the permanent application programs that have been written previously.

b) Database Administrator:
  - ❖ The person who has such central control over the system is called the Database Administrator (DBA). The functions of the DBA include the following:
    - **Schema Definition**
    - **Storage Structure and Access-Method Definition**
    - **Schema and Physical-Organization Modification**
    - **Granting of Authorization for Data Access**
    - **Integrity-Constraint Specification**

- **Schema Definition:**
  - ❖ The DBA creates the original database schema by writing a set of definitions that is translated by the DDL compiler to a set of tables that is stored permanently in the data dictionary.

- **Storage Structure and Access-Method Definition:**
  - ❖ The DBA creates appropriate storage structures and access methods by writing a set of definitions, which is translated by the data-storage and data-definition-language compiler.

- **Schema and Physical-Organization Modification:**
  - ❖ Programmers accomplish the modifications either to the database schema or to the description of the physical storage organization by writing a set of definitions that is used by either the DDL compiler or the data-storage and data-definition-language compiler to generate modifications to the appropriate internal system tables.

- **Granting of Authorization for Data Access:**
  - ❖ The granting of different types of authorization allows the database administrator to regulate which parts of the database various users can access.

- **Integrity-Constraint Specification:**
  - ❖ The data values stored in the database must satisfy certain consistency constraints. The constraints specified explicitly by the database administrator.

UNIT-II
RELATIONAL MODEL
STRUCTURE OF RELATIONAL DATABASES

1. Explain about the Structure of Relational Databases. (Part-C)
   What is Query Language? (Part-A)
   What is Domain? (Part-A)

❖ A relational database consists of collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values.

a) Basic Structure:

❖ Consider the account table, contains three column headers: branch-name, account-number, and balance. For each attribute, there is a set of permitted values, called the domain of that attribute. The attribute branch-name, the domain is the set of all branch names.

❖ Let D_1 denote the branch names, D_2 denote the set of all account numbers and D_3 is the set of all balances. Any row of account must consist of a 3-tuple (V_1, V_2, V_3), where V_1 is a branch-name, V_2 is an account-number, and V_3 is a balance. The account will contain only a subset of the set of all possible rows. The account is a subset of

      ▪ $D_1 \times D_2 \times D_3$

❖ In general, a table of n attributes must be a subset of

      ▪ $D_1 \times D_2 \times \ldots\ldots \times D_{n-1} \times D_3$

Example:

❖ The account relation consists of the following tuples. The tuple variable t refers to the first tuple of the relation.

Branch-Name	Account-Number	Balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700

❖ The notation **t[branch-name]** to denote the value of t on the branch-name attribute.

      ▪ **t[branch-name]="Downtown"**
      ▪ **t[balance]=500**

❖ The t[1] to denote the value of tuple t on the first attribute (branch-name), t[2] to denote the value of tuple t on the second attribute (account-number) and so on.

❖ A relation is a set of tuples, the notation **t∈r** to denote that tuple t is in relation r. The domains of all attributes of r must be atomic. The domain also contains the null value, which signifies that the value is unknown or does not exist.

b) Database Schema:

❖ The logical design of the database is called as **database schema**.

❖ The **database instance,** which is a snapshot of the data in the database at a given instant in the time.

❖ The concept of a relation corresponds to the programming-language notion of a **variable**. The concept of a relation schema corresponds to the programming language notion of **type definition**.

❖ The user provides the name for relation and relation schema. The lowercase names for relations, and names beginning with an uppercase letter for relation schemas. The user use Account-schema to denote the relation schema for relation account. Thus,

      ▪ **Account-Schema=(branch-name, account-number, balance)**

❖ The account is a relation on Account-Schema by

      ▪ **Account(Account-Schema)**

the relation schema comprises a list of attributes and their corresponding domains.

Keys:

❖ For the clarity in DBMS, the keys are preferred and they are important part of the arrangement of a table. The keys make sure to uniquely identify a table's each part or record of a field or combination of fields. A database is made up of tables, which (tables) are made up of records, which (records) further made up of fields. Let us take an example to illustrate what are keys in database management system. This article is about different keys in database management system (DBMS).

Example:

Serial No.	Item Name	Quantity	Price
1.	Cashew nuts	1 Kg	800
2.	Almonds	1Kg	900
3.	Pine nuts	1 Kg	1000

Field   Record

Keys In Database Management

• In the above data item, each column is a field and each row is a record.

Types of Keys in Database Management System: Each key which has the parameter of uniqueness is as follows:

1. Super key
2. Candidate key
3. Primary key
4. Composite key
5. Secondary or Alternative key
6. Non- key attribute
7. Non- prime attribute
8. Foreign key
9. Simple key
10. Compound key
11. Artificial key

1.**Super key:**

✓ Super Key is a set of properties within a table; it specially identifies each record in a table. Candidate key is a unique case of super key.

• **Example:** Roll No. of a student is unique in relation. The set of properties like roll no., name, class, age, sex, is a super key for the relation student.
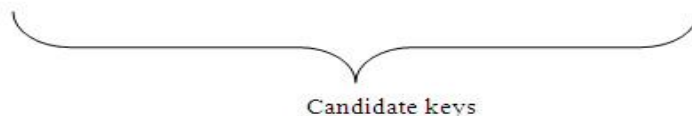
2. **Candidate key:**

✓ It is a set of fields; primary key can be selected from these fields.
✓ A set of properties or attributes acts as a primary key for a table.

Every table must have at least one candidate key or several candidate keys. It is a super key's subset.

• **Example:**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Candidate keys

  ✓ The above fields of a candidate key uniquely identify a student.
  ✓ It has the properties like – Being unique and Parameter of irreducibility.
3. **Primary key:**
  ✓ The candidate key which is very suitable to be the main key of table is a primary key.
  ✓ The primary keys are compulsory in every table.
  ✓ The properties of a primary key are:
    o Model stability
    o Occurrence of minimum fields
    o Defining value for every record i.e. being definitive
    o Feature of accessibility
  • **Example**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Primary key

4. **Composite key:**
  ✓ It has two or more properties which specially identifies the occurrence of an entity.
  • **Example:**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Candidate keys

  ✓ In the above example the customer identity and order identity has to combine to uniquely identify the customer details.
5. **Secondary or Alternative key:**
  ✓ The rejected candidate keys as primary keys are called as secondary or alternative keys.
  • **Example:**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Secondary or Alternative keys

6. **Non- key Attribute:**
✓ The attributes excluding the candidate keys are called as non-key attributes.
• **Example:**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Non-key attribute

7. **Non-prime Attribute:**
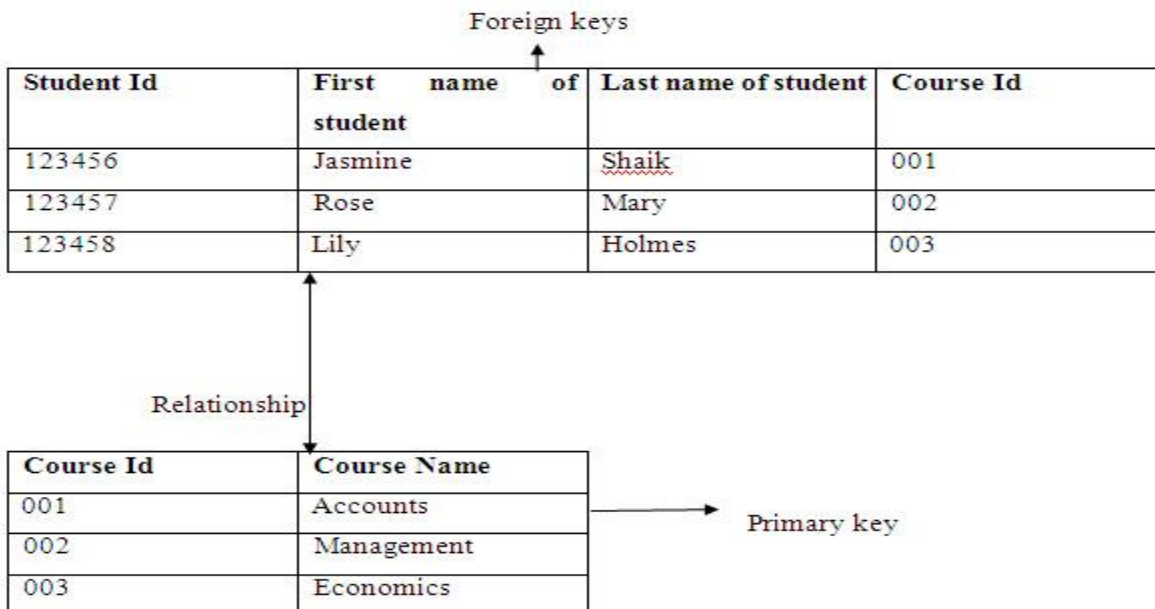✓ Excluding primary attributes in a table are non-prime attributes.
• **Example:**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Non prime attributes

8. **Foreign key:**
✓ Generally foreign key is a primary key from one table, which has a relationship with another table.
• **Example:**

Foreign keys

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Relationship

Course Id	Course Name
001	Accounts
002	Management
003	Economics

Primary key

21

9. **Simple key:**
   - ✓ Simple keys have a single field to specially recognize a record. The single field cannot be divided into more fields. Primary keys is also a simple key.
   - • **Example:** In the below example student id is a single field because no other student will have same Id. Therefore, it is a simple key.

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Simple key

10. **Compound key:**
   - ✓ Compound key has many fields to uniquely recognize a record.

11. **Artificial key:**
   - ✓ Artificial keys do not have meaning to the firms. They are allowed when
       - • No property has the parameter of primary key
       - • The primary key is huge and complex
   - **Example:** Table which has the details of the student has primary key but it is large and complex. The addition of row id column to it is the DBA's decision, where the primary key is row id.

Schema Diagrams :
   - ❖ A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by schema diagram.

branch-name	account-number	customer-name	customer-name
branch-city	branch-name	account-number	customer-street
assets	balance		customer-city
branch	account	depositor	

customer

loan-number	customer-name
branch-name	loan-number
amount	
loan	borrower
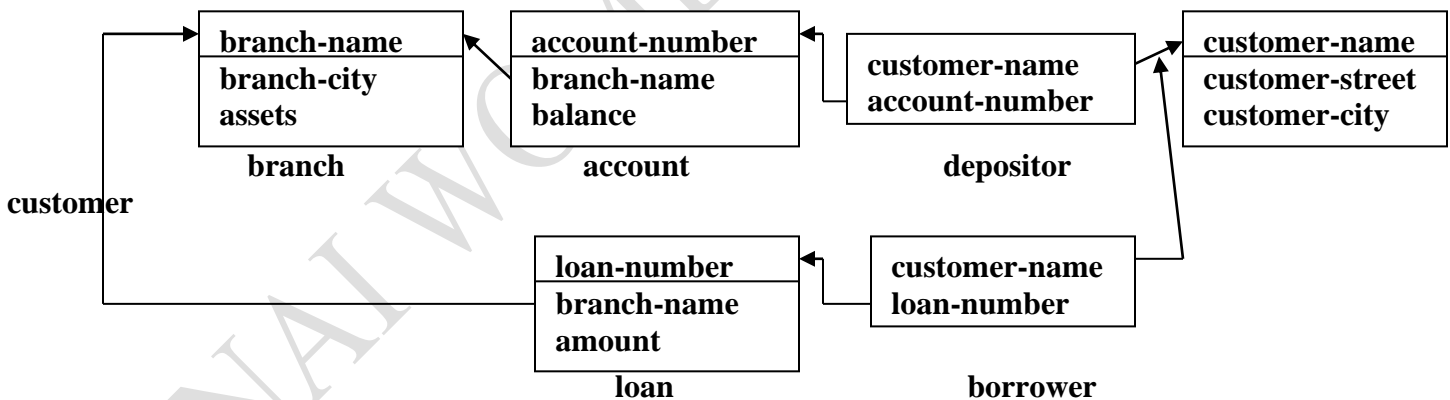
Query Languages:
   - ❖ A query language is a language in which a user requests information from the database. These languages higher than a standard programming language. It can be categorized into two languages. They are:
       - ▪ **Procedural Language**
       - ▪ **Nonprocedural Language**
   - ❖ In **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result.
   - ❖ In a **nonprocedural language**, the user describes the information desired, without giving a specific procedure for obtaining that information.
   - ❖ A compute data-manipulation language includes not only a query language, but also a language for database modification. Such languages include commands to insert and delete tuples as well as commands to modify parts of existing tuples.

BASIC RELATIONAL ALGEBRA OPERATIONS

1. What is Relational Algebra? (Part-A)
   List out the Fundamental Operations used in Relational Algebra. (Part-A)
   List out the Unary and Binary Operations (Part-A)

      A set of operations that take one or two relations as input and produce a new relation as their result is called as relational algebra. It is a procedural query language.

- ❖ The fundamentals operations in the relational algebra are:
  - **Select**
  - **Project**
  - **Union**
  - **set Difference**
  - **Cartesian Product**
  - **Rename**
- ❖ The operations use single relation is called as unary operations. The operations use pairs of relation is called as binary operations.
- ❖ The unary operations are:
  - **Select**
  - **Project**
  - **Rename**
- ❖ The binary operations are:
  - **Union**
  - **Set Difference**
  - **Cartesian Product**

Unary Operations:
Select Operation:

- ❖ The select operation selects tuples that satisfy a given predicate. The Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The relation is given in parentheses following the σ.

   Example:

- ❖ To find the tuples of loan relation where the branch is "Perry ridge"

$$\sigma_{\text{branch-name="Perryridge"}} (\text{loan})$$

- ❖ To find the tuples of loan relation where the amount is more than \$2000.

$$\sigma_{\text{amount>2000}} (\text{loan})$$

- ❖ To find the tuples of loan relation where the branch name is "Perryridge" and amount is more than \$ 2000.

$$\sigma_{\text{branch-name="Perryridge"}\wedge\text{amount>2000}}(\text{loan})$$

Project Operation:

- ❖ This operation is used to list the values of particular attribute. It is denoted by Greek letter pi ($\prod$).

   Example:

- ❖ To list all loan numbers and the amount of the loan can be written as

$$\prod_{\text{loan-number, amount}} (\text{loan})$$

- ❖ To find those customer who live in "Harrison"

$$\prod_{\text{customer-name}}(\sigma_{\text{customer-city="Harrison"}} (\text{customer}))$$

Rename Operation:

- ❖ The results of relational-algebra expressions do not have a name to refer to them. It is useful to be able to give them names.
- ❖ It is denoted by lower-case Greek letter rho (ρ). The relational-algebra expression E, returns the result of expression E under the name x.

$$\rho_x (E)$$

- ❖ The relational algebra expression E has arity n. The expression denoted as the following:

$$\rho_{x(A1,A2,.....An)} {}^{(E)}$$

❖ The above expression returns the result of expression E under the name x, and the attributes renamed to $A_1, A_2, \ldots, A_n$

Binary Operations:
Union Operation:
❖ This operation returns all rows that appear in either or both of two tables. It is a binary operation denoted by **U**.

Example:
❖ To find the names of all bank customers who have either an account or a loan or both.
$$\Pi_{customer-name} {}^{(borrower)} \quad U \quad \Pi_{customer-name} {}^{(depositor)}$$
❖ The borrower relation consists of two attributes. They are:
  ▪ customer-name
  ▪ loan-number
❖ The depositor relation consists of two attributes. They are:
  ▪ customer-name
  ▪ account-number
❖ The union operation **rUs** to be valid, the following condition hold:
  ▪ The relation r and s must be of the same arity.
  ▪ The domains of the ith attribute of r and the ith attribute of s must be the same for all i.

Set Difference Operation:
❖ This operation allows us to find tuples that are in one relation but are not in another. It is denoted by minus symbol (**-**). The expression r-s results in a relation containing those tuples in r but not in s.

Example:
❖ To find all customers of the bank who have an account but not a loan.
$$\Pi_{customer-name} {}^{(depositor)} \quad - \quad \Pi_{customer-name} {}^{(borrower)}$$

Cartesian Product:
❖ This operation denoted by a cross (**x**) allows us to combine information from any two relations. The Cartesian product of relations r_1 and r_2 as **r_1 x r_2**.

Example:
❖ Consider the r_1 relation consists of two attributes Person and Department. The r_2 relation consists of two attributes Person and Department.

Person	Department
Ram	Accounts
Sham	Sales

Person	Department
Radha	Accounts
Geetha	Sales

r_1 relation   **r_2 relation**

r_1_Person	r_1_Department	r_2_Person	r_2_Department
Ram	Accounts	Radha	Accounts
Ram	Accounts	Geetha	Sales
Sham	Sales	Radha	Accounts
Sham	Sales	Geetha	Sales

r_1 x r_2 relation

Formal Definition of the Relational Algebra:

❖ The above operations allow us to give a complete definition of an expression in the relational algebra. A basic expression in the relational algebra consists of either one of the following:
  ▪ A relation in the database
  ▪ A constant relation

❖ Let E_1 and E_2 be relational-algebra expressions. Then, the following are all relational-algebra expressions:

  ▪ E_1 U E_2
  ▪ $E_1 - E_2$
  ▪ $E_1 * E_2$
  ▪ $\sigma_p(E1)$, where p is a predicate on attributes in E_1
  ▪ $\pi_s(E1)$, where s is a list consisting of some of the attributes in E_1.
  ▪ $\rho_x(E1)$, where x is the new name for the result of E_1.

ADDITIONAL RELATIONAL ALGEBRA OPERATIONS
1. Discuss about the Additional Operations Used in Relational Algebra. (Part-C)

❖ The fundamental operations of the relational algebra are sufficient to express any relational-algebra query. The additional operations that do not add any power to the algebra, but that simplify common queries.

i. Set-Intersection Operation:

❖ This operation returns all rows that appear in both of two tables. It is denoted by ∩.

Example:

❖ To find all customers who have both a loan and an account.

$$\Pi_{\text{customer-name}} (\text{borrower}) \cap \Pi_{\text{customer-name}} (\text{depositor})$$

❖ Consider the two tables A and B contains the same attribute Cname and City.

Cname	City
Rupa	Pune
Subha	Mumbai

Cname	City
Ravi	Delhi
Rupa	Pune

Cname	City
Rupa	Pune

Relation A   **Relation B**   **A intersect B**

ii. Natural-Join Operation:

❖ It is a binary operation that allows to combine certain selections and a Cartesian product into one operation. It is denoted by "Join" symbol ⋈ .

❖ It forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

Example:

❖ Consider the following S1 and R1 relation. The S1 relation consists of Sid, Sname and Age attributes. The R1 relation consists of Sid and bid attributes. The equijoin expression **S1** ⋈ **R1.Sid=S1.Sid** **R1** is actually a natural join and can simply denoted by **S1** ⋈ **R1**.

Sid	Sname	Age
22	xxx	45
31	yyy	55
58	zzz	35

Sid	bid
22	101
58	103

S1 Relation   **R1 Relation**

❖ The Cartesian product of above two relations are:

Sid	Sname	Age	Sid	bid
22	xxx	45	22	101
22	xxx	45	58	103
31	yyy	55	22	101
31	yyy	55	58	103
58	zzz	35	22	101
58	zzz	35	58	103

S1XR1 (Cartesian Product)

❖ The Natural join **S1**⋈ **R1**denoted by the following:

Sid	Sname	Age	bid
22	xxx	45	101
58	zzz	35	103

$$S1 \bowtie_{R1.Sid=S1.Sid} R1$$

iii. Divison Operation:

❖ The division operator is useful for express some kinds of queries. The two relation A and B in which A has two fields x and y and B has just one field y, with the same domain as in A. The division operation A/B as the set of all x values such that for every y value in B, there is a tuple <X, Y> in A.

Example:

X	y
S1	P1
S1	P2
S2	P2
S3	P2
S4	P2
S1	P3
S4	P4

Relation A

y
P2

Relation B1

y
P2
P4

Relation B2

y
S1
S2
S3
S4

Y
S4

A/B1   **A/B2**

iv. Assignment Operation:

❖ It is used to express complex queries. It must always be made to a temporary relation variable. It does not provide any additional power to the algebra. It is denoted by ←similar to assignment in a programming language. The result of expression to the right of the ←is assigned to the relation variable on left of the ← . This relation variable may be used in subsequent expressions.

EXTENDED RELATIONAL ALGEBRA OPERATIONS

➢ The basic relational-algebra operations have been extended in several ways.
➢ A simple extension is to allow arithmetic operations as part of projection.
➢ An important extension is to allow aggregate operations such as computing the sum of the elements of a set, or their average.
➢ Another important extension is the outer-join operation, which allows relational-algebra expressions to deal with null values, which model missing information.

Generalized Projection

❖ The **generalized-projection** operation extends the projection operation by allowing arithmetic functions to be used in the projection list.

❖ The generalized projection operation has the form

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression, and each of F1, F2,... , Fn is an arithmetic expression involving constants and attributes in the schema of E. As a special case, the arithmetic expression may be simply an attribute or a constant.

customer-name	limit	credit-balance
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400

Figure 3.25   The *credit-info* relation.

Example:
- Suppose we have a relation credit-info, as in Figure 3.25, which lists the credit limit and expenses so far (thecredit-balance on the account).
- If we want to find how much more each person can spend, we can write the following expression:

$$\Pi_{\text{customer-name, limit} - \text{credit-balance}}(\text{credit-info})$$

- The attribute resulting from the expression limit − credit –balance does not have a name.
- We can apply the rename operation to the result of generalized projection in order to give it a name.
- As a notational convenience, renaming of attributes can be combined with generalized projection as illustrated below:

$$\Pi_{\text{customer-name, (limit} - \text{credit-balance) as credit-available}}(\text{credit-info})$$

- The second attribute of this generalized projection has been given the name credit- available.
- Figure 3.26 shows the result of applying this expression to the relation in Figure 3.25.

customer-name	credit-available
Curry	250
Jones	5300
Smith	1600
Hayes	0

Figure 3.26   The result of $\Pi_{\text{customer-name, (limit} - \text{credit-balance) as credit-available}}$ (credit-info).

Aggregate Functions
❖ It take a collection of values and return a single value as a result.
Example:
- The aggregate function **sum** takes a collection of values and returns the sum of the values.
- Thus, the function **sum** applied on the collection

$$\{1, 1, 3, 4, 4, 11\}$$

returns the value 24.
- The aggregate function **avg** returns the average of the values. When applied to the preceding collection, it returns the value 4.
- The aggregate function **count** returns the number of the elements in the collection, and returns 6 on the preceding collection.

- Other common aggregate functions include **min** and **max**, which return the minimum and maximum values in a collection; they return 1 and 11, respectively, on the preceding collection.

employee-name	branch-name	salary
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

Figure 3.27   The *pt-works* relation.

❖ To illustrate the concept of aggregation, we shall use the *pt-works* relation in Figure 3.27, for part-time employees. Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query is:

$$\mathcal{G}_{sum(salary)}(pt\text{-}works)$$

❖ The symbol G is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation G signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied.
❖ Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank.
❖ To do so, we need to partition the relation *pt-works* into **groups** based on the branch, and to apply the aggregate function on each group.
❖ The following expression using the aggregation operator G achieves the desired result:

$$_{branch\text{-}name}\mathcal{G}_{sum(salary)}(pt\text{-}works)$$

❖ In the expression, the attribute *branch-name* in the left-hand subscript of G indicates that the input relation *pt-works* must be divided into groups based on the value of *branch-name*. Figure 3.28 shows the resulting groups.

employee-name	branch-name	salary
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

Figure 3.28   The *pt-works* relation after grouping.

❖ The expression **sum**(*salary*) in the right-hand subscript of G indicates that for each group of tuples (that is, each branch), the aggregation function **sum** must be applied on the collection of values of the *salary* attribute.
❖ The output relation consists of tuples with the branch name, and the sum of the salaries for the branch, as shown in Figure 3.29.

branch-name	sum of salary
Austin	3100
Downtown	5300
Perryridge	8100

Figure 3.29 Result of $_{branch-name}\mathcal{G}_{sum(salary)}(pt\text{-}works)$.

❖ The general form of the **aggregation operation** G is as follows:

$$_{G_1,G_2,...,G_n}\mathcal{G}_{F_1(A_1),\ F_2(A_2),...,\ F_m(A_m)}(E)$$

where E is any relational-algebra expression; $G1,\ G2,...\ ,\ Gn$ constitute a list of at- tributes on which to group; each Fi is an aggregate function; and each Ai is an attribute name. The meaning of the operation is as follows. The tuples in the result of expression E are partitioned into groups in such a way that

1. All tuples in a group have the same values for $G1,\ G2,...\ ,\ Gn$.
2. Tuples in different groups have different values for $G1,\ G2,...\ ,\ Gn$.

Outer Join

❖ The **outer-join** operation is an extension of the join operation to deal with missing information.
❖ Suppose that we have the relations with the following schemas, which contain data on full-time employees:

employee (employee-name, street, city)
ft-works (employee-name, branch-name, salary)

❖ Consider the *employee* and *ft-works* relations in Figure 3.31.

employee-name	street	city
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	branch-name	salary
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

Figure 3.31 The *employee* and *ft-works* relations.

❖ Suppose that we want to generate a single relation with all the information (**street, city, branch name, and salary**) about full-time employees.
❖ A possible approach would be to use the **natural- join operation** as follows:

employee ⋈ ft-works

The result of this expression appears in Figure 3.32.

employee-name	street	city	branch-name	salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

Figure 3.32 The result of *employee* ⋈ *ft-works*.

❖ Notice that we have lost the street and city information about Smith, since the tuple describing Smith is absent from the *ft-works* relation; similarly, we have lost the branch name and salary information about Gates, since the tuple describing Gates is absent from the *employee* relation.
❖ We can use the **outer-join** operation to **avoid** this **loss of information**.

29

❖ There are actually three forms of the operation:
- **Left Outer Join** ⋈
- **Right Outer Join** ⋈ **And**
- **Full Outer Join** ⋈

❖ All three forms of outer join compute the join, and add extra tuples to the result of the join. The results of the expressions

employee ⋈ *ft –works,, employee* ⋈ *ft –works,* and *employee* ⋈ *ft –works*

appear in Figures 3.33, 3.34, and 3.35, respectively.

1. Left Outer Join:

❖ The **left outer join** ( ) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.

❖ In Figure 3.33, tuple (Smith, Revolver, Death Valley, *null*, *null*) is such a tuple.

❖ All information from the left relation is present in the result of the left outer join.

employee-name	street	city	branch-name	salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	null	null

Figure 3.33   Result of *employee* ⋈ *ft-works.*

2. Right Outer Join:

❖ The **right outer join** ( ) is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. In Figure 3.34, tuple (Gates, *null*, *null*, Redmond, 5300) is such a tuple.

❖ Thus, all information from the right relation is present in the result of the right outer join.

employee-name	street	city	branch-name	salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	null	null	Redmond	5300

Figure 3.34   Result of *employee* ⋈ *ft-works.*

3. Full Outer Join:

❖ The **full outer join**( ) does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join. Figure 3.35 shows the result

employee-name	street	city	branch-name	salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	null	null
Gates	null	null	Redmond	5300

Figure 3.35   Result of *employee* ⋈ *ft-works.*

❖ Since outer join operations may generate results containing null values.

NULL VALUES

- ❖ A value of NULL indicates that the value is unknown.
- ❖ A value of NULL is different from an empty or zero value.
- ❖ No two null values are equal. Comparisons between two null values, or between a NULL and any other value, return unknown because the value of each NULL is unknown.
- ❖ Null values generally indicate data that is unknown, not applicable, or that the data will be added later.

  Example: A customer's middle initial may not be known at the time the customer places an order.
- ❖ Following is information about nulls:
  - To test for null values in a query, use IS NULL or IS NOT NULL in the WHERE clause.
  - When query results are viewed in SQL Server Management Studio Code editor, null values are shown as **NULL** in the result set.
  - Null values can be inserted into a column by explicitly stating NULL in an INSERT or UPDATE statement, by leaving a column out of an INSERT statement, or when adding a new column to an existing table by using the ALTER TABLE statement.
  - Null values cannot be used for information that is required to distinguish one row in a table from another row in a table, such as primary keys.

MODIFICATION OF THE DATABASE

1. Explain about the Modification of the Database. (Part-B)
  - ✓ To express database modifications by using the assignment operation.

Deletion:

  - ✓ The users express a delete request in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database.
  - ✓ The user deletes only particular attributes. In relational algebra a deletion is expressed by
    - ▪ **r←r-E**

    Where r is a relation and E is relational-algebra query.

  Example:
  - ✓ Delete all of smith's account records.
    - ▪ **depositor←depositor -** $\sigma_{customer\text{-}name="Smith"}$ **(depositor)**
  - ✓ Delete all loans with amount in the range 0 to 50.
    - ▪ **loan←loan -** $\sigma_{amount>=0\ and\ amount<=50}$ **(loan)**

Insertion:

  - ✓ To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The relational algebra expresses an insertion by
    - ▪ r←**r U E**

    Where r is a relational and E is relational-algebra expression.

  Example:
  - ✓ To insert the value into the account relation by the following values: **(A-973,"perryridge",1200)**
    - ▪ **account←account U {(A-973,"perryridge",1200)}**

Updating:

  - ✓ To change a value in a tuple without changing all values in the tuple. We can use the generalized-projection operator to do this task:
    - ▪ **r←**$\pi_{F1,F2....Fn}$ **(r)**

  Example:
  - ✓ All balances of the account relation to be increased by 5 percent.
    - ▪ **account←**$\pi_{account\text{-}number,branch\text{-}name,balance\ *.05}$**(account)**

UNIT-III
SQL
BACKGROUND

1. Discuss about the Background of SQL. (Part-B)

List out and explain the Various Parts of SQL. (Part-B)

- ❖ The SQL established itself as the standard relational-database language. It consists of various versions. The original version was developed at IBM's **S**an **J**ose **R**esearch **L**aboratory.
- ❖ This language, originally called sequel in 1970s after that the name changed to SQL Structured Query Language.
- ❖ The **A**merican **N**ational **S**tandards **I**nstitute **(ANSI)** and the **I**nternational **S**tandards Organization **(ISO)** published an SQL standard, called SQL86. The current version of the **ANSI/ISO** SQL standard is the SQL-92.

Parts of SQL Language:

- ❖ The Structured Query Language consists of the following parts:
  - ▪ **Data-Definition Language (DDL)**
  - ▪ **Interactive Data-Manipulation Language (DML)**
  - ▪ **Embedded SQL**
  - ▪ **View Definition**
  - ▪ **Authorization**
  - ▪ **Integrity**
  - ▪ **Transaction Control**

Data-Definition Language (DDL):

- ❖ The SQL DDL provides commands for defining relation schemas, deleting relations, creating indices, and modifying relation schemas.

Interactive Data-Manipulation Language (DML):

- ❖ It includes a query language based on the relational algebra. It also includes commands to insert tuples, delete tuples and modify tuples in the database.

Embedded SQL:

- ❖ The embedded form of SQL is designed for use within general-purpose programming languages such as COBOL, Pascal, FORTRAN and C.

View Definition:

- ❖ The SQL DDL includes commands for defining views.

Authorization:

- ❖ The SQL DDL includes commands for specifying access rights to relations and views.

Integrity:

- ❖ The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy.

Transaction Control:

- ❖ The SQL includes commands for specifying the beginning and ending of transactions. Several implementations also allow explicit locking of data for concurrency control.

DATA-DEFINITION LANGUAGE

1. Explain about Data-Definition Language. (Part-C)

List out the Built-in Domain Types Used in SQL-92 Standard (Part-B)

Discuss about the Schema Definition in SQL. (Part-B)

- ❖ The set of relations in a database must be specified to the system by using a **Data Definition Language (DDL)**. It allows the specification of set of relations and information about each relation includes the following:
  - ▪ **The schema for each relation**
  - ▪ **The domain of values associated with each attribute**

- **The integrity constraints**
- **The set of indices to be maintained for each relation**
- **The security and authorization information for each relation**
- **The physical storage structure of each relation on disk**

Basic Domain Types in SQL:

❖ The SQL-92 standard supports a variety of built-in domain types, includes the following:
- **Char(n)** is a fixed-length character string, with user-specified length n. The full form is character.
- **Varchar(n)** is a variable-length character string, with user-specified maximum length n. The full form is character varying.
- **Int** is an integer.
- **Smallint** is a small integer.
- **Numeric (p,d)** is a fixed-point number, with user-specified precision. The number consists of p digits and d of the p digits are to the right of the decimal point.
- **Real, double Precision** are floating-point and double-precision numbers, with machine-dependent precision.
- **Float (n)** is a floating-point number, with user-specified precision of at least n digits.
- **Date** is a calendar date, containing year, month and day of the month.
- **Time** is the time of day, in hours, minutes, and seconds.

❖ The comparison by casting small integer x as an integer. This transformation is called as type coercion. It is used routinely in common programming languages and database systems.

❖ The SQL allows the domain declaration of an attribute to include the specification not null, and thus prohibits the insertion of a null value for this attribute.

Example:

❖ The SQL-92 allows us to define domains using a create domain clause.
- **create domain person-name char(15)**

Basic Schema Definition in SQL:

❖ To define the SQL relation using the create table command:
- **create table r ($A_1D_1,A_2D_2,….,A_nD_n$,**

    \<Integrity-constraint$_1$>,

    ….,

    \<Integrity-constraint$_k$>)

❖ Where **r** is the name of the relation, each A_i is the name of an attribute in the schema of relation **r**, and **D_i** is the domain type of values in the domain of attribute **A_i**. The allowed integrity constraints include the following:
- **Primary key ($A_{j1},A_{j2},…,A_{jm}$)**

    and

   Check(P)

❖ The primary key specification says that attributes **$A_{j1},A_{j2},…,A_{jm}$** form the primary key for the relation. It is optional. The attributes of a relation that are declared to be a primary key are requested to be not null and unique.

❖ The check clause specifies a predicate P that must be satisfied by every tuple in the relation. It is also ensure that attribute values satisfy specified conditions.

Example 1:

- **Create table Customer**

    (Customer-Name char(15) not null,
    Customer-Street char(30),
    Customer-City char(15),
    Primary key(Customer-Name))

- **Create table account**
  (Account-number char(15) not null,
  Branch-name char(10),
  Balance integer,
  Primary key(Account-number),
  Check(balance>=0))

❖ To remove a relation from the database, use the drop table command. It deletes all information about the dropped relation from the database.
- **Drop table r**
❖ The alter table command in SQL-92 to add attributes to an existing relation.
- **Alter table r add A D**
❖ In the above format r is the name of an existing relation, **A** is the name of the attribute to be added, and **D** is the domain of the added attribute. To drop attributes from a relation using a command.
- **Alter table r drop A**
Where **r** is the name of the existing relation and **A** is the name of an attribute of the relation.

BASIC STRUCTURE OF SQL QUERIES
1. Explain about the Basic Structure of SQL. (Part-C)
   Write down the Syntax for SQL Query. (Part-A)
◈ A relational database consists of a collection of relations, each of which is assigned a unique name. The basic structure of an SQL consists of three clauses:
- **Select**
- **From**
- **Where**

Format:

        Select $A_1,A_2,\ldots.,A_n$
        From $r_1,r_2,\ldots.,r_m$
        Where P

◈ In the above format each A_i represents an attribute, and each r_i is a relation and P is a predicate.
i) The Select Clause:
◈ The **Select** clause corresponds to the **projection** operation of the relational algebra. It is used to list the attributes desired in the result of a query. The result of SQL query is also a relation.
Example:
◈ To find the names of all branches in the loan relation
- **Select branch-name from loan**
In the above query the relation consisting of a single attribute with the heading branch-name
◈ To eliminate the duplicate values insert the keyword **distinct** after **select**. The above query written as
- **Select distinct branch-name from loan**
◈ The **select** clause also contains arithmetic expressions involving the operators **+, - , *,** and /.
- **Select branch-name, loan-number, amount*10 from loan**
◈ The asterisk symbol " ***** " used to denote **"all attributes"**. The form **select *** indicates that all attributes of all relations appearing in the **from** clause must be selected.
- **Select * from loan;**

ii) The Where Clause:
◈ The **where** clause corresponds to the **selection** predicate of the relational algebra. It consists of predicate involving attributes of the relations that appear in the from clause.
◈ The SQL uses the logical connectives **and**, **or**, and **not.** It also allows us to use the comparison operators to compare strings and arithmetic expression.

Example 1:
◈ To find all loan numbers for loans made at the perryridge branch with loan amounts greater than $1200.

▪ **Select loan-number from loan**
Where branch-name="perryridge" and amount>1200

Example 2:
◈ The SQL includes **between** comparison operator to simplify where clauses. It also contain **not between** operator.
◈ To find the loan numbers of those loans with amount between $90,000 and 1,00,000.

▪ **Select loan-number from loan**
Where amount between 90000 and 100000
(OR)
▪ **Select loan-number from loan**
Where amount<=100000 and amount>=90000

iii) The From Clause:
◈ The **from** clause corresponds to the **Cartesian-product** operation of the relational algebra. It lists the relation to be scanned in the evaluation of the expression. The users also use more than one relation in this clause.

Example:
▪ **Select Sname, S1.Sid from S1,R1 where S1.Sid=R1.Sid**

iv) Rename Operation:
◈ The SQL provide a mechanism for renaming the relations. It uses the **as** clause, taking the following format:
▪ **Rename old-name as new-name**

Example 1:
▪ **Rename S1 as S2**
◈ In the above query the relation name S1 is renamed in to S2.

Example 2:
▪ **Select Sname, S1.Sid as Sid-no from S1,R1 where S1.Sid=R1.Sid**

v) Tuple Variable:
◈ The **as** clause is particularly useful in defining the tuple variable. It is associated with a particular relation. It also defined in the **from** clause via the use of the as clause.
◈ It is defined in the **from** clause by placing it after the name of the relation with which it is associated, with the keyword as in between. The keyword **as** is optional.

Example:
▪ **Select Sname,T.Sid from S1 as T,R1 as S where T.Sid=S.Sid**

vi) String Operations:
◈ The commonly used operation on strings is Pattern matching. Using the operator **like**. The patterns using the following two special characters:
▪ **Percent(%):** The % character matches any substring.
▪ **Underscore(_):** The _character matches any character.
◈ The patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa.

Example 1:
▪ **'Perry%'** matches any string beginning with 'Perry'.
▪ **'%idge%'** matches any string containing 'idge' as a substring.
▪ **'_ _ _'** matches any string of exactly three characters.
▪ **'_ _ _%'** matches any string of at least three characters.

Example 2:
▪ **Select * from account where branch-name like 'Perry%'**
▪ **Select * from account where branch-name like '%idge%'**
▪ **Select * from account where branch-name like ' _ _ _ '**

35

- ▪ **Select * from account where branch-name like ' _ _ _ %'**
- ▪ **Select * from account where branch-name like '% e'**

vii) Ordering the Display of Tuples:

- ◈ The order by clause causes the tuples in the result of a query to appear in sorted order. To specify the sort order, specify **desc** for descending order or **asc** for ascending order.
- ◈ The ordering can be performed on multiple attributes. The sorting of large number of tuples may be costly, it is desirable to sort only when necessary.

Example 1:
- ▪ **Select * from loan order by amount desc**
- ▪ **Select * from loan order by loan-number asc**
- ▪ **Select * from loan order by amount desc, loan-number asc**

viii)   Duplicates:

- ◈ The use of relations with duplicates has proved useful in several situations. The SQL defines not only what tuples are in the result of a query, and also define how many copies of each of those tuple appear in the result.
- ◈ To define the duplicate semantics of an SQL query using multiset versions of the relational operators. The multiset relations r_1 and r_2,
  - ▪ If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.
- ◈ The SQL query of the form is following:

   Select $A_1, A_2, \ldots, A_n$
   From $r_1, r_2, \ldots, r_m$
   Where P

   is equivalent to the relational-algebra expression

   $$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

   Using the multiset versions of the relational operators σ, Π and **x**.

ADDITIONAL BASIC OPERATIONS

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

The Rename Operation

- ✓ Consider again the query that we used earlier:

   select name, course id
   from instructor, teaches
   where instructor.ID= teaches.ID;

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

   name, course id

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

   old-name as new-name

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

Example: If we want the attribute name *name* to be replaced with the name ***instructor name***, we can rewrite the preceding query as:

> **select** *name* **as** *instructor name*, *course id*
> **from** *instructor*, *teaches*
> **where** *instructor.ID= teaches.ID*;

- ✓ The **as** clause is particularly useful in renaming relations.

String Operation

- ✓ SQL specifies strings by enclosing them in single quotes.
     Example: 'Computer'
- ✓ A single quote character that is part of a string can be specified by using two single quote characters;   **Example:** the string "It's right" can be specified by "It"s right".
- ✓ SQL also permits a variety of functions on character strings, such as
  - ➤ Concatenating (using "_")
  - ➤ Extracting substrings,
  - ➤ Finding the length of strings,
  - ➤ Converting strings to Uppercase (using the function **upper**(*s*) where *s* is a string)
  - ➤ Converting strings to Lowercase (using the function **lower**(*s*)),
  - ➤ Removing spaces at the end of the string (using **trim**(*s*)) and so on.
- ✓ Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:
  - • **Percent (%):** The % character matches any substring.
  - • **Underscore ( ):** The character matches any character.
- ✓ Patterns are case sensitive; (i.e) uppercase characters do not match lowercase characters, or vice versa.
     Example:
  - • 'Intro%' matches any string beginning with "Intro".
  - • '%Comp%' matches any string containing "Comp" as a substring, for example,
    'Intro. to Computer Science', and 'Computational Biology'.
  - • ' ' matches any string of exactly three characters.
  - • ' %' matches any string of at least three characters.

Attribute Specification in Select Clause

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

> **select** *instructor.**
> **from** *instructor*, *teaches*
> **where** *instructor.ID= teaches.ID*;

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

Ordering the Display of Tuples

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

> **department, we write:**
> **select** *name*
> **from** *instructor*
> **where** *dept name* = **'Physics'**
> **order by** *name*;

- ✓ By default, the order by clause lists items in ascending order.
- ✓ To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order.
     Example:

> **select** *

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

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

> **select name**
> **from instructor**
> **where salary between 90000 and 100000;**

instead of:

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

SET OPERATIONS

1. Explain about Set Operations. (Part-B)
- ❖ The following set operations used in Structured Query Language (SQL):
  - ▪ **Union**
  - ▪ **Intersect**
  - ▪ **Except**

Union:
- ❖ This operation returns all rows that appear in either or both of two tables. It automatically eliminates duplicates, unlike the select clause.

Example 1:
- ▪ **(select customer-name from depositor) union**
  (select customer-name from borrower)

Example 2:
  - o If we want to retain all duplicates use **union all** in place of union:
    - ▪ **(select customer-name from depositor) union all**
      (select customer-name from borrower)

Intersect:
- ❖ This operation returns all rows that appear in both of two tables. It automatically eliminates duplicates. To retain the duplicates use **intersect all** in place of intersect.

Example 1:
- ▪ **(select customer-name from depositor) intersect**
  (select customer-name from borrower)

Example 2:
- ❖ If we want to retail all duplicates, we must write **intersect all** in place of **intersect.**
  - ▪ **(select customer-name from depositor) intersect all**
    (select customer-name from borrower)

Except:
- ❖ This operation returns all rows appear in one table not in other table. It automatically eliminates duplicates. To retain the duplicates use except all in place of except.

Example 1:
- ▪ **(select customer-name from depositor) except**
  (select customer-name from borrower)

Example 2:
- ▪ **(select customer-name from depositor) except all**
  (select customer-name from borrower)

<div align="center">

AGGREGATE FUNCTIONS
</div>

1. List out and explain the Aggregate Functions Used in SQL. (Part-B)
  Discuss about Group by Function. (Part-B)
  ❖ The aggregate functions are functions that take a collection of values as input and return a single value as output. It consists of the following built-in aggregate functions:
  - **Average: avg**
  - **Minimum: min**
  - **Maximum: max**
  - **Total: sum**
  - **Count: count**

  i. Average (avg):
  ❖ This function takes collection of numbers as input and returns average value of particular attribute as output.
   Example 1:
  - **select avg(balance) from account**
   Example 2
  - **select avg(balance) from account where branch-name='Perryridge'**

  ii. Minimum (min):
  ❖ It returns the minimum value of particular attribute.
   Example:
  - **select min(balance) from account**

  iii. Maximum (max):
  ❖ It returns the maximum value of particular attribute.
   Example:
  - **select max(balance) from account**

  iv. Total (sum):
  ❖ It returns the total value of particular attribute.
   Example:
  - **select sum(balance) from account**

  v. Count:
  ❖ It returns or count the number of records in the particular relation.
   Example:
  - **select count(*) from customer**

  vi. Group by Clause:
  ❖ The grouping is an additional feature of relational algebra. It combines the values based on a common reference. It is used to group set of tuples. The tuples with the same value on all attributes in the group by clause are placed in one group.
  ❖ The user also specifies the predicate in this clause after apply the formation of groups. The keyword **having** used to represent the condition.
   Example 1:
  - **select branch-name,avg(balance) from account**
    group by branch-name
   Example 2:
  - **select branch-name,avg(balance) from account**
    group by branch-name having avg(balance)>200

NULL VALUES
1. Discuss about Null Values With Example. (Part-B)
◈ The **null** value indicates the absence of information about the value of an attribute. The special keyword null in a predicate to test for a null value. The predicate **not null** tests for the absence of null values.

◈ All aggregate function except **count**(*) ignore null values in their input collection.
Example 1:
  ▪ **select loan-number from loan where amount is null**
Example 2:
  ▪ **select loan-number from loan where amount is not null**

NESTED SUBQUERIES
1. Discuss about the Nested Subqueries With Example. (Part-C)
What is Subquery & Nested Query? (Part-A)
❖ A subquery is a **select-from-where** expression that is nested within another query. The one SELECT statement inside another is called as nested query.
❖ A common use of subqueries is to perform tests for **set membership**, **set comparisons** and **set cardinality.**

Set Membership:
❖ The **in** connective tests for set membership, where the set is a collection of values produced by select clause. The **not in** connective tests for the absence of set membership.
Example 1:
  ▪ **select customer-name from borrower**
    where customer-name in(select customer-name from depositor)
Example 2:
  ▪ **select customer-name from borrower**
    where customer-name not in(select customer-name from depositor)

Set Comparison:
❖ It is used to compare the inner and outer query. The inner query executed first and then executed the outer query.
Example 1:
  ▪ **select name, total from student**
    where total> (select avg(total) from student)
Example 2:
❖ The greater than at least one is represented in SQL by >**some**.
  ▪ **select branch-name from branch**
    where assets>some (select assets from branch where branch-city='Brooklyn')
❖ In the above example the > some comparison in the where clause of the other select is true if the assets value of the tuple is greater than at least one member of the set of all asset values for branches in Brooklyn.
Example 3:
❖ The construct >all corresponds to the phrase "greater than all". The SQL allows <all, <=all, >=all, =all, and <> all comparisons. The **<> all** is identical to **not in**.
  ▪ **select branch-name from branch**
    where assets>all(select assets from branch where branch-city='Brooklyn')

Testing for Empty Relations:

❖ To test the existence and nonexistence of tuples in a subquery by using **exists** and **not exists** construct. The SQL includes a feature for testing whether a subquery has any tuples in its result. The exists construct returns the value true if the argument subquery is nonempty.

Example:

▪ **select customer-name from borrower**
where exists(select * from depositor
where depositor.customer-name=borrower.customer-name)

Test for the Absence of Duplicate Tuples:

❖ The SQL includes a feature for testing whether a subquery has any duplicate tuples in its result. The unique construct returns the value true if the argument subquery contains no duplicate tuple.

Example:

▪ **select customer-name from borrower**
where unique(select * from depositor
where depositor.customer-name=borrower.customer-name)

❖ In the above example the user also specify the not unique construct for test the existence of duplicate tuples in a subquery.

MODIFICATION OF THE DATABASE

1. How can we modify the Information in the Database? (Part-C)
Explain about the Modification of Database. (Part-C)

◈ The users extract the information from the database and add, remove, or change information using Structured Query Language.

Deletion:

◈ A delete request is expressed in much the same way as a query. Using this delete the whole tuples and cannot delete value on only particular attributes.

◈ This command operates on only one relation. To delete tuples from several relations, use one delete command for each relation. It is expressed by the following format:

▪ **delete from r**
where P

◈ In the above format the P represents a predicate and r represents a relation. This statement first finds all tuples t in r for which P(T) is true, and then deletes from r. The where clause can be omitted, all tuples in **P** are deleted.

Example 1:

▪ **delete from account where branch-name='Perryridge'**

Example 2:

▪ **delete from loan where amount between 1300 and 1500**

Example 3:

◈ The delete request can contain a nested select that references the relation from which tuples are to be deleted.

▪ **delete from account where balance>(select balance from account)**

Insertion:

◈ To insert data into a relation, specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The insert statement is a request to insert one tuple.

▪ **insert into account values ('Perryridge','A-123',1200)**

◈ The attributes to be specified as part of the insert statement. The above example also written by

▪ **insert into account(branch-name,account-number,balance)**
values('Perryridge','A-123',1200)   (OR)
▪ **insert into account(account-number, branch-name,balance)**
values('A-123', 'Perryridge',1200)

◈ The set of tuples also inserted into account relation using the select statement.
- **insert into account select * from account**

◈ To insert tuples to be only on some attributes of the schema. The remaining attribute are assigned a null value denoted by null.
- **insert into account values(null,'A-124',1200)**

Updates:

◈ This statement is used to change a value in a tuple without changing all values in the tuple.

Example 1:
- **update account set balance=balance*1.05**

Example 2:
- **update account set balance=balance*1.5 where balance>1000**

Example 3:
- **update account set balance=balance*0.5**
  where balance> select avg(balance) from account

Update of a View:

◈ The view-update anomaly becomes more difficult to handle when a view is defined in terms of several relations.

Example:
- **create view v3 as select branch-name, loan-number from loan**

◈ The SQL allows a view name to appear wherever a relation name is allowed. The insertion is represented by an
- **insert into v3 values('Perryridge', L-103)**

insertion into the relation loan, since loan is the actual relation from which the view v3 is constructed.

Transactions:

◈ A transaction consists of a sequence of query and / or update statements. The SQL standard specifies that a transaction begins implicitly when as SQL statement is executed. One of the following SQL statements must end the transaction:
- **Commit** work commits the current transaction; it makes the updates performed by the transaction become permanent in the database.
- **Rollback** work causes the current transaction to be rolled back; that is, it undoes all updates performed by the SQL statements in the transaction.

JOINED RELATIONS

1. Explain about Joined Relations. (Part-C)
Discuss about the Join Types and Conditions. (Part-C)

❖ The SQL-92 provides various other mechanisms for joining relations, including condition joins, and natural joins and various forms of outer joins.
❖ The join operations take two relations and return as a result another relation. The join operations in SQL-92 consist of a **join type** and a **join condition**.
❖ The join condition defines which tuples in the two relations match, and what attributes are present in the result of the join. The join type defines how tuples in each relation that do not match any tuple in the other relation.

Join Types:

❖ It consists of the following types:
- **Inner Join**
- **Natural Inner Join**
- **Left Outer Join**
- **Right Outer Join**
- **Full Outer Join**

Inner Join:

❖ The attributes of the result consist of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation. The common attributes appears twice in the result.

Example:

branch-name	loan-number	amount
Downtown	L-171	3000
Redwood	L-231	4000
Perryridge	L-261	1700

loan

customer-name	loan-number
Jones	L-171
Smith	L-231
Hayes	L-156

borrower

branch-name	loan-number	amount	customer-name	loan-number
Downtown	L-171	3000	Jones	L-171
Redwood	L-231	4000	Smith	L-231

loan inner join borrower on loan.loan-number=borrower.loan-number

Natural Inner Join:

❖ In this join the common attributes appears only once in the result of the natural join.

branch-name	loan-number	amount	customer-name
Downtown	L-171	3000	Jones
Redwood	L-231	4000	Smith

loan natural inner join borrower

Left Outer Join:

❖ The attribute of tuples r that are derived from the left-hand-side relation are filled in with the values from tuple t, and the remaining attributes of r are filled with null values.

branch-name	loan-number	amount	customer-name	loan-number
Downtown	L-171	3000	Jones	L-171
Redwood	L-231	4000	Smith	L-231
Perryridge	L-261	1700	Null	Null

loan left outer join borrower on loan.loan-number=borrower.loan-number

Right Outer Join:

❖ This type is symmetric to the left outer join. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the   right-outer-join.

❖ The attributes $A_1, A_2, \ldots, A_n$ must consists of only attributes that are common to both relations, and they appear only once in the result of the join.

branch-name	loan-number	amount	customer-name
Downtown	L-171	3000	Jones
Redwood	L-231	4000	Smith
Null	L-156	Null	Hayes

43

Full Outer Join:
- ❖ It is a combination of the left and right outer-join types. After the result of the inner join is computed, tuples from left-hand-side relation that did not match with any from the right-hand-side are extended with nulls are added to the result.
- ❖ The tuples from right-hand-side relation that did not match with any tuples from the left-hand-side relation are also extended with nulls, and are added to the result.

branch-name	loan-number	amount	customer-name
Downtown	L-171	3000	Jones
Redwood	L-231	4000	Smith
Perryridge	L-261	1700	Null
Null	L-156	Null	Hayes

loan full outer join borrower using(loan-number)

AUTHORIZATION IN SQL

1. Explain about the Mechanism for Defining Authorizations. (Part-C)
  Explain about the Authorization in SQL. (Part-C)

Authorization:
- ✓ To assign users for several forms of authorization on parts of the database.
  - ▪ **Read Authorization** allows reading, not modification of data.
  - ▪ **Insert Authorization** allows insertion of new data not modification of existing data.
  - ▪ **Update Authorization** allows modification, not allow deletion of data.
  - ▪ **Delete Authorization** allows deletion of data.

- ❖ The SQL language provides powerful mechanisms for defining authorizations.

Privileges in SQL:
- ❖ The SQL standard includes the privileges delete, insert, select and update. The **select** privilege corresponds to the read privilege. A **references** privilege that permits a user/role to declare foreign keys when creating relations.
- ❖ The SQL data-definition language includes commands to grant and revoke privileges. The grant statement is used to confer authorization. The basic form of this statement is:
  - ▪ **Grant <privilege list> on <relation name or view name> to <user/role list>**
- ❖ The privilege list allows the granting of several privileges in one command. The below grant statement grants users U1,U2 and U3 select authorization on account relation:
  - ▪ **Grant select on account to U1,U2,U3**
- ❖ The update authorization may be given either on all attributes of the relation or only some. The below grant statement gives users U1,U2 and U3 update authorization on the **amount** attribute of the **loan** relation:
  - ▪ **Grant update (amount) on loan to U1,U2,U3**
- ❖ The SQL references privilege is granted on specific attributes like update privilege. The following grant statement allows user U1 to create relations that reference the key branch-name of the branch relation as a foreign key:
  - ▪ **Grant references (branch-name) on branch to U1.**

Roles:
- ❖ The privileges of a user or a role consist of
  - ▪ All privileges directly granted to the user/roles.
  - ▪ All privileges granted to roles that have been granted to the user/role.
- ❖ The roles can be created in SQL: 1999 as follows:
  - ▪ **Create role** teller

❖ The roles can then be granted privileges like users, the statement are:
- **Grant select on** account **to** teller

❖ Roles can be assigned to the users, to some other roles:
- Grant teller to john
- Create role manager
- Grant teller to manager
- Grant manager to mary

The Privilege to Grant Privileges:

❖ If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate grant command.
- **Grant select on branch to U1 with grant option**

❖ To revoke an authorization, use the revoke statement. It takes the form almost identical to that of grant.
- **Revoke <privilege list> on <relation name or view name> from <user/role list> [restrict|cascade]**

❖ To revoke the privileges that we granted previously, by
- Revoke select on branch from U1,U2,U3.
- Revoke update (amount) on loan from U1, U2, U3.
- Revoke references (branch-name) on branch U1.

❖ To revoke statement may alternatively specify **restrict**:
- **Revoke select on** branch from U1,U2,U3 **restrict**

the system returns an error if there are any revokes, and does not carry out the revoke action.

Other Features:

❖ The creator of an object gets all privileges on the object, including the privilege to grant privileges to others.

❖ The SQL standard specifies the authorization mechanism for the database schema: only one owner of the schema can carry out any modification to the schema.

Limitations of SQL Authorization:

❖ The task of authorization falls on the application server; the entire authorization scheme of SQL is by passed. The problems are these:
- The code for checking authorization becomes intermixed with the rest of the application code.
- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes.

VIEWS

1. Explain about Views. (Part-B)
What is View? (Part-A)

⊙ A view is a logical table that derives its data from other tables. It does not contain any data of its own. Its content are taken from other tables through the execution of a query. The other table that provides data to a view are called base tables. The base tables usually contain data.

⊙ It is temporarily populated. When the lifetime of the query is over, the data in the view is discarded.

⊙ To define a view using the create view statement. The format of create view statement is
- **create view v as <query expression>**

Where <query expression> is any legal relational-algebra expression. The view name represented by v.

Example 1:

create view v1 as (select * from account);

select * from v1;

⊙ In the above query display all the records in the base table account.

Example 2:

        create view v2 as select Sid, Sname from S1;

        select * from v2;

⊙ In the above query display all the records in the particular attributes **(Sid, Sname).**

Example 3:

        create view v3 as select bid from R1 where bid=103;

        select * from v3;

⊙ In the above query display the record where the bid value is 103.

Updating Data through Views:

⊙ Whenever the user updates view, it updates data in the base table.

Example:

        create view v4 as select city from A;

        select * from A;

        update v4 set city='kanpur' where city='pune';

⊙ In the above update query will be executed successfully the modification will be affected in the base table.

Views Defined Using Other Views:

⊙ The one view may be used in the expression defining another view. A view relation v_1 is said to depend directly on a view relation v_2 if v_2 is used in the expression defining v_1. A view relation v_1 is said to depend on view relation v_2 if and only if there is a path in the depending graph from v_2 to v_1.

⊙ The view v_1 depends on view v_2 if either v_1 depends directly on v_2, or there is a sequence of view relations $r_1, r_2, \ldots, r_{n-1}, r_n$ such that v_1 depends directly on r_1, r_1 depends directly on r_2, and so on, until r_{n-1} depends directly on r_n, and r_n depends directly on v_2.

⊙ A view relation v is said to be recursive if it depends on itself. The depending graph would have a cycle involving v if and only if v is recursive. The dependency graph would have a cycle involving v if and only if v is recursive.

⊙ Consider the below graph the view all-customer depends directly on the relation borrower and loan, the all-customer uses the other two views. The view **xcustomer** depends on the view all-customer.

           xcustomer

                                 Figure: Dependency Graph

         all-customer

        borrower loan
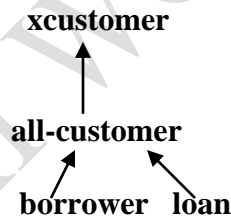
⊙ The above graph has a node for each view, and a directly edge from view v2 to view v_1 if v_1 depends directly on v_2.

INTEGRITY CONSTRAINTS

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

Examples

      • An instructor name cannot be *null*.

      • No two instructors can have the same instructor ID.

      • Every department name in the *course* relation must have a matching department name in the *department* relation.

      • The budget of a department must be greater than \$0.00.

❖ Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations.

Constraints on a Single Relation

❖ We described in Section 3.2 how to define tables using the **create table** command.
❖ The **create table** command may also include integrity-constraint statements.
❖ In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command.
❖ The allowed integrity constraints include
- **not null**
- **unique**
- **check**(<predicate>)

a) Not Null Constraint

➤ The null value is a member of all domains, and as a result is a legal value for every attribute in SQL by default.
➤ Consider a tuple in the *student* relation where *name* is *null*. Such a tuple gives student information for

an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be *null*.

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

➤ The **not null** specification prohibits the insertion of a null value for the attribute.

b) Unique Constraint

➤ SQL also supports an integrity constraint:

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

➤ The **unique** specification says that attributes $Aj1$, $Aj2$, . . . , Ajm form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes.

c) The check Clause

➤ When applied to a relation declaration, the clause **check**(*P*) specifies a predicate *P* that must be satisfied by every tuple in a relation.
➤ A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system.
➤ For instance, a clause **check** (*budget* > 0) in the **create table** command for relation *department* would ensure that the value of **budget** is nonnegative.

As another example, consider the following:

create table section
(course id **varchar** (8),
sec id **varchar** (8),
semester **varchar** (6),
year **numeric** (4,0),
building **varchar** (15),
room number **varchar** (7),
time slot id **varchar** (4),
primary key (course id, sec id, semester, year),
check (semester **in** ('Fall', 'Winter', 'Spring', 'Summer')));

d) Referential Integrity

➤ Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
➤ This condition is called **referential integrity**.
➤ Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause. We illustrate foreign-key declarations by using the SQL DDL definition of part of our university database, shown in Figure 4.8.

```
create table classroom
(building varchar (15),
room number varchar (7),
capacity numeric (4,0),
primary key (building, room number))


create table department
(dept name varchar (20),
building varchar (15),
budget numeric (12,2) check (budget > 0),
primary key (dept name))


create table course
(course id varchar (8),
title varchar (50),
dept name varchar (20),
credits numeric (2,0) check (credits > 0),
primary key (course id),
foreign key (dept name) references department)


create table instructor
(ID varchar (5),
name varchar (20), not null
dept name varchar (20),
salary numeric (8,2), check (salary > 29000),
primary key (ID),
foreign key (dept name) references department)


create table section
(course id varchar (8),
sec id varchar (8),
semester varchar (6), check (semester in
('Fall', 'Winter', 'Spring', 'Summer'),
year numeric (4,0), check (year > 1759 and year < 2100)
building varchar (15),
room number varchar (7),
time slot id varchar (4),
primary key (course id, sec id, semester, year),
foreign key (course id) references course,
foreign key (building, room number) references classroom)
```

Figure 4.8 SQL data definition for part of the university database.

- ➤ The definition of the *course* table has a declaration "**foreign key** (*dept name*) **references** *department*".
- ➤ Requirements of this form are called **referential-integrity constraints**, or **subset dependencies**.
- ➤ We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

  dept name **varchar**(20) **references** *department*

- ➤ When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back).

> However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *course*:

> **create table** *course*
> (. . .
> **foreign key** (*dept name*) **references** *department*
> **on delete cascade**
> **on update cascade**,
> . . .);

Integrity Constraint Violation During a Transaction

> Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation.
> For instance, suppose we have a relation *person* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *person*.

Assertion

> An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
> Domain constraints and referential-integrity constraints are special forms of assertions.
> **Two examples** of such constraints are:
>   • For each tuple in the *student* relation, the value of the attribute *tot cred* must equal the sum of credits of courses that the student has completed successfully.
>   • An instructor cannot teach in two different classrooms in a semester in the same time slot.
> An assertion in SQL takes the form:
>   **create assertion** <assertion-name> **check** <predicate>;
> In Figure 4.9, we show how the first example of constraints can be written in SQL. Since SQL does not provide a "for all *X*, *P*(*X*)" construct (where *P* is a predicate), we are forced to implement the constraint by an equivalent construct, "not exists *X* such that not *P*(*X*)", that can be expressed in SQL.

> **create assertion** *credits earned constraint* **check**
> (**not exists** (**select** ID
> **from** *student*
> **where** *tot cred* <> (**select sum**(*credits*)
> **from** *takes* **natural join** *course*
> **where** *student.ID*= *takes.ID*
> **and** *grade* **is not null and** *grade*<> 'F')
Figure 4.9 An assertion example.

SQL DATA TYPES AND SCHEMAS

> A number of built-in data types supported in SQL such as integer types, real types, and character types.
> There are additional built-in datatypes supported by SQL, which we describe below.

a) Date and Time Types in SQL

❖ The SQL standard supports several data types relating to dates and times:
   • **date**: A calendar date containing a (four-digit) year, month, and day of the month.
   • **time**: The time of day, in hours, minutes, and seconds. A variant, **time**(*p*), can be used to specify   the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying **time with timezone**.

• **timestamp**: A combination of **date** and **time**. A variant, **timestamp**(*p*), can be used to specify the number of fractional digits for seconds (the default here being 6). Time-zone information is also stored if **with timezone** is specified.

❖ Date and time values can be specified like this:

> **date** '2001-04-25'
> **time** '09:30:00'
> **timestamp** '2001-04-25 10:29:01.45'

❖ Dates must be specified in the format year followed by month followed by day, as shown.

❖ The seconds field of time or timestamp can have a fractional part, as in the timestamp above.

b) Default Values

❖ SQL allows a default value to be specified for an attribute as illustrated by the following create table statement:

> **create table student**
> **(ID varchar (5),**
> **name varchar (20) not null,**
> **dept name varchar (20),**
> **tot cred numeric (3,0) default 0,**
> **primary key (ID));**

❖ The default value of the tot cred attribute is declared to be 0. As a result, when a tuple is inserted into the student relation, if no value is provided for the tot cred attribute, its value is set to 0.

❖ The following insert statement illustrates how an insertion can omit the value for the tot_cred attribute.

> **insert into student(ID, name, dept_name)**
> **values ('12789', 'Newman', 'Comp. Sci.');**

c) Index Creation

❖ An index on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.

For example, if we create in index on attribute ID of relation student, the database system can find the record with any specified ID value, such as 22201, or 44553, directly, without reading all the tuples of the student relation.

❖ An index can also be created on a list of attributes

For example, on attributes name, and dept_name of student.

❖ Although the SQL language does not formally define any syntax for creating indices, many databases support index creation using the syntax illustrated below.

> **create index studentID_index on student(ID);**

❖ The above statement creates an index named studentID index on the attribute ID of the relation student.

d) Large-Object Types

❖ Many current-generation database applications need to store attributes that can be **large** (of the order of many kilobytes), such as a photograph, or **very large** (of the order of many megabytes or even gigabytes), such as a high-resolution medical image or **video clip**.

❖ SQL therefore provides large-object data types for character data (clob) and binary data (blob).

❖ The letters "lob" in these data types stand for "Large OBject."

❖ For example, we may declare attributes

> **book review clob(10KB)**
> **image blob(10MB)**
> **movie blob(2GB)**

e) **User-Defined Types**
   - ❖ SQL supports two forms of user-defined data types.
     - i) The first form, which we cover here, is called distinct types.
     - ii) The other form, called structured data types, allows the creation of complex data types with nested record structures, arrays, and multisets.
   - ❖ It is possible for several attributes to have the same data type.
   - ❖ **For example**, the name attributes for student name and instructor name might have the same domain: the set of all person names.
   - ❖ The create type clause can be used to define new types.
   - ❖ **For example,** the statements:

     create type Dollars as numeric(12,2) final;
     create type Pounds as numeric(12,2) final;

     define the user-defined types Dollars and Pounds to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point.
   - ❖ The newly created types can then be used, as types of attributes of relations.
   - ❖ **For example,** we can declare the department table as:

     create table department
     (dept name varchar (20),
     building varchar (15),

f) **Create Table Extensions**
   - ❖ Applications often require creation of tables that have the same schema as an existing table.
   - ❖ SQL provides a create table like extension to support this task:

     create table temp instructor like instructor;
   - ❖ The above statement creates a new table temp instructor that has the same schema as instructor.
   - ❖ **For example,** the following statement creates a table t1 containing the results of a query.

     create table t1 as
     (select *
     from instructor
     where dept name= 'Music')
     with data;

UNIT-III COMPLETED

<div align="center">

UNIT-IV

RELATIONAL LANGUAGES

TUPLE RELATIONAL CALCULUS
</div>

1. Explain about Tuple Relational Calculus. (Part-C)
- ✓ It is a non-procedural query language. It describes the desired information, without giving a specific procedure for obtaining that information.
- ✓ A query in the tuple relational calculus is expressed as
  - **{T|P (T)}**

  that is, it is the set of all tuples t such that predicate P is true for t.

i) Example Queries:
- ✓ To find the branch-name, loan-number and amount for loans of over \$1200.
  - **{t| t ε loan Λ t[amount]>1200}**
- ✓ Suppose we need those tuples on (loan-number) such that there is a tuple in loan with the amount attribute >1200. To express this request, we need the construct "there exists" from mathematical logic. The notation
  - **∃t ε r (Q(t))**

  means "there exists a tuple t in relation r such that predicate **Q(t)** is true.
- ✓ To find the loan-number for loan of an amount greater than \$1200.
  - **{t| ∃ s ε loan (t[loan-number]=s[loan-number] Λ s[amount]>1200)}**

ii) Formal Definition:
- ✓ A tuple relational calculus expression is of the form
  - **{T|P(T)}**

  where P is a formula. Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by ∃ or ∀. Thus, in

  t ε loan Λ ∃ s ε customer (t[branch-name]=s[branch-name])

  t is a free variable. Tuple variable s is said to be a bound variable.
- ✓ We build up formula from atoms by using the following rules:
  - An atom is a formula.
  - If p_1 is a formula, then so are $\neg p_1$ and (p_1).
  - If p_1 and p_2 are formula, then so are $p_1 \vee p_2$, $p_1 \wedge p_2$, and $p_1 => p_2$.
  - If $p_1(s)$ is a formula containing a free tuple variable s, and r is a relation, then
    - **∃ s ε r(p_1(s)) and ∀s ε r (p_1(s))**

    are also formula. In the tuple relational calculus, these equivalence include the following three rules:
      - $p_1 \wedge p_2$ is equivalent to $\neg (\neg (p_1) \vee \neg (p_2))$
      - ∀ tεr (p_1(t)) is equivalent to $\neg \exists$ **t ε** r($\neg p_1$(t))
      - p1=>p2 is equivalent to $\neg (p_1) \vee p2$.

iii) Safety of Expressions:
- ✓ A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression   **{t/$\neg$ (tεloan)}**
- ✓ To define a restriction of the tuple relational calculus, we introduce the concept of the domain of a tuple relational formula, P.
- ✓ The domain of P, denoted dom(P), is the set of all values referenced by p.

<div align="center">52</div>

iv) Expressive Power of Languages:

- ✓ The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra.
- ✓ Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expressions in the tuple relational calculus, and for every tuple- relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression.

DOMAIN RELATIONAL CALCULUS

1. Explain about Domain Relational Calculus. (Part-C)
Write down the Format of Domain Relational Calculus. (Part-A)

- ✓ It uses domain variables that take on values from an attributes domain, rather then values for an entire tuple. This calculus is closely related to the tuple relational calculus.

Formal Definition:

- ✓ An expression in the domain relational calculus is of the form

$$\{<x_1, x_2, \text{--------} x_n> | p(x_1, x_2, \text{------} x_n)\}$$

where $x_1, x_2, \text{----} x_n$ represent domain variables. P represents a formula composed of atoms. An atom in the domain relational calculus has one of the following forms:

- ▪ $<x_1, x_2, ... x_n> \varepsilon$ Rel, where Rel is a relation with n attributes; each x_i, **1≤i≤n** is either a variable or a constant.
- ▪ x op y
- ▪ x op constant, or constant op x

- ✓ A formula is recursively defined to be one of the following, where p and q are themselves formulas, and P(X) denotes a formula in which the variable X appears:

- ▪ Any atomic formula
- ▪ $\neg p$, $p \wedge q$, pvq, or p=>q
- ▪ X(P(X)), Where X is the domain variable
- ▪ X(P(X)), where X is a domain variable

Example Queries:

- ✓ To find the loan no, branch-name, and amount for loan of over \$1200.

$$\{<l,b,a>|<l,b,a>\varepsilon \text{ loan} \wedge a>1200\}$$

- ✓ To find all loan numbers for loans with an amount greater than \$1200.

$$\{<l>|\exists b,a(<l,b,a>\varepsilon loan \wedge a >1200\}$$

Safety of Expressions:

- ✓ A domain relational calculus expression may generate an infinite relation. An expression such as

$$\{<l,b,a>| \neg (<l,b,a) \varepsilon loan)\}$$

is unsafe, because it allows values in the result that are not in the domain of the expression.

Expressive Power of Languages:

- ✓ When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus to safe expressions. The restricted tuple relational calculus is equivalent to the relational algebra; all three of the following are equivalent:

- ▪ The basic relational algebra
- ▪ The tuple relational   calculus restricted to safe expressions.
- ▪ The domain relational calculus restricted to safe expressions.

OVERVIEW OF THE DESIGN PROCESS

What is Database Design?

- ❖ Database Design is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems.
- ❖ It helps produce database systems
    1. That meet the requirements of the users
    2. Have high performance.

✓ The main objectives of database designing are to produce logical and physical designs models of the proposed database system.

✓ **Logical Design Model:**
- ✓ The logical model concentrates on the data requirements and the data to be stored independent of physical considerations.
- ✓ It does not concern itself with how the data will be stored or where it will be stored physically.

❖ **Physical Design Model:**
- ✓ The physical data design model involves translating the logical design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

Why Database Design is Important ?
❖ Database designing is crucial to **high performance** database system.
❖ Apart from improving the performance, properly designed database are easy to maintain, improve data consistency and are cost effective in terms of disk storage space.

Database development life cycle
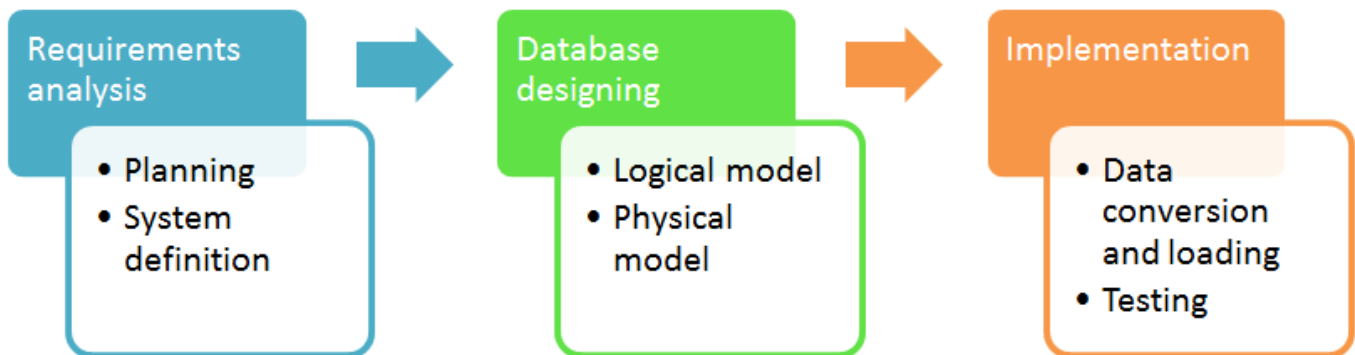
Requirements analysis
- **Planning** - This stages concerns with planning of entire Database Development Life Cycle. It takes into consideration the Information Systems strategy of the organization.
- **System definition** - This stage defines the scope and boundaries of the proposed database system.
  Database designing
- **Logical model** - This stage is concerned with developing a database model based on requirements. The entire design is on paper without any physical implementations or specific DBMS considerations.
- **Physical model** - This stage implements the logical model of the database taking into account the DBMS and physical implementation factors.
  Implementation
- **Data conversion and loading** - this stage is concerned with importing and converting data from the old system into the new database.
- **Testing** - this stage is concerned with the identification of errors in the newly implemented system .It checks the database against requirement specifications.

Two Types of Database Techniques
1. **Normalization**
2. **ER Modeling**

ENTITY- RELATIONSHIP MODEL

1. Explain about the Basic Concepts of E-R Model. (Part-C)
   List out and explain about the Different Types of Attributes. (Part-B)

Introduction:
   - ✓ The Entity-Relationship (E-R) data model is based on perception of a real world that consists of a set of basic objects called entities, and of relationships among these objects.
   - ✓ There are three basic notation used in E-R data model. They are:
        - ▪ **Entity Sets**
        - ▪ **Attributes**
        - ▪ **Relationship Sets**

Entity Sets:
   - ✓ An entity is collection of object. The set of entities of the same type that share the same properties or attributes is called as entity sets.
   - ✓ The entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
   - ✓ It is represented by a set of attributes. The attributes are descriptive properties possessed by each member of an entity set.
   - ✓ An attribute of an entity set is a function that maps from the entity set into a domain.
   - ✓ Each entity set have several attributes, each entity can be described by a set of (attribute, data value) pairs, one pair for each attribute of the entity set.
   - ✓ A database includes a collection of entity sets each of which contains any number of entities of the same type.

   Example:
   - A bank database which consists of two entity sets: customer and loan
   - The customer entity may be described by the set **{(name, smith), (Customer-street, Main Road)}**. The set of all persons who are customers at a given bank can be defined as the entity set customer. The set of loans awarded by a particular bank can be defined as the entity set loan.

customer-id	customer-name	customer-street	customer-city	loan-number	amount
321-12-3123	Jones	Main	Harrison	L-17	1000
019-28-3746	Smith	North	Rye	L-23	2000
677-89-9011	Hayes	Main	Harrison	L-15	1500
555-55-5555	Jackson	Dupont	Woodside	L-14	1500
244-66-8800	Curry	North	Rye	L-19	500
963-96-3963	Williams	Nassau	Princeton	L-11	900
335-57-7991	Adams	Spring	Pittsfield	L-16	1300
customer				*loan*	

Fig: Entity Sets Customer and Loans

Attributes:
- ✓ An attribute used in E-R model, can be characterized by the following types:
    - **Simple and Composite Attributes**
    - **Single-valued and Multivalued Attributes**
    - **Null Attribute**
    - **Derived Attribute**

i. **Simple and Composite Attributes:**
- ✓ The attributes not divided into subparts is called as **simple** attributes. The attributes divided into subparts is called as **composite** attributes. It may appear as a hierarchy.

Example:

Entity Set:	**Customer**
Composite Attributes:	**Customer-name** **Customer-address**
Component Attributes:	**First-Name Middle-Name Last-Name Street City State**

Street-number   Street-name   Apt-number
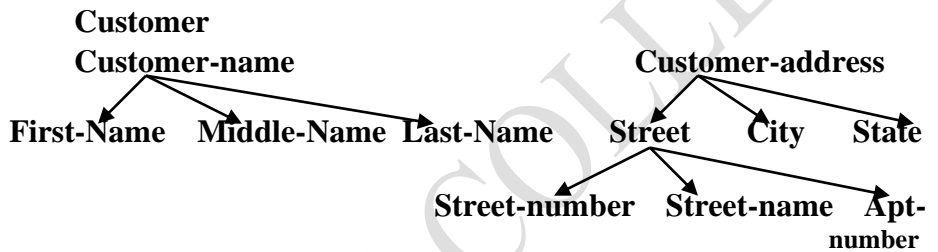
Figure: Composite attributes Customer-name and Customer-address

ii. **Single-valued and Multi-valued Attributes:**
- ✓ The attribute have single value for a particular entity is called as **single-valued** attributes.
    - **Example:** Rollno attribute for a specific student entity refers to only one value.
- ✓ The attributes have different numbers of values for a particular entity is called as **multi-valued** attributes.
    - **Example:** Phone no attribute for a specific employee entity refers more than one value.

iii. **Null Attributes:**
- ✓ An entity does not have a value for an attribute is called as **Null** attributes. It designate that an attribute value is unknown. An unknown value may be either missing or not known.
    - **Example:** Apt-number attribute for a specific customer entity refers unknown value.

iv. **Derived Attributes:**
- ✓ The value of attribute can be derived from the values of other related attributes or entities are called as **derived** attribute.
    - **Example:** The value of employment-length can be derived from the value of start-date and the current date.

Relationship Sets:
- ✓ A relationship is an association among several entities. A set of relationships of the same type is called as Relationship Set.
- ✓ If $E_1, E_2, ….E_n$ are entity sets, then a relationship set R is a subset of

    $\{(e_1, e_2, …, e_n) \mid e_1 \in E_1, e_2 \in E_2, …, e_n \in E_n\}$

    Where $(e_1, e_2,…., e_n)$ is a relationship
- ✓ The association between entity sets is referred to as participation; that is the entity sets $E_1, E_2,…,E_n$ participate in relationship set R.
- ✓ The entity sets participating in a relationship set are generally distinct roles are implicit and are not usually specified. When the entity sets of a relationship set are not distinct; that is the same entity set participates in a relationship set more than once, in different roles. This type of relationship set called as recursive relationship set.

✓ A relationship may also have descriptive attributes. Consider a relationship set depositor with entity sets customer and account.

✓ The attribute access-date associate with relation to specify the most recent date the customer accessed an account.

✓ The number of entity sets that participate in a relationship set is also called as **degree** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.
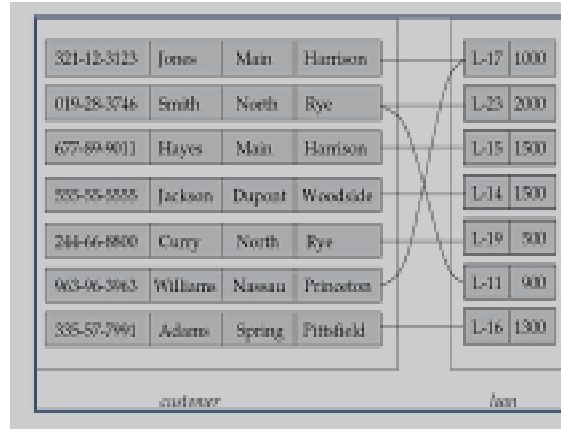
Figure: Relationship Set Borrower

REDUCTION TO RELATIONAL SCHEMAS

Reduction of an E-R Schema to Tables

❖ A database that conforms to an E-R database schema can be represented by a collection of tables.
❖ For each entity set and for each relationship set, there is a unique table.
❖ A table is a chart with rows and columns.
❖ The set of all possible rows is the Cartesian product of all columns.
❖ A row is also known as a *tuple* or a *record*.
❖ A table has an unlimited number of rows. Each column is also known as a *field*.

Strong Entity Sets

❖ It is common practice for the table to have the same name as the entity set. There is one column for each attribute.

Weak Entity Sets

❖ There is one column for each attribute, plus the attribute(s) the form the primary key of the strong entity set that the weak entity set depends upon.

Relationship Sets

❖ We represent a relationship with a table that includes the attributes of each of the primary keys plus any descriptive attributes (if any).
❖ There is a problem that if one of the entities in the relationship is a weak entity set, there would be no unique information in the relationship table, and therefore may be omitted.
❖ Another problem can occur if there is an existance dependency. In that case, you can combine the two tables.

Multivalued Attributes

❖ When an attribute is multivalued, remove the attribute from the table and create a new table with the primary key and the attribute, but each value will be a separate row.

Generalization

❖ Create a table for the higher-level entity set.
❖ For each lower-level entity set, create a table with the attributes for that specialization and include the primary key from the higher-level entity set.

ENTITY- RELATIONSHIP DESIGN ISSUES

1. Explain about the Basic Issues in the Design of an E-R Database Schema. (Part-B)
- ✓ The basic issues in the design of an E-R database schema:
    - **Use of Entity Sets versus Attributes**
    - **Use of Entity Sets versus Relationship Sets**
    - **Binary Versus n-ary Relationship Sets**
    - **Placement of Relationship Attributes**

Use of Entity Sets versus Attributes:
- ✓ Consider the entity set employee with attributes employee-name and telephone-number. It can be easily argued that a telephone is an entity in its own right with attributes telephone-number and location. The employee entity set must be redefined as follows:
    - **The employee entity set with attribute employee-name.**
    - **The telephone entity set with attributes telephone-number and location.**
    - **The relationship set emp-telephone, which denotes the association between employees and the telephones.**
- ✓ In the first definition implies that every employee has precisely one telephone number associated with him. In the second definition implies that the employees may have several telephone numbers associated with him. It the second one is more general than the first one, and may more accurately reflect the real-world situation. The distinctions mainly depend on the structure of the real-world enterprise being modeled, and on the semantics associated with the attribute.

Use of Entity Sets versus Relationship Sets:
- ✓ It is not always clear whether an object is best expressed by an entity set or a relationship set. The problems arise as a result of the replication
    - The data are stored multiple times, wasting storage space.
    - Updates potentially leave the data in an inconsistent state, where the values differ in two relationships for attributes that are supposed to have the same value.
- ✓ To avoid the replication by using normalization. One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach is useful in deciding whether certain attributes may be more appropriately expressed as relationships.

Binary Versus n-ary Relationship Sets:
- ✓ It is always possible to replace a non-binary (n-ary, for $n>2$) relationship set by a number of distinct binary relationship sets. Consider the abstract ternary (n=3) relationship set R, relating entity sets A, B, and C. We replace the relationship set R by an entity set E, and create three relationship sets:
    - R_A, relating E and A
    - R_B, relating E and B
    - R_C, relating E and C
- ✓ If the relationship set R had any attributes, these are assigned to entity set E; a special identifying attribute is created for E. For each relationship (a_i, bi, ci) in the relationship set R, create a new entity e_i in the entity set E. In each of the three new relationship sets, we insert a relationship as follows:
    - (e_i, a_i) in R_A
    - (e_i, b_i) in R_B
    - (e_i, c_i) in R_C
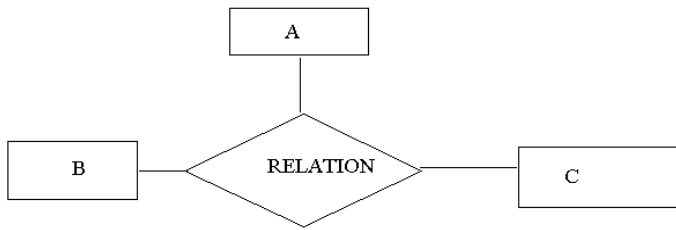
Fig-1 Ternary Relation

For simplicity, consider the abstract ternary (n = 3) relationship set R, relating entity sets A, B, and C (Fig-1). We replace the relationship set R by an entity set E, and create three relationship sets:
• $R1$, relating E and A
• $R2$, relating E and B
• $R3$, relating E and C
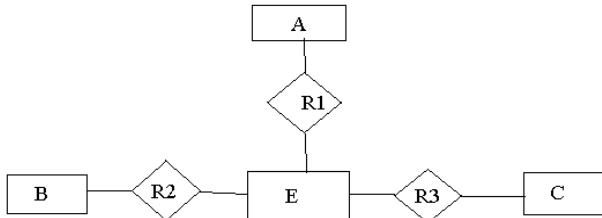and thus the Ternary relation is converted into binary relation

Fig-2 Binary Relation

Placement of Relationship Attributes:

✓ The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one, one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set.

Figure 2.6   *Access-date* as attribute of the *account* entity set.
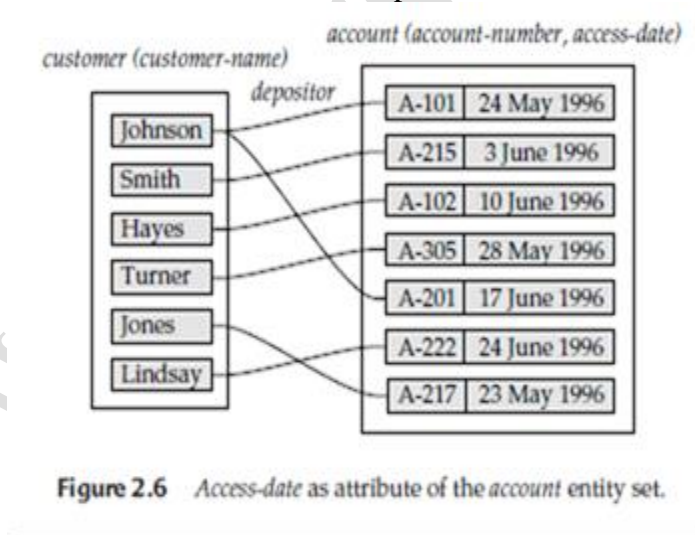
EXTENDED E-R FEATURES

1. Explain about the Extended Features of E-R Model. (Part-B)
   Write Short Note on Specialization and Generalization. (Part-B)
   ✓ The Entity-Relationship model consists of the following features:
       ▪ **Specialization**
       ▪ **Generalization**

Specialization:

✓ An entity set include sub groupings of entities that are distinct in some way from other entities in the set. A subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. It is depicted by a triangle component labeled **ISA**. The label **ISA** stands for **"is a"**.

✓ It is the result of taking a subset of a higher-level entity set to form a lower-level entity set.

59

Example:

- ✓ The entity set account with attributes account-number and balance. An account is further classified into the following accounts:
  - **Savings-account**
  - **Checking-account**
- ✓ The below diagram the savings-account entities are described further by the attribute interest-rate and checking-account entities are described further by the attribute overdraft-amount. The process of designating sub groupings within an entity set is called specialization.
- ✓ The specialization of checking-account by account type yields the following entity sets:
  - An **Standard** account, the bank keeps track of the number of checks written from an account each month. (Attribute:num-checks)
  - An **Gold** account the bank monitors the minimum balance and the interest paid for each month. (Attribute: min-balance and interest-payment)
  - An **Senior** account a record of the customer's date of birth is associated with this type of account. (Attribute: date-of-birth)
- ✓ The ISA relationship may also be referred to as a superclass-subclass relationship.
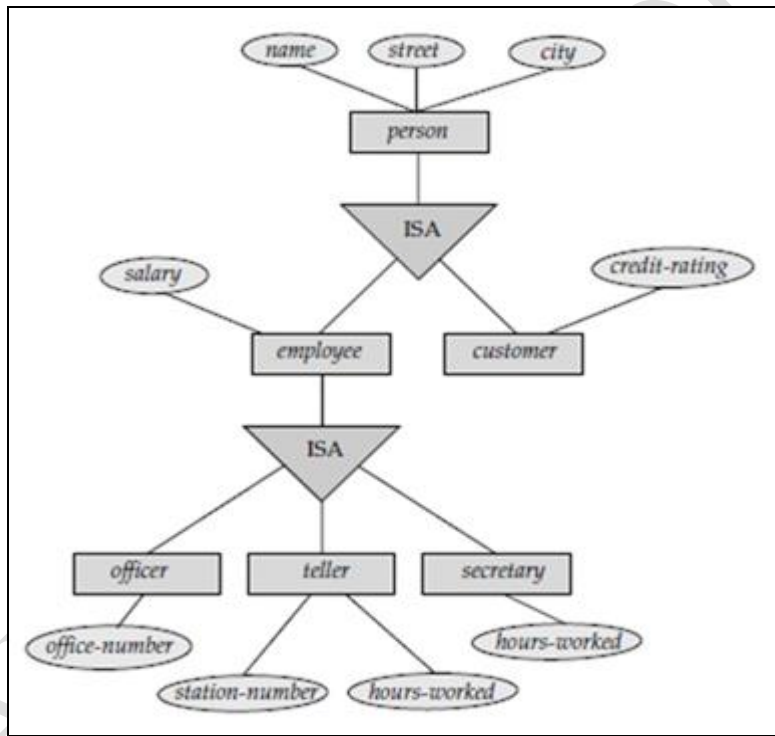
Figure: Specialization and Generalization

Generalization:

- ✓ The refinement from an initial entity set into successive levels of entity sub groupings represents a top-down design process in which distinctions are made explicit.
- ✓ The design process may also proceed in a bottom-up manner, in which multiple entity sets are synthesized into a higher-level entity set based on common features.
- ✓ The database designer may have first identified a checking-account entity set with the attributes account-number, balance, and overdraft-amount.
- ✓ It is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set.
- ✓ The attributes of higher-level entity sets are inherited by lower-level entity sets.
- ✓ It is a simple inversion of specialization. The higher-level entity set designated by the term superclass and lower-level entity set referred as subclass.

Attribute Inheritance:

- ✓ A crucial property of the higher-and lower-level entities created by specialization and generalization is attribute inheritance. The attributes of the higher-level entity sets are said to be inherited by lower-level entity set. For example, Customer and Employee inherit the attributes of person.
- ✓ A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates.
- ✓ Whether a given portion of an E-R model was arrived at by specialization or generalization the outcome is basically the same:
    - A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets.
    - Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set.
- ✓ In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship; that is, entity sets in this above diagram have only single inheritance.
- ✓ If an entity set is lower-level entity in more than one ISA relationship, then the entity set has multiple inheritance, and the resulting structure is said to be a **Lattice.**

Constraints on Generalization:

- ✓ To model an enterprise more accurately, the database designer may choose to place certain constraints on a particular generalization.
- ✓ One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following.
    - **Condition-defined:**
        - In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate.
        - **Example:**
        - The higher-level entity set account has the attribute account-type. All account entities are evaluated on the defining account-type attribute. All the lower-level entities are evaluated on the basis of the same attribute, this type of generalization is said to be attribute-defined.
    - **User-defined:**
        - User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set.
- ✓ A second type of constraints relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower-level entity sets may be one of the following:
    - **Disjoint:**
        - This constraint requires that an entity an entity belongs to no more than one lower-level entity set.
    - **Overlapping:**
        - In overlapping generalizations, the same entity may belong to more than one lower-level entity set within a single generalization.
- ✓ The completeness constraint on generalization or specialization specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization.
- ✓ This constraint may be one of the following:
    - **Total Generalization or Specialization:** Each higher-level entity must belong to a lower-level entity set.
    - **Partial Generalization or Specialization:** Some higher-level entities may not belong to any lower-level entity set.

Aggregation:

✓ It is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.

✓ It appears that the relationship sets works-on and manages can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some employee, branch, job combinations many not have a manager.
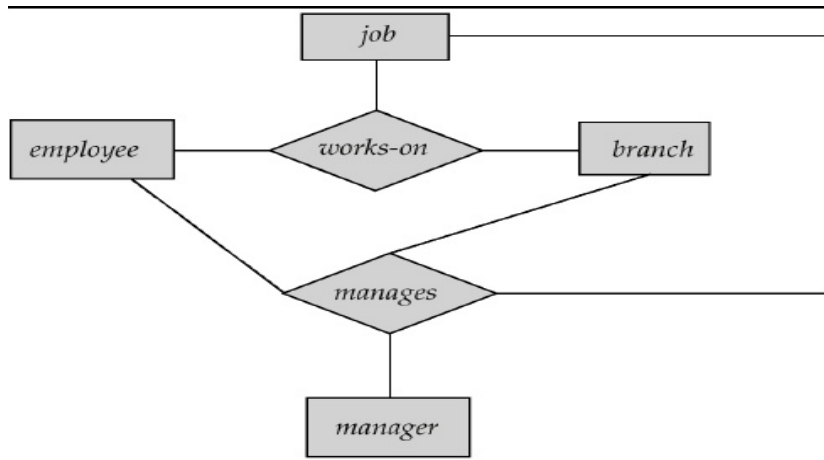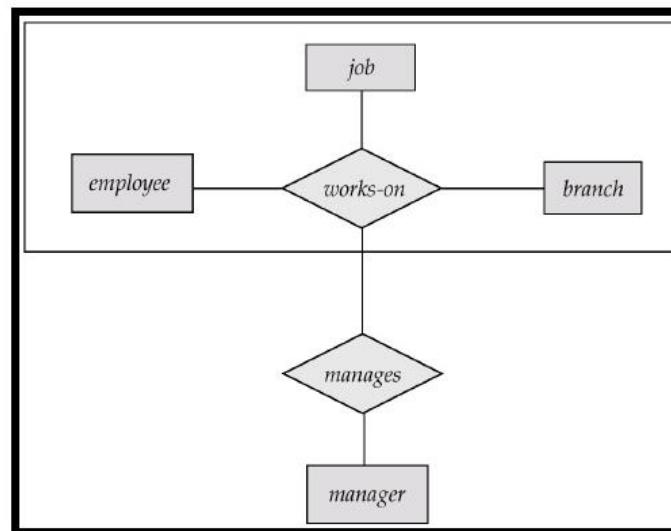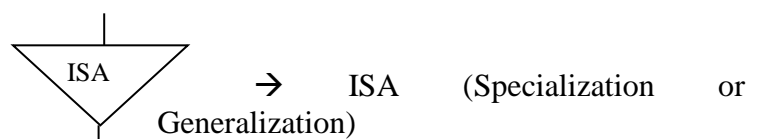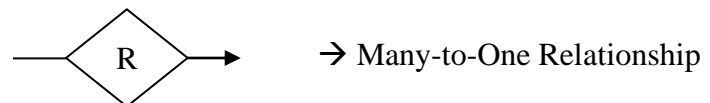

Figure: E-R diagram with redundant relationships

E-R Diagram With Aggregation

Alternatives E-R Notations:

E → Entity Set	R → Many-to-Many Relationship
E → Weak Entity Set	R → One-to-One Relationship
R → Relationship set	R → Many-to-One Relationship
R → Identifying Relationship Set for Weak Entity Set	ISA → ISA (Specialization or Generalization)

A → Attribute

A → Multivalued Attribute

A → Derived Attribute

A → Primary Key

ISA → Total Generalization

R ═══ E → Total Participation of Entity Set in Relationship

✓ The following alternative notations that are widely used. The primary key attributes are indicated by listing them at the top, with a line separating them from the other attributes.
✓ Cardinality constraints are indicated by different ways. The labels * and 1 on the edges out of the relationship are sometimes used for depicting many-to-one, many-to-many, one-to-one relationships.

E
A_1
A_2
A_3

Entity set E with Attributes A_1, A_2, A_3 and primary key A_1

*   R   *   1   R   1   *   R   1

Many-to-Many
Relationship

One-to-One
Relationship

Many-to-One
Relationship
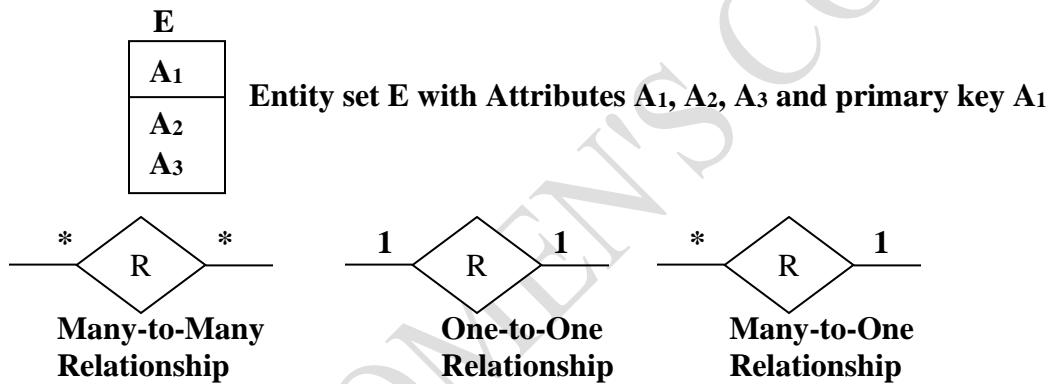
ALTERNATIVE NOTATION FOR MODELING DATA

❖ A diagrammatic representation of the data model of an application is a very important part of designing a database schema.
❖ Creation of a database schema requires not only data modeling experts, but also domain experts who know the requirements of the application but may not be familiar with data modeling.
❖ A number of alternative notations for modeling data have been proposed, of which E-R diagrams and UML class diagrams are the most widely used.

❖ Thereis no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations.

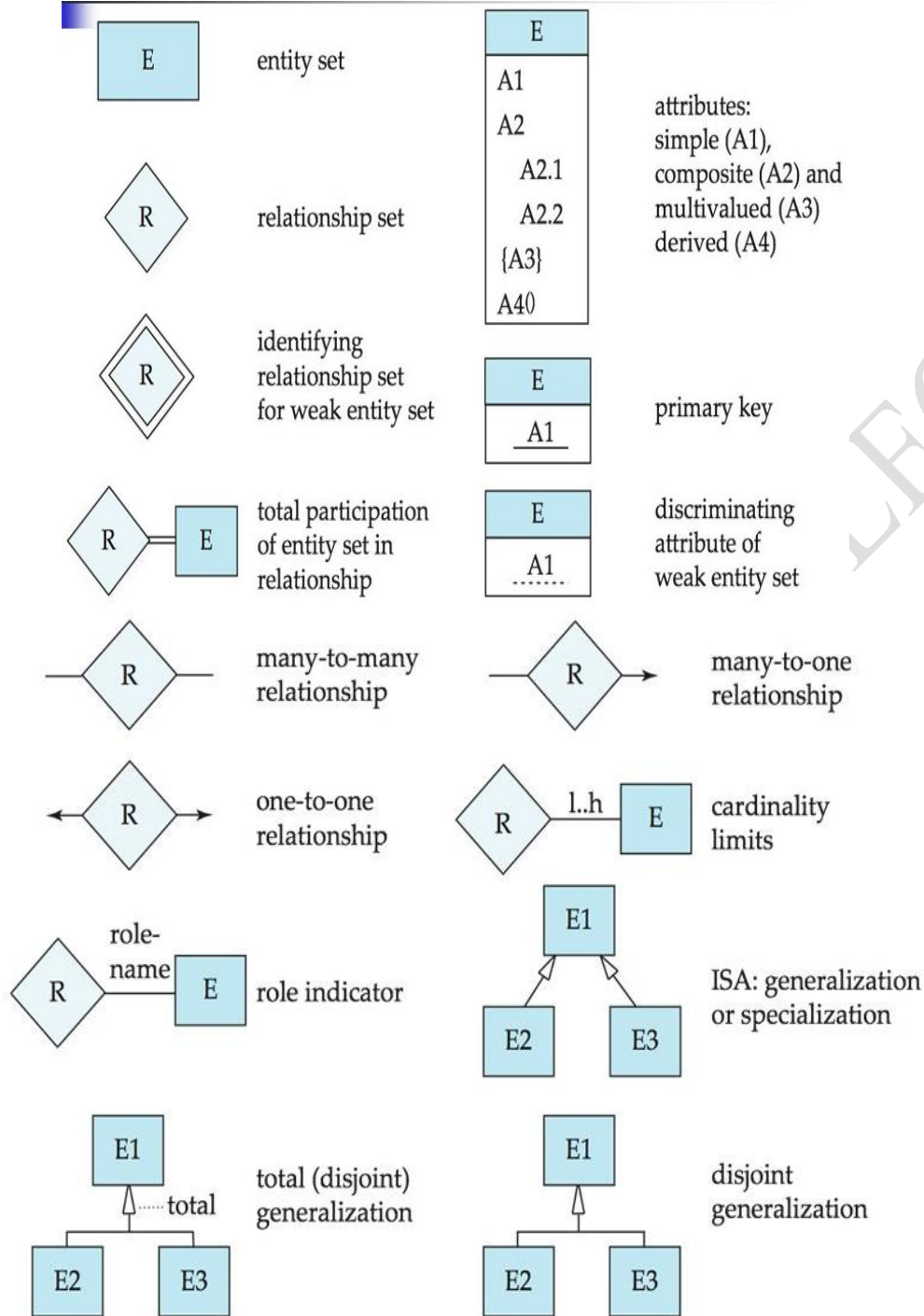❖ Figure 7.24 summarizes the set of symbols we have used in our E-R diagram notation.

Figure 7.24 Symbols used in the E-R notation.

a) **Alternative E-R Notations**
   ❖ The below diagram figure 7.25 indicates some of the alternative E-R notations that are widely used.
   ❖ One alternative representation of attributes of entities is to show them in ovals connected to the box representing the entity; primary key attributes are indicated by underlining them.
   ❖ The above notation is shown at the top of the figure. Relationship attributes can be similarly represented, by connecting the ovals to the diamond representing the relationship.
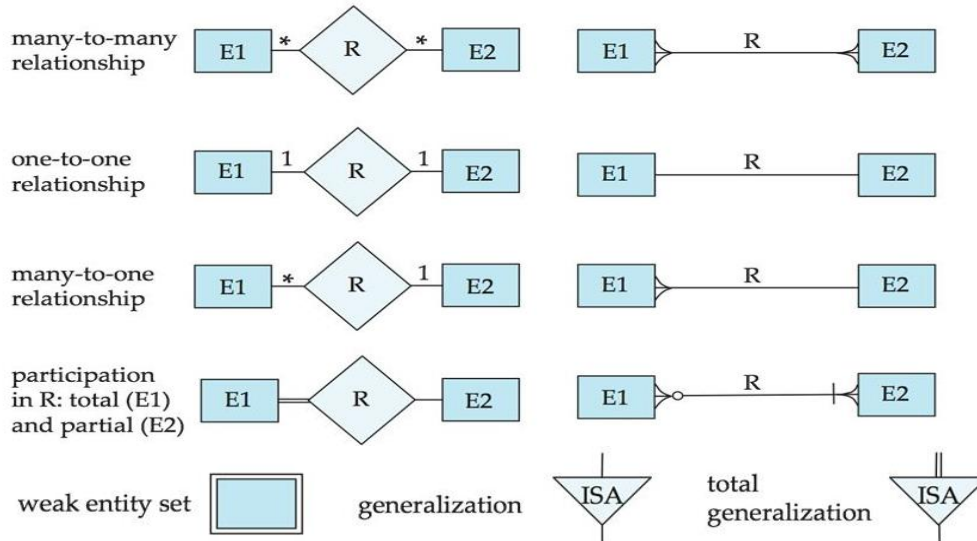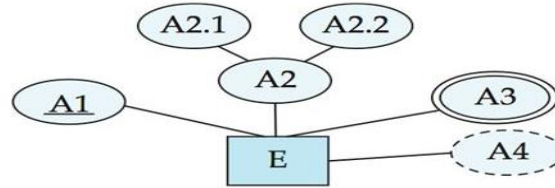
Figure 7.25 Alternative E-R notations.

b) The Unified Modeling Language UML
- ❖ Entity-relationship diagrams help model the data representation component of a software system.
- ❖ Data representation, however, forms only one part of an overall system design.
- ❖ Other components include models of user interactions with the system, specification of functional modules of the system and their interaction, etc.
- ❖ The **Unified Modeling Language** (UML) is a standard developed under the auspices of the Object Management Group (OMG) for creating specifications of various components of a software system.
- ❖ Some of the parts of UML are:
    - • **Class diagram**. A class diagram is similar to an E-R diagram. Later in this section we illustrate a few features of class diagrams and how they relate to E-R diagrams.
    - • **Use case diagram**. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
    - • **Activity diagram**.Activity diagrams depict the flow of tasks between various components of a system.
    - • **Implementation diagram**. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

OTHER ASPECTS OF DATABASE DESIGN

Data Constraints and Relational Database Design
- ❖ A variety of data constraints that can be expressed using SQL, including primary-key constraints, foreign-key constraints, **check** constraints, assertions, and triggers.
- ❖ Constraints serve several purposes.
- ❖ The most obvious one is the automation of consistency preservation.

❖ A further advantage of stating constraints explicitly is that certain constraints are particularly useful in designing relational database schemas.
❖ **For example**: That a social-security number uniquely identifies a person, then we can use a person's social-security number to link data related to that person even if these data appear in multiple relations.
❖ Contrast that with**, for example**, eye color, which is not a unique identifier. Eye color could not be used to link data pertaining to a specific person across relations because that person's data could not be distinguished from data pertaining to other people with the same eye color.

Usage Requirements: Queries, Performance

❖ Database system performance is a critical aspect of most enterprise information systems.
❖ Performance pertains not only to the efficient use of the computing and storage hardware being used, but also to the efficiency of people who interact with the system and of processes that depend upon database data.
❖ There are two main metrics for performance:

   • **Throughput**—the number of queries or updates (often referred to as *transactions*) that can be processed on average per unit of time.
   • **Response time**—the amount of time a *single* transaction takes from start to finish in either the average case or the worst case.

Authorization Requirements

❖ Authorization constraints affect design of the database as well because SQL allows access to be granted to users on the basis of components of the logical design of the database.
❖ A relation schema may need to be decomposed into two or more schemas to facilitate the granting of access rights in SQL.
❖ **For example,** an employee record may include data relating to payroll, job functions, and medical benefits.

Data Flow, Workflow

❖ Database applications are often part of a larger enterprise application that interacts not only with the database system but also with various specialized applications.
❖ **For example**, in a manufacturing company, a computer-aided design (CAD) system may assist in the design of new products.
❖ The term **workflow** refers to the combination of data and tasks involved in processes like those of the preceding examples.
❖ Workflows interact with the database system as they move among users and users perform their tasks on the workflow.

UNIT-IV COMPLETED

<div align="center">

UNIT-V

RELATIONAL DATABASE DESIGN
</div>

1. What is the Goal of Relational-Database Design? (Part-A)
- ✓ The goal of a relational-database design is to generate a set of relational schemas that allow us to store information without unnecessary redundancy and allows us to retrieve information easily. One approach is to design schemas that are in an appropriate normal form.

<div align="center">

FEATURES OF GOOD RELATIONAL DESIGNS
</div>

- ❖ To study precise ways of assessing the desirability of a collection of relation schemas.
- ❖ However, we can go a long way toward a good design using concepts we have already studied.
- ❖ For ease of reference, we repeat the schemas for the university database in Figure 8.1.

<div align="center">

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)

Figure 8.1 Schema for the university database.
</div>

Design Alternative: Larger Schemas
- ❖ Now, let us explore features of this relational database design as well as some alternatives. Suppose that instead of having the schemas *instructor* and *department*, we have the schema:

<div align="center">

inst_dept (ID, name, salary, dept name, building, budget)
</div>

- ❖ This represents the result of a natural join on the relations corresponding to *instructor* and *department*.
- ❖ This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design.
- ❖ Let us consider the instance of the *inst _dept* relation shown in Figure 8.2.
- ❖ Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department.
- ❖ For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt.

ID	name	salary	dept_name	building	budget
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci	Taylor	100000
98345	Kim	8000	Elec. Eng	Taylor	85000
76766	Cruck	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp.Sci	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

<div align="center">

Figure 8.2 The inst_dept table.

67
</div>

Design Alternative: Smaller Schemas

- ❖ A real-world database has a large number of schemas and an even larger number of attributes.
- ❖ The number of tuples can be in the millions or higher.
- ❖ Discovering repetition would be costly.
- ❖ In the case of **inst dept**, our process of creating an E-R design successfully avoided the creation of this schema. However, this fortuitous situation does not always occur.
- ❖ Therefore, we need to allow the database designer to specify rules such as "each specific value for *dept_name* corresponds to atmost one *budget*" even in cases where *dept_name* is not the primary key for the schema in question.
- ❖ In other words, we need to write a rule that says "if there were a schema (*dept_name*, *budget*), then *dept_name* is able to serve as the primary key." This rule is specified as a **functional dependency.**

<div align="center">

dept_name→budget

</div>

ATOMIC DOMAINS AND FIRST NORMAL FORM

- ❖ In the relational model, we formalize this idea that attributes do not have any substructure.
- ❖ A domain is atomic if elements of the domain are considered to be indivisible units.
- ❖ We say that a relation schema R is in first normal form (1NF) if the domains of all attributes of R are atomic. A set of names is an example of a non-atomic value.
- ❖ **For example:**
  - If the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.
- ❖ **Composite attributes**, such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip* also have non-atomic domains.
- ❖ It consider an organization that assigns employees identification numbers of the following form:
  - ➢ The first two letters specify the department and the remaining four digits are a unique number within the department for the employee.
  - ➢ **Examples** of such numbers would be "CS001" and "EE1127". Such identification numbers can be divided into smaller units, and are therefore non-atomic.
  - ➢ If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

DECOMPOSITION USING FUNCTIONAL DEPENDENCIES

- ❖ We noted that there is a formal methodology for evaluating whether a relational schema should be decomposed.
- ❖ This methodology is based upon the concepts of keys and functional dependencies.

a) Keys and Functional Dependencies

- ❖ A database models a set of entities and relationships in the real world.
- ❖ There are usually a variety of constraints (rules) on the data in the real world.
- ❖ **For example,** some of the constraints that are expected to hold in a university database are:
  - **1.** Students and instructors are uniquely identified by their ID.
  - **2.** Each student and instructor has only one name.
  - **3.** Each instructor and student is (primarily) associated with only one department.
  - **4.** Each department has only one value for its budget, and only one associated building.
- ❖ An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a legal instance of a database is one where all the relation instances are legal instances.
- ❖ Consider a relation schema $r(R)$, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

• Given an instance of *r* (*R*), we say that the instance **satisfies** the **functional dependency** α → β if for all pairs of tuples *t1* and *t2* in the instance such that *t1*[α] = *t2*[α], it is also the case that *t1*[β] = *t2*[β].

• We say that the functional dependency α → β **holds** on schema *r* (*R*) if, inevery legal instance of *r* (*R*) it satisfies the functional dependency.

❖ Functional dependencies allow us to express constraints that we cannot express with superkeys.
❖ In Section 8.1.2, we considered the schema:

inst_dept (ID, name, salary, dept_name, building, budget)

in which the functional dependency *dept_name* → *budget* holds because for each department (identified by *dept name*) there is a unique budget amount.
❖ We denote the fact that the pair of attributes (*ID*, *dept name*) forms a superkey for *inst dept* by writing:

ID, dept_name→name, salary, building, budget

b) Boyce-Codd Normal Form (BCNF)

❖ A table is in Boyce-Codd Normal Form (BCNF) if only determinants in the table are the candidate keys.
❖ A table is in Boyce-Codd Normal Form (BCNF) if every column, on which some other column is fully functionally dependent, is also a candidate for the primary key of the table.
❖ One of the more desirable forms is Boyce-Codd Normal Form (BCNF). A relation schema R is in BCNF with respect to a set of F of functional dependencies if for all functional dependencies in F⁺ of the form α→β, where α ⊆ R, and β ⊆ R, at least one of the following holds:
   ▪ α→β is a trivial functional dependencies (β ⊆ α)
   ▪ α is a superkey for schema R.

Example:

❖ Consider the school table consisting of three columns: Student, Subject and Teacher. One Student can study zero or more Subjects. For a given Student-Subject pair, there is always exactly one Teacher. The many teachers teaching the same Subject (to different students). The one Teacher can teach only one subject.

Student	Subject	Teacher
aaa	English	Meena
zzz	Hindi	Kalpana
xxx	English	Meena

❖ The Student and Subject, find out the Teacher who is teaching that subjects
   ▪ {Student, Subject} → Teacher
❖ The Student and a Teacher, find out the Subject that is being taught by the Teacher to the Student
   ▪ {Student, Teacher} → Subject
❖ The Teacher, find out the subject that the Teacher teaches
   ▪ Teacher→ Subject

Functional Dependency	Candidate Key
{Student, Subject} → Teacher	{Student, Subject}
{Student, Teacher} → Subject	{Student, Teacher}
Teacher→ Subject	None

c) BCNF and Dependency Preservation

❖ We have seen several ways in which to express database consistency constraints:
primary-key constraints, functional dependencies, **check** constraints, assertions, and triggers.
❖ Testing these constraints each time the database is updated can be costly and, therefore, it is useful to design the database in a way that constraints can be tested efficiently.

- ❖ One way to implement this change using the E-R design is by replacing the *advisor* relationship set with a ternary relationship set, *dept_advisor*, involving entity sets *instructor*, *student*, and *department* that is many-to-one from the pair *{student, instructor}* to *department*as shown inFigure8.6.
- ❖ The E-R diagram specifies the constraint that "a student may have more than one advisor, but at most one corresponding to a given department".
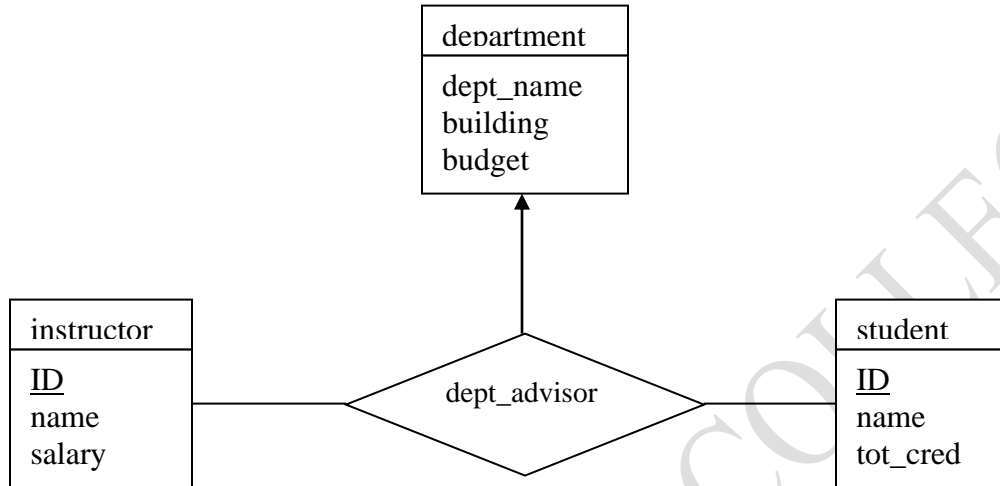

Figure 8.6 The dept advisor relationship set.

d) Third Normal Forms

- ❖ A table is in the Third Normal Form (3NF) if it is in the second normal form and if all non-key columns in the table depend non-transitively on the entire primary key.
- ❖ A relation schema R is in 3NF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$, and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is a trivial functional dependency.
  - α is a superkey for R.
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Example:

- ❖ Consider the following relation
  - Banker-info-schema = (branch-name, customer-name, banker-name, office-number)
- ❖ The functional dependency for this relation schema are
  - banker-name $\rightarrow$ branch-name office-number
  - customer-name branch-name $\rightarrow$ banker-name
- ❖ Decompose the Banker-info-schema by using the above functional dependencies.
  - Banker-office-schema = (banker-name, branch-name, office-number)
  - Banker-schema = (customer-name, branch-name, banker-name)

e) Higher Normal Forms

- ❖ Using functional dependencies to decompose schemas may not be sufficient to avoid unnecessary repetition of information in certain cases.
- ❖ Consider a slight variation in the *instructor* entity-set definition in which we record with each instructor a set of children's names and a set of phone numbers.
- ❖ The phone numbers may be shared by multiple people.
- ❖ Thus, *phone number* and *child name* would be multivalued attributes and, following our rules for generating schemas from an E-R design, we would have two schemas, one for each of the multivalued attributes, *phone_ number* and *child_name*:

    (ID, child_name)
    (ID, phone_number)

❖ If we were to combine these schemas to get

(ID, child_name, phone_number)

❖ We would find the result to be in BCNF because only nontrivial functional dependencies hold.
❖ As a result we might think that such a combination is a good idea.
❖ **For example**, let the instructor with *ID* 99999 have two children named "David"
   and "William" and two phone numbers, 512-555-1234 and 512-555-4321.
❖
❖ In the combined schema, we must repeat the phone numbers once for each dependent:

       (99999, David, 512-555-1234)
       (99999, David, 512-555-4321)
       (99999, William, 512-555-1234)
       (99999, William, 512-555-4321)

FUNCTIONAL DEPENDENCY THEORY

1. Explain about Functional Dependencies. (Part-B)
Discuss about Canonical Cover. (Part-B)

◈ The functional dependencies are a generalization of key dependencies. They require that the value for a certain set of attributes determines uniquely the value for another set of attributes. Using this determine the set of functional dependencies logically implied by a set of **F** of functional dependencies.

◈ It represents the constraints on the set of legal relations. The super key represents as follows:
   ▪ Let **R** be a relation schema. A subset of **k** of **R** is a superkey of **R** if, in any legal relation **r(R)**, for all pairs t_1 and t_2 of tuples in **r** such that $t_1 \neq t_2$, then $t_1[k] \neq t_2[k]$. That is no two tuples in any legal relation **r(R)** may have the same value on attribute set **k**.

◈ The notion of functional dependency generalizes the notion of super key. Let $\alpha \subseteq R$ and $\beta \subseteq R$. The functional dependency

        $\alpha \rightarrow \beta$

holds on **R** if, in any legal relation **r(R)**, for all pairs of tuples t_1 and t_2 in **r** such that $t_1[\alpha] \neq t_2[[\alpha]$.

◈ Using this notation **k** is a super key of **R** if $K \rightarrow R$. That is, **k** is super key if, whenever $t_1[k]=t_2[[k]$; it is also the case that $t_1[R]=t_2[[R]$ $(t_1=t_2)$
   ▪ On Branch-Schema
          branch-name $\rightarrow$ branch-city
          branch-name $\rightarrow$ assets
   ▪ On Customer-Schema
          customer-name $\rightarrow$ customer-city
          customer-name $\rightarrow$ customer-street

◈ To use the functional dependencies in two ways:
   ▪ To specify constraints on the set of legal relations. Consider the relation schema **R** that satisfies a set **F** of functional dependencies, say that **F** holds on **R**.
   ▪ If a relation **r** is legal under a set **F** of functional dependencies, say that **r** satisfies **F**.

◈ The functional dependencies are said to be trivial because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute **A**. The functional dependency of the form $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

Closure of a Set of Functional Dependencies:

◈ To determine the set of functional dependencies logically implied by a set of **F** or functional dependencies. This set is called the closure of **F**.

◈ The given relation schema R=(A,B,C,G,H,I) and the set of functional dependencies are:
   ▪ $A \rightarrow B$
   ▪ $A \rightarrow C$

- CG→H
- CG→I
- B→I

◈ The following functional dependency is logically implied.
- A→H

◈ Let **F** be a set of functional dependencies. The closure of **F** is the set of all functional dependencies logically implied by **F**. The closure of **F** is denoted by **F⁺**. Given **F**, compute **F⁺** directly from the formal definition of functional dependency. If **F** were large, this process would be lengthy and difficult.

◈ It consists of **various techniques**. The first technique is based on three axioms or rules of inference for functional dependencies. By applying these rules repeatedly find all **F⁺** in given **F**.

◈ The Greek letters **(α,β,γ,…)** for set of attributes, and uppercase Roman letters from the beginning of the alphabet for individual attributes. Use αβ to denote α∪β**.**
- **Reflexivity Rule.** If α is a set of attributes and β⊆α**,** then α→β holds.
- **Augmentation Rule.** If α→β holds and γ is a set of attributes, then γα→γβ holds.
- **Transitivity Rule.** If α→β holds β→γ holds, then α→γ holds.

◈ The above rules are sound and complete because they do not generate any incorrect functional dependencies and for a given set of F of functional dependencies, they allow us to generate all **F⁺**. This collection of rules is called Armstrong's axioms. To simplify further use the following additional rules:
- **Union Rule.** If α→β and α→γ holds, then α→βγ holds.
- **Decomposition Rule.** If α→βγ holds, then α→β holds and α→γ holds.
- **Pseudotransitivity Rule.** If α→β holds and γβ→δ holds, then αγ→δ holds.

Example:

◈ Consider the schema R=(A,B,C,G,H,I) and the set F of functional dependencies {A→B, A→C, CG→H, CG→I, B→H}. The list of F⁺ are:
- A→H (Since A→B and B→H holds)
  (Apply Transitivity Rule)
- CG→HI (Since CG→H and CG→I)
  (Apply Union Rule)
- AG→I (Since A→C and CG→I)
  (Apply Pseudotransitivity Rule)

Closure of Attribute Sets:

◈ Let α be a set of attributes. The set of all attributes functionally determined by α under a set **F** of functional dependencies the closure of α under **F**, denoted by α⁺. The input is a set **F** of functional dependencies and the set α of attributes. The output is stored in the variable result.

```
result:= α
while(changes to result) do
    for each functional dependency β→γ in F do
        begin
            if β⊆result then result:=result ∪ γ;
        end
```

Algorithm to compute α⁺ the closure of α under F

Canonical Cover:

◈ To reduce the size of a set F of functional dependency without changing the closure. This set is called a canonical cover **Fc** for F. Any database satisfies the simplified set of functional dependencies will also satisfy the original set, and vice versa, since the two sets have the same closure. The simplified set is easier to test.

◈ An attribute of a functional dependency is extraneous if we can remove it without changing the closure of the set of functional dependencies. The extraneous attributes defined as follows. Consider the set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in **F**.

- Attribute A is extraneous in α if **A$\epsilon\alpha$,** and **F** logically implies **(F-{$\alpha \rightarrow \beta$})U {α-A)$\rightarrow$ β}.**
- Attribute A is extraneous in β if **A$\epsilon\beta$**, and set of functional dependencies **(F-{$\alpha \rightarrow \beta$}) U {α-(β-A)}** logically implies F.

◈ A canonical cover F_c for F is a set of of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. The F_c has the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique.

◈ A canonical cover for a set of functional dependencies F can be computed as follows:

     repeat

      use the union rule to replace any dependencies in F of the form

         $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with, then $\alpha_1 \rightarrow \beta_1\beta_2$

      find a functional dependency $\alpha \rightarrow \beta$ with an extraneous attribute either in α or in β

      if an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

     until F does not change

Figure 8.9 Computing canonical cover.

Example:

◈ Consider the set F of functional dependencies on schema (A,B,C)

- A$\rightarrow$BC
- B$\rightarrow$C
- A$\rightarrow$B
- AB$\rightarrow$C

◈ There are two functional dependencies with the same set of attributes on the left side of the arrow:

- A$\rightarrow$BC
- A$\rightarrow$B

Combine these functional dependencies into A$\rightarrow$BC.

◈ A is extraneous in AB$\rightarrow$C because F logically implies (F-{AB$\rightarrow$C})U{B$\rightarrow$C}.

◈ C is extraneous in A$\rightarrow$BC, since A$\rightarrow$BC is logically implied by A$\rightarrow$B and B$\rightarrow$C

- The canonical cover is

     A$\rightarrow$B

     B$\rightarrow$C

Decomposition

❖ The decomposition refers to the breaking down of one table into multiple tables. Any database design process involves decomposition.

❖ Using this decomposition to minimize the data redundancy. The data redundancy not only leads to duplication of data it also has other side effects such as loss of data integrity and data consistency.

❖ It consists of the following types:

- **Lossy Decomposition**
- **Lossless Decomposition or Non-Lossy Decomposition**

    Lossy Decomposition: The loss of information due to decomposition is called lossy decomposition.

    Lossless Decomposition: When all information found in the original database is preserved after decomposition, call it lossless decomposition or non-lossy decomposition.

Lossless-Join Decomposition:
- ☉ Let R be a relation schema, and F be a set of functional dependencies on R. Let R_1 and R_2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in F^+:
  - ▪ $R_1 \cap R_2 \rightarrow R_1$
  - ▪ $R_1 \cap R_2 \rightarrow R_2$
- ☉ The decomposition of Lending-schema is a lossless-join decomposition by showing a sequence of steps that generate the decomposition.

Dependency Preservations:
- ☉ It is another goal of relational-database design. When an update is made to the database, the system should be able to check that the update will not create an illegal relation that is one that does not satisfy all the given functional dependencies.
- ☉ If the user checks the updates efficiently, we should design relational-database schemas that allow update validation without the computation of joins.
- ☉ Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. The restriction of F to R_i is the set F_i of all functional dependencies in F^+ that include only attributes of R_i. All functional dependencies in a restriction involve attributes of only one relation schemas it is possible to test satisfaction of such a dependency by checking only one relation.
- ☉ The decomposition of Lending-Schema is dependency preserving. Each member of the set F of functional dependencies that hold on Lending-Schema, and show that each one can be tested in at least one relation in the decomposition.
  - ▪ Test the functional dependency:
    branch-name$\rightarrow$ branch-city assets
    using Branch-Schema=(branch-name, branch-city,assets)

ALGORITHMS FOR DECOMPOSITION

BCNF Decomposition:
- ❖ The definition of BCNF can be used directly to test if a relation is in BCNF.
- ❖ However, computation of $F+$ can be a tedious task.
- ❖ We first describe below simplified tests for verifying if a relation is in BCNF.
- a) **Testing for BCNF**
  - ➢ Testing of a relation schema *R* to see if it satisfies BCNF can be simplified in some cases:
    • To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute $\alpha+$ (the attribute closure of α), and verify that it includes all attributes of *R*; that is, it is a super key of *R*.
    • To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than check all dependencies in *F+*.
  - ➢ We can show that if none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in *F+* will cause a violation of BCNF, either.
  - ➢ An alternative BCNF test is sometimes easier than computing every dependency in *F+*.
  - ➢ To check if a relation *Ri* in a decomposition of *R* is in BCNF, we apply this test:
    • For every subset α of attributes in *Ri*, check that $\alpha+$ (the attribute closure of α under *F*) either includes no attribute of *Ri* − α, or includes all attributes of *Ri* .

```
result := {R};
done := false;
compute F+;
while (not done) do
if (there is a schema Ri in result that is not in BCNF)
then begin
```

> **let** α→β **be a nontrivial functional dependency that holds**
> **on Ri such that** α→ **Ri is not in F+, and** α∩ β = ∅;
> **result := (result − Ri) ∪ (Ri − β) ∪ (α, β);**
> **end**
> **else done := true;**

Figure 8.11 BCNF decomposition algorithm.

➢ If the condition is violated by some set of attributes α in *Ri* , consider the following functional dependency, which can be shown to be present in *F*+:

α → (α+ − α) ∩ *Ri* .

➢ The above dependency shows that *Ri* violates BCNF.

b) BCNF Decomposition Algorithm

➢ We are now able to state a general method to decompose a relation schema so as to satisfy BCNF. Figure 8.11 shows an algorithm for this task.

➢ If *R* is not in BCNF, we can decompose *R* into a collection of BCNF schemas *R1, R2, . . . , Rn* by

the algorithm.

➢ The algorithm uses dependencies that demonstrate violation of BCNF to perform the decomposition.

➢ The decomposition that the algorithm generates is not only in BCNF, but is also a lossless decomposition. To see why our algorithm generates only lossless decompositions, we note that, when we replace a schema *Ri* with (*Ri* − β) and (α,β), the dependency α→β holds, and (*Ri* − β) ∩ (α,β)=α.

➢ As a longer example of the use of the BCNF decomposition algorithm, suppose we have a database design using the *class* schema below:

class (course id, title, dept name, credits, sec id, semester, year, building,
room number, capacity, time slot id)

➢ The set of functional dependencies that we require to hold on *class* are:

course id→ title, dept name, credits
building, room number→capacity
course id, sec id, semester, year→building, room number, time slot id

3NF Decomposition

❖ The set of dependencies *Fc* used in the algorithm is a canonical cover for *F*.

❖ Note that the algorithm considers the set of schemas *Rj , j = 1, 2, . . . , i*; initially *i = 0*, and in this case the set is empty.

❖ Let us apply this algorithm to our example of Section 8.3.4, where we showed that:

dept advisor (s_ID, i ID, dept _name)

is in 3NF even though it is not in BCNF. The algorithm uses the following functional dependencies in *F*:

f1: i_ID→dept _name
f2: s _ID, dept_ name→i_ID

Correctness of the 3NF Algorithm

❖ The 3NF algorithm ensures the preservation of dependencies by explicitly building a schema for each dependency in a canonical cover.

❖ It ensures that the decomposition is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed.

❖ This algorithm is also called the **3NF synthesis algorithm**, since it takes a set of dependencies and adds one schema at a time, instead of decomposing the initial schema repeatedly

Comparison of BCNF and 3NF:

- ◈ The 3NF and BCNF are the two normal forms for relational-database schemas. There is an advantage to 3NF: it is always possible to obtain a 3NF design without sacrificing a lossless join or dependency preservation. The disadvantage to 3NF is that there is a possibility to use null values to represent some of the possible meaningful relationships among data items, and there is a problem of repetition of information. The three design goals for a relational-database design are:
    - ▪ BCNF
    - ▪ Lossless join
    - ▪ Dependency Preservation
- ◈ If all the above three cannot be achieved, then accept
    - ▪ 3NF
    - ▪ Lossless join
    - ▪ Dependency Preservation

DECOMPOSITION USING MULTIVALUED DEPENDENCIES

- ❖ Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information.
- ❖ Consider a variation of the university organization where an instructor may be associated with multiple departments.

    inst (***ID***, *dept_name*, *name*, *street*, *city*)

- ❖ The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

    ID*→ *name*, *street*, *city

    and because *ID* is not a key for *inst*.

- ❖ Following the BCNF decomposition algorithm, we obtain two schemas:

    r1 (***ID***, *name*)
    r2 (***ID***, *dept_name*, *street*, *city*)

Multivalued Dependencies:

- ❖ The new form of constraint is multivalued dependency. It is used to define a normal form for relation schemas. This normal form called Fourth Normal Form (4NF) is more restrictive than BCNF.
- ❖ It do not rule out the existence of certain tuples. They require that other tuples of a certain form be present in the relation. The functional dependencies sometimes are referred to as equality-generating dependencies and multivalued dependencies are referred to as tuple-generating dependencies.
- ❖ Let R be a relation schema and let $\alpha \subseteq R$, and $\beta \subseteq R$. The multivalued dependency

    - $\alpha \twoheadrightarrow \beta$

    holds on R if , in any legal relation r(R) , for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha]=t_2[\alpha]$, there exists tuples t_3 and t_4 in r such that

    $$t_1[\alpha]=t_2[\alpha]=t_3[\alpha]=t_4[\alpha]$$
    $$t_3[\beta]=t_1\beta]$$
    $$t_3[R- \beta]=t_2[R- \beta]$$
    $$t_4[\beta]=t_2[\beta]$$

Fourth Normal Form

- ❖ A table is in the Fourth Normal Form (4NF) if it is in BCNF and does not have any independent multi-valued parts of the primary key.
- ❖ This normal form is related to the concept of a Multivalued Dependency (MVD). A relation schema R is in 4NF with respect to a set of D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$, and $\beta \subseteq R$, at least one of the following holds:
  - ▪ $\alpha \twoheadrightarrow \beta$ is a trivial functional dependency.
  - ▪ α is a superkey for schema R.
- ❖ Let R be a relation schema, and let D be a set of functional and multivalued dependencies on R. Let R_1 and R_2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if and only if at least one of the following multivalued dependencies is in D^+.
  - ▪ $R_1 \cap R_2 \twoheadrightarrow R_1$
  - ▪ $R_1 \cap R_2 \twoheadrightarrow R_2$

4NF Decomposition

- ❖ The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema into 4NF. Figure 8.16 shows the 4NF decomposition algorithm.

  result := {R};
  done := false;
  compute D+; Given schema R_i, let Di denote the restriction of D+ to R_i

  while (**not** done) **do**
  if (there is a schema R_i in result that is not in 4NF w.r.t. D_i)

  then begin
  let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued dependency that holds
  on R_i such that $\alpha \to R_i$ is not in Di, and $\alpha \cap \beta = \emptyset$;
  result := (result − R_i) ∪ (R_i − β) ∪ (α,β);
  end
  else done := true;
  Figure 8.16 4NF decomposition algorithm.

- ❖ It is identical to the BCNF decomposition algorithm of Figure 8.11, except that it uses multivalued dependencies and uses the restriction of *D+* to *Ri* .
- ❖ If we apply the algorithm of Figure 8.16 to (*ID*, *dept_ name*, *street*, *city*), we find that *ID→→dept_name* is a nontrivial multivalued dependency, and *ID* is not a superkey for the schema.
- ❖ Following the algorithm, we replace it by two schemas:

  ***r*21 (*ID*, *dept_name*)**
  ***r*22 (*ID*, *street*, *city*)**

MORE NORMAL FORMS

- ❖ The fourth normal form is by no means the "ultimate" normal form.
- ❖ As we saw earlier, multivalued dependencies help us understand and eliminate some forms of repetition of information that cannot be understood in terms of functional dependencies.
- ❖ There are types of constraints called **join dependencies** that generalize multivalued dependencies, and lead to another normal form called **project-join normal form (PJNF)** (PJNF is called **fifth normal form** in some books).
- ❖ There is a class of even more general constraints that leads to a normal form called **domain-key normal form (DKNF)**.

DATABASE-DESIGN PROCESS

- ❖ So far we have looked at detailed issues about normal forms and normalization.
- ❖ In this section, we study how normalization fits into the overall database-design process.
- ❖ There are several ways in which we could have come up with the schema *r* (*R*):

> **1.** *r* (*R*) could have been generated in converting an E-R diagram to a set of relation schemas.
>
> **2.** *r* (*R*) could have been a single relation schema containing *all* attributes that are of interest. The normalization process then breaks up *r* (*R*) into smaller schemas.
>
> **3.** *r* (*R*) could have been the result of an ad-hoc design of relations that we then test to verify that it satisfies a desired normal form.

E-R Model and Normalization

- ❖ When we define an E-R diagram carefully, identifying all entities correctly, the relation schemas generated from the E-R diagram should not need much further normalization
- ❖ For instance, suppose an *instructor* entity set had attributes *dept_name* and *dept_ address*, and there is a functional dependency *dept_name → dept_ address*.
- ❖ We would then need to normalize the relation generated from *instructor*.
- ❖ Functional dependencies can help us detect poor E-R design.
- ❖ If the generated relation schemas are not in desired normal form, the problem can be fixed in the ERdiagram.
- ❖ That is, normalization can be done formally as part of data modeling.
- ❖ Alternatively, normalization can be left to the designer's intuition during E-R modeling, and can be done formally on the relation schemas generated from the E-R model.
- ❖ If a multivalued dependency holds and is not implied by the corresponding functional dependency, it usually arises from one of the following sources:

> • A many-to-many relationship set.
>
> • A multivalued attribute of an entity set.

Naming of Attributes and Relationships

- ❖ A desirable feature of a database design is the **unique-role assumption**, which means that each attribute name has a unique meaning in the database.
- ❖ This prevents us from using the same attribute to mean different things in different schemas.
- ❖ For example, we might otherwise consider using the attribute *number* for phone number in the *instructor* schema and for room number in the *classroom* schema.
- ❖ Although technically, the order of attribute names in a schema does not matter, it is convention to list primary-key attributes first.
- ❖ This makes reading default output (as from **select ***) easier.

Denormalization for Performance

- ❖ Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications.
- ❖ The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.
- ❖ For instance, suppose all course prerequisites have to be displayed along with a course information, every time a course is accessed.
- ❖ In our normalized schema, this requires a join of *course* with *prereq*.
- ❖ One alternative to computing the join on the fly is to store a relation containing all the attributes of *course* and *prereq*.
- ❖ This makes displaying the "full" course information faster.
- ❖ The process of taking a normalized schema and making it nonnormalized is called **denormalization**,

Other Design Issues

- ❖ There are some aspects of database design that are not addressed by normalization, and can thus lead to bad database design. Data pertaining to time or to ranges of time have several such issues.
- ❖ We give examples here; obviously, such designs should be avoided.
- ❖ Consider a university database, where we want to store the total number of instructors in each department in different years.
- ❖ A relation *total_ inst*(*dept_ name*, *year*, *size*) could be used to store the desired information.
- ❖ The only functional dependency on this relation is *dept name*, *year→size*, and the relation is in BCNF.
- ❖ An alternative design is to use multiple relations, each storing the size information for a different year.
- ❖ Let us say the years of interest are 2007, 2008, and 2009; we would then have relations of the form *total _inst 2007*, *total_inst 2008*, *total_inst 2009*, all of which are on the schema (*dept _name*, *size*). The only functional dependency here on each relation would be *dept_name → size*, so these relations are also in BCNF.

UNIT-V COMPLETED