Title: Database Management System

Authors Name: Er. Karan Kumar

DATABASE MANAGEMENT SYSTEM (KCS501)

Course Outcome (CO)	**Bloom's Knowledge Level (KL)**
At the end of course , the student will be able to:	
CO 1 Apply knowledge of database for real life applications.	K3
CO 2 Apply query processing techniques to automate the real time problems of databases.	K3, K4
CO 3 Identify and solve the redundancy problem in database tables using normalization.	K2, K3
CO 4 Understand the concepts of transactions, their processing so they will familiar with broad range of database management issues including data integrity, security and recovery.	K2, K4
CO 5 Design, develop and implement a small database project using database tools.	K3, K6

Syllabus

Unit-I Introduction: Overview, Database System vs File System, Database System Concept and Architecture, Data Model Schema and Instances, Data Independence and Database Language and Interfaces, Data Definitions Language, DML, Overall Database Structure. Data Modeling Using the Entity Relationship Model: ER Model Concepts, Notation for ER Diagram, Mapping Constraints, Keys, Concepts of Super Key, Candidate Key, Primary Key, Generalization, Aggregation, Reduction of an ER Diagrams to Tables, Extended ER Model, Relationship of Higher Degree.

Unit-II Relational data Model and Language: Relational Data Model Concepts, Integrity Constraints, Entity Integrity, Referential Integrity, Keys Constraints, Domain Constraints, Relational Algebra, Relational Calculus, Tuple and Domain Calculus. Introduction on SQL: Characteristics of SQL, Advantage of SQL. SQl Data Type and Literals. Types of SQL Commands. SQL Operators and Their Procedure. Tables, Views and Indexes. Queries and Sub Queries. Aggregate Functions. Insert, Update and Delete Operations, Joins, Unions, Intersection, Minus, Cursors, Triggers, Procedures in SQL/PL SQL

Unit-III Data Base Design & Normalization: Functional dependencies, normal forms, first, second, third normal forms, BCNF, inclusion dependence, loss less join decompositions, normalization using FD, MVD, and JDs, alternative approaches to database design

Unit-IV Transaction Processing Concept: Transaction System, Testing of Serializability, Serializability of Schedules, Conflict & View Serializable Schedule, Recoverability, Recovery from Transaction Failures, Log Based Recovery, Checkpoints, Deadlock Handling. Distributed Database: Distributed Data Storage, Concurrency Control, Directory System.

Unit-V Concurrency Control Techniques: Concurrency Control, Locking Techniques for Concurrency Control, Time Stamping Protocols for Concurrency Control, Validation Based Protocol, Multiple Granularity, Multi Version Schemes, Recovery with Concurrent Transaction, Case Study of Oracle.

Index:

UNIT 1

INTRODUCTION

1.1. Overview

Database is a collection of related data and data is a collection of facts and figures that can be processed to produce information.

Mostly data represents recordable facts. Data aids in producing information, which is based on facts. For example, if we have data about marks obtained by all students, we can then conclude about toppers and average marks.

A **database management system** stores data in such a way that it becomes easier to retrieve, manipulate, and produce information.

1.1.1. Characteristics

Traditionally, data was organized in file formats. DBMS was a new concept then, and all the research was done to make it overcome the deficiencies in traditional style of data management. A modern DBMS has the following characteristics −

- **Real-world entity** − A modern DBMS is more realistic and uses real-world entities to design its architecture. It uses the behavior and attributes too. For example, a school database may use students as an entity and their age as an attribute.
- **Relation-based tables** − DBMS allows entities and relations among them to form tables. A user can understand the architecture of a database just by looking at the table names.
- **Isolation of data and application** − A database system is entirely different than its data. A database is an active entity, whereas data is said to be passive, on which the database works and organizes. DBMS also stores metadata, which is data about data, to ease its own process.
- **Less redundancy** − DBMS follows the rules of normalization, which splits a relation when any of its attributes is having redundancy in values. Normalization is a mathematically rich and scientific process that reduces data redundancy.
- **Consistency** − Consistency is a state where every relation in a database remains consistent. There exist methods and techniques, which can detect attempt of leaving database in

inconsistent state. A DBMS can provide greater consistency as compared to earlier forms of data storing applications like file-processing systems.

- **Query Language** − DBMS is equipped with query language, which makes it more efficient to retrieve and manipulate data. A user can apply as many and as different filtering options as required to retrieve a set of data. Traditionally it was not possible where file-processing system was used.
- **ACID Properties** − DBMS follows the concepts of **A**tomicity, **C**onsistency, **I**solation, and **D**urability (normally shortened as ACID). These concepts are applied on transactions, which manipulate data in a database. ACID properties help the database stay healthy in multi-transactional environments and in case of failure.
- **Multiuser and Concurrent Access** − DBMS supports multi-user environment and allows them to access and manipulate data in parallel. Though there are restrictions on transactions when users attempt to handle the same data item, but users are always unaware of them.
- **Multiple views** − DBMS offers multiple views for different users. A user who is in the Sales department will have a different view of database than a person working in the Production department. This feature enables the users to have a concentrate view of the database according to their requirements.
- **Security** − Features like multiple views offer security to some extent where users are unable to access data of other users and departments. DBMS offers methods to impose constraints while entering data into the database and retrieving the same at a later stage. DBMS offers many different levels of security features, which enables multiple users to have different views with different features. For example, a user in the Sales department cannot see the data that belongs to the Purchase department. Additionally, it can also be managed how much data of the Sales department should be displayed to the user. Since a DBMS is not saved on the disk as traditional file systems, it is very hard for miscreants to break the code.

1.1.2. Users: A typical DBMS has users with different rights and permissions who use it for different purposes. Some users retrieve data and some back it up. The users of a DBMS can be broadly categorized as follows:

- **Administrators** − Administrators maintain the DBMS and are responsible for administrating the database. They are responsible to look after its usage and by whom it should be used. They create access profiles for users and apply limitations to maintain isolation and force security. Administrators also look after DBMS resources like system license, required tools, and other software and hardware related maintenance.
- **Designers** − Designers are the group of people who actually work on the designing part of the database. They keep a close watch on what data should be kept and in what format. They identify and design the whole set of entities, relations, constraints, and views.
- **End Users** − End users are those who actually reap the benefits of having a DBMS. End users can range from simple viewers who pay attention to the logs or market rates to sophisticated users such as business analysts.

1.2. Database System Vs File System

- There are following differences between DBMS and File system:

Table: 1.1. DBMS vs. File System

DBMS	File System
DBMS is a collection of data. In DBMS, the user is not required to write the procedures.	File system is a collection of data. In this system, the user has to write the procedures for managing the database.
DBMS gives an abstract view of data that hides the details.	File system provides the detail of the data representation and storage of data.
DBMS provides a crash recovery mechanism, i.e., DBMS protects the user from the system failure.	File system doesn't have a crash mechanism, i.e., if the system crashes while entering some data, then the content of the file will lost.

DBMS provides a good protection mechanism.	It is very difficult to protect a file under the file system.
DBMS contains a wide variety of sophisticated techniques to store and retrieve the data.	File system can't efficiently store and retrieve the data.
DBMS takes care of Concurrent access of data using some form of locking.	In the File system, concurrent access has many problems like redirecting the file while other deleting some information or updating some information.

1.3. Database System Concept and Architecture

The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

1.3.1. Three-Tier Architecture

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.

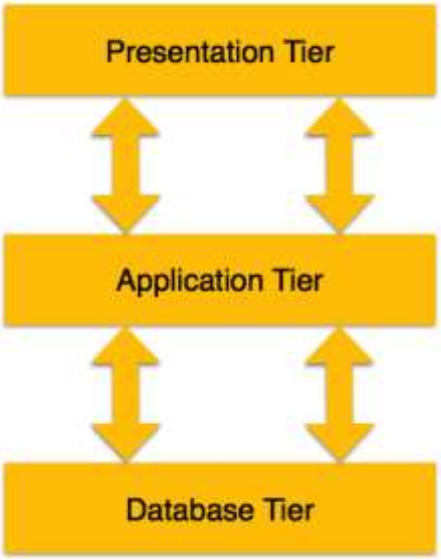


Figure 1.1: Three-Tier Architecture

- **Database (Data) Tier** − At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.
- **Application (Middle) Tier** − At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.
- **User (Presentation) Tier** − End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.

Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

1.4. Database Schema and Database Instance

1.4.1. Database Schema

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It's the database designers who design the schema to help programmers understand the database and make it useful.

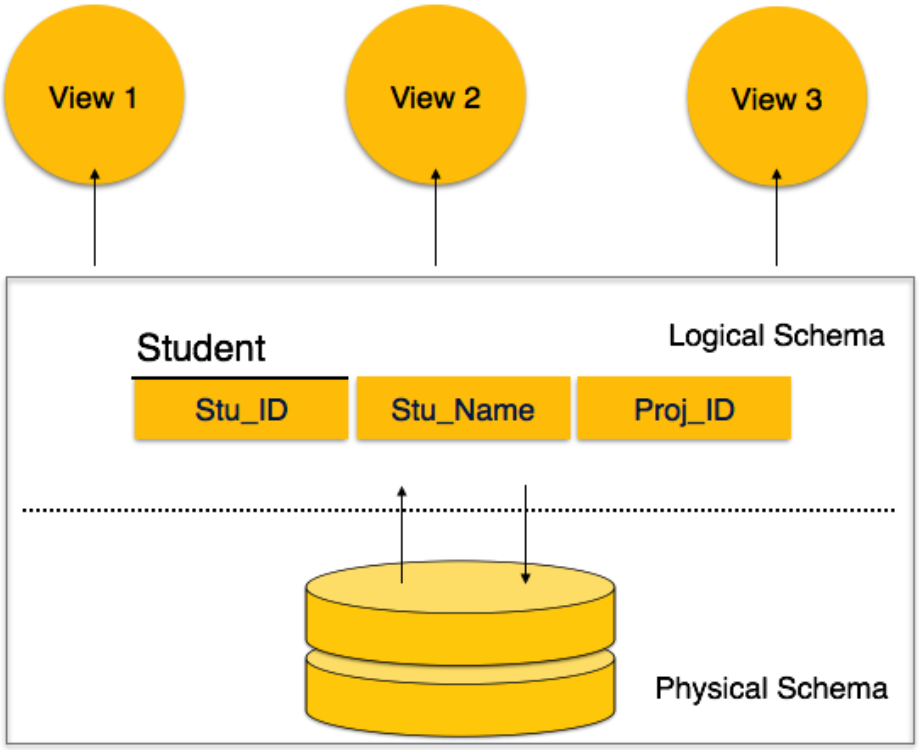


Figure 1.2: Database Schema

A database schema can be divided broadly into two categories −

- **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

1.4.2. Database Instance

It is important that we distinguish these two terms individually. Database schema is the skeleton of database. It is designed when the database doesn't exist at all. Once the database is operational, it is very difficult to make any changes to it. A database schema does not contain any data or information.

A database instance is a state of operational database with data at any given time. It contains a snapshot of the database. Database instances tend to change with time. A DBMS ensures that its

every instance (state) is in a valid state, by diligently following all the validations, constraints, and conditions that the database designers have imposed.

1.5. Data Independence

If a database system is not multi-layered, then it becomes difficult to make any changes in the database system. Database systems are designed in multi-layers as we learnt earlier. A database system normally contains a lot of data in addition to users' data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.

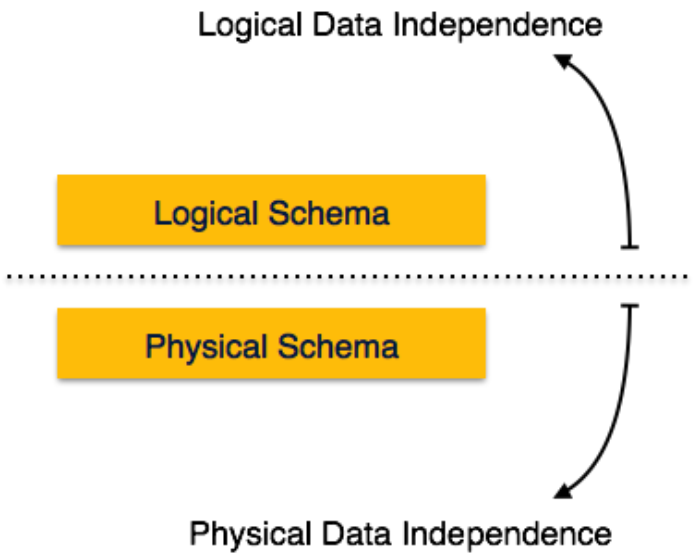


Figure 1.3: Data Independence

Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

Logical Data Independence: Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

Physical Data Independence: All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.

For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

1.6. Database Language

- A DBMS has appropriate languages and interfaces to express database queries and updates.
- Database languages can be used to read, store and update the data in the database.

Types of Database Language

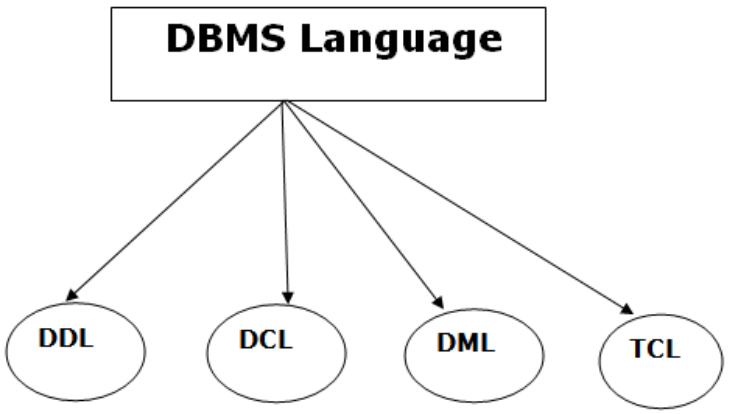


Figure 1.4: Types of Database Language

1.6.1. Data Definition Language

- **DDL** stands for **D**ata **D**efinition **L**anguage. It is used to define database structure or pattern.
- It is used to create schema, tables, indexes, constraints, etc. in the database.
- Using the DDL statements, you can create the skeleton of the database.
- Data definition language is used to store the information of metadata like the number of tables and schemas, their names, indexes, columns in each table, constraints, etc.

Here are some tasks that come under DDL:

- **Create:** It is used to create objects in the database.
- **Alter:** It is used to alter the structure of the database.
- **Drop:** It is used to delete objects from the database.
- **Truncate:** It is used to remove all records from a table.
- **Rename:** It is used to rename an object.
- **Comment:** It is used to comment on the data dictionary.

These commands are used to update the database schema that's why they come under Data definition language.

1.6.2. Data Manipulation Language

DML stands for **D**ata **M**anipulation **L**anguage. It is used for accessing and manipulating data in a database. It handles user requests.

Here are some tasks that come under DML:

- **Select:** It is used to retrieve data from a database.
- **Insert:** It is used to insert data into a table.
- **Update:** It is used to update existing data within a table.
- **Delete:** It is used to delete all records from a table.
- **Merge:** It performs UPSERT operation, i.e., insert or update operations.
- **Call:** It is used to call a structured query language or a Java subprogram.
- **Explain Plan:** It has the parameter of explaining data.
- **Lock Table:** It controls concurrency.

1.6.3. Data Control Language

- **DCL** stands for **D**ata **C**ontrol **L**anguage. It is used to retrieve the stored or saved data.
- The DCL execution is transactional. It also has rollback parameters.

  (But in Oracle database, the execution of data control language does not have the feature of rolling back.)

Here are some tasks that come under DCL:

- **Grant:** It is used to give user access privileges to a database.
- **Revoke:** It is used to take back permissions from the user.

There are the following operations which have the authorization of Revoke:

CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

1.6.4. Transaction Control Language

TCL is used to run the changes made by the DML statement. TCL can be grouped into a logical transaction.

Here are some tasks that come under TCL:

- **Commit:** It is used to save the transaction on the database.
- **Rollback:** It is used to restore the database to original since the last Commit.

1.7. Interfaces in DBMS

A database management system (DBMS) interface is a user interface which allows for the ability to input queries to a database without using the query language itself.

User-friendly interfaces provide by DBMS may include the following:

1. **Menu-Based Interfaces for Web Clients or Browsing –**
   These interfaces present the user with lists of options (called menus) that lead the user through the formation of a request. Basic advantage of using menus is that they removes the tension of remembering specific commands and syntax of any query language, rather than query is basically composed step by step by collecting or picking options from a menu that is basically shown by the system. Pull-down menus are a very popular technique in *Web based interfaces*. They are also often used in *browsing interface* which allow a user to look through the contents of a database in an exploratory and unstructured manner.
2. **Forms-Based Interfaces –**
   A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert a new data, or they can fill out only certain entries, in which case the DBMS will redeem same type of data for other remaining entries. This type of forms are usually designed

or created and programmed for the users that have no expertise in operating system. Many DBMSs have *forms specification languages* which are special languages that help specify such forms.

Example: SQL* Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema.b>

3. **Graphical User Interface –**

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

4. **Natural language Interfaces –**

   These interfaces accept request written in English or some other language and attempt to understand them. A Natural language interface has its own schema, which is similar to the database conceptual schema as well as a dictionary of important words.

   The natural language interface refers to the words in its schema as well as to the set of standard words in a dictionary to interpret the request.If the interpretation is successful, the interface generates a high-level query corresponding to the natural language and submits it to the DBMS for processing, otherwise a dialogue is started with the user to clarify any provided condition or request. The main disadvantage with this is that the capabilities of this type of interfaces are not that much advance.

5. **Speech Input and Output –**

   There is an limited use of speech say it for a query or an answer to a question or being a result of a request it is becoming commonplace Applications with limited vocabularies such as inquiries for telephone directory, flight arrival/departure, and bank account information are allowed speech for input and output to enable ordinary folks to access this information.

   The Speech input is detected using a predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech take place.

6. **Interfaces for DBA –**

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

1.8. Data Definitions Language and DML

DDL is Data Definition Language and is used to define the structures like schema, database, tables, constraints etc. Examples of DDL are create and alter statements.

DML

DML is Data Manipulation Language and is used to manipulate data. Examples of DML are insert, update and delete statements.

Following are the important differences between DDL and DML.

Table 1.2: Differences between DDL and DML

Sr. No.	Key	DDL	DML
1	Stands for	DDL stands for Data Definition Language.	DML stands for Data Manipulation Language.
2	Usage	DDL statements are used to create database, schema, constraints, users, tables etc.	DML statement is used to insert, update or delete the records.
3	Classification	DDL has no further classification.	DML is further classified into procedural DML and non-procedural DML.
4	Commands	CREATE, DROP, RENAME and ALTER.	INSERT, UPDATE and DELETE.

1.9. Overall Database Structure

Database Management System (DBMS) is a software that allows access to data stored in a database and provides an easy and effective method of –

- Defining the information.
- Storing the information.
- Manipulating the information.
- Protecting the information from system crashes or data theft.
- Differentiating access permissions for different users.

The database system is divided into three components: Query Processor, Storage Manager, and Disk Storage. These are explained as following below.

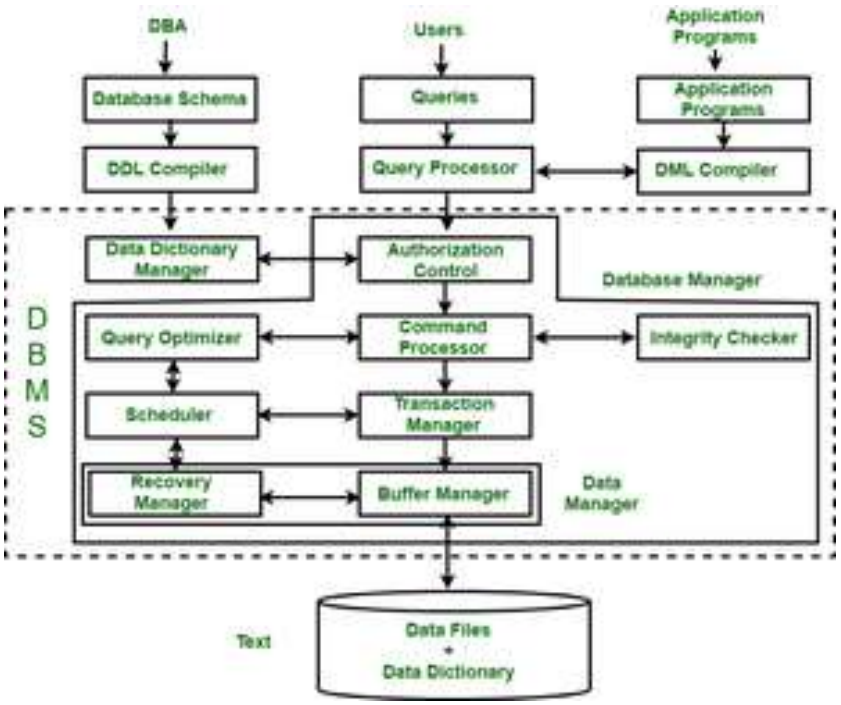


Figure 1.5: Overall Database Structure

1. Query Processor :

It interprets the requests (queries) received from end user via an application program into instructions. It also executes the user request which is received from the DML compiler.

Query Processor contains the following components –

DML Compiler –

It processes the DML statements into low level instruction (machine language), so that they can be executed.

DDL Interpreter –

It processes the DDL statements into a set of table containing meta data (data about data).

Embedded DML Pre-compiler –

It processes DML statements embedded in an application program into procedural calls.

Query Optimizer –

It executes the instruction generated by DML Compiler.

2. Storage Manager :

Storage Manager is a program that provides an interface between the data stored in the database and the queries received. It is also known as Database Control System. It maintains the consistency and integrity of the database by applying the constraints and executes the DCL statements. It is responsible for updating, storing, deleting, and retrieving data in the database.

It contains the following components –

Authorization Manager –

It ensures role-based access control, i.e,. checks whether the particular person is privileged to perform the requested operation or not.

Integrity Manager –

It checks the integrity constraints when the database is modified.

Transaction Manager –

It controls concurrent access by performing the operations in a scheduled way that it receives the transaction. Thus, it ensures that the database remains in the consistent state before and after the execution of a transaction.

File Manager –

It manages the file space and the data structure used to represent information in the database.

Buffer Manager –

It is responsible for cache memory and the transfer of data between the secondary storage and main memory.

3. Disk Storage :

It contains the following components –

Data Files –

It stores the data.

Data Dictionary –

It contains the information about the structure of any database object. It is the repository of information that governs the metadata.

Indices –

It provides faster retrieval of data item.

1.10. Data Modeling Using the Entity Relationship Model

The ER model defines the conceptual view of a database. It works around real-world entities and the associations among them. At view level, the ER model is considered a good option for designing databases.

1.10.1. ER Model Concepts

Entity

An entity can be a real-world object, either animate or inanimate, that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity.

An entity set is a collection of similar types of entities. An entity set may contain entities with attribute sharing similar values. For example, a Students set may contain all the students of a school; likewise a Teachers set may contain all the teachers of a school from all faculties. Entity sets need not be disjoint.

Entities are represented by means of their properties, called **attributes**. All attributes have values. For example, a student entity may have name, class, and age as attributes.

There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of Attributes

- **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits.
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.
- **Derived attribute** − Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data_of_birth.
- **Single-value attribute** − Single-value attributes contain single value. For example − Social_Security_Number.
- **Multi-value attribute** − Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email_address, etc.

These attribute types can come together in a way like −

- simple single-valued attributes

- simple multi-valued attributes
- composite single-valued attributes
- composite multi-valued attributes

Relationship

The association among entities is called a relationship. For example, an employee **works_at** a department, a student **enrolls** in a course. Here, Works_at and Enrolls are called relationships.

Relationship Set

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called **descriptive attributes**.

Degree of Relationship

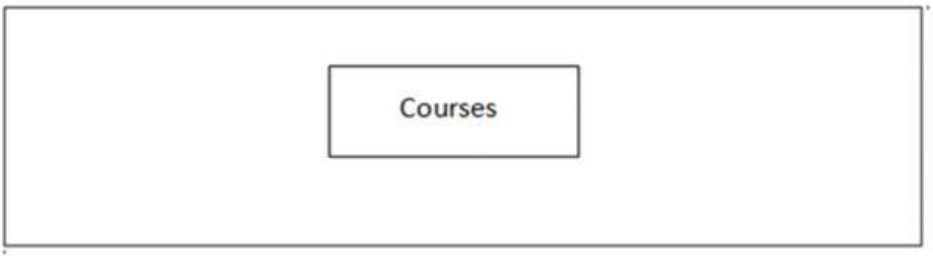
The number of participating entities in a relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

1.10.2. Notations of E-R Diagram

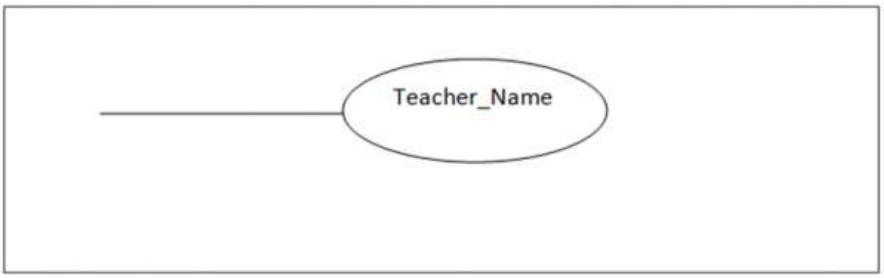
Entity

Entity in DBMS can be a real-world object with an existence, For example, in a **School** database, the entities can be **Teachers**, **Students**, **Courses**, etc.



Attributes

Entities has attributes, which can be considered as properties describing it, for example, for **Teachers** entity, the attributes are **Teacher_Name, Teacher_Address, Teacher_Subject**, etc. The attribute value gets stored in the database.



Weak Entity

The weak entity in DBMS does not have a primary key and are dependent on the parent entity. It mainly depends on other entities, for example, dependents of a professor.

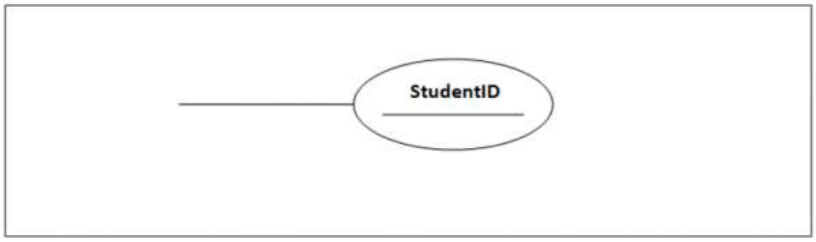
Strong Entity

The strong entity has a primary key. It has weak entities that are dependent on strong entity. Its existence is not dependent on any other entity.

For example, Professor is a strong entity −

Primary Key

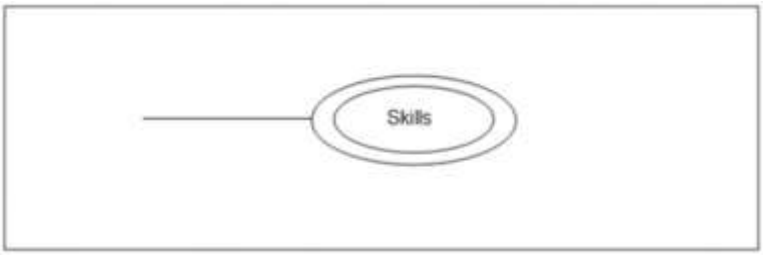
Every table has one Primary key and cannot have null values. A primary key can be **StudentID, SSN, AccountNumber**, etc.



Multivalued Attribute

An attribute that has multiple values for a single entity at a time is called a Multivalued Attribute.

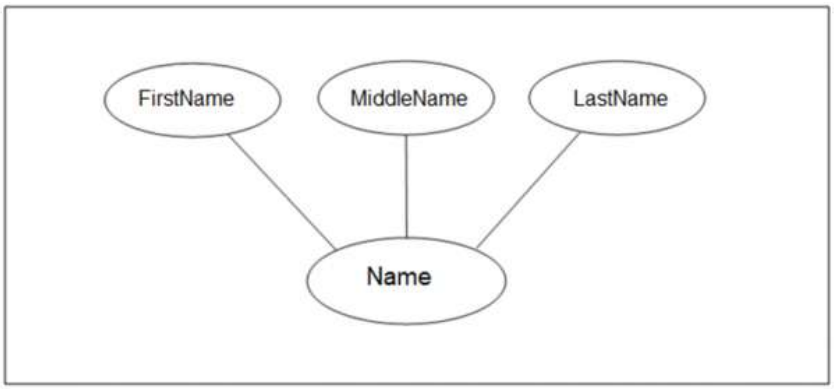
For example, technical skills of a student that can be programming, web development, etc.



Composite Attribute

If an attribute has two or more other attributes, then it is called a Composite Attribute.

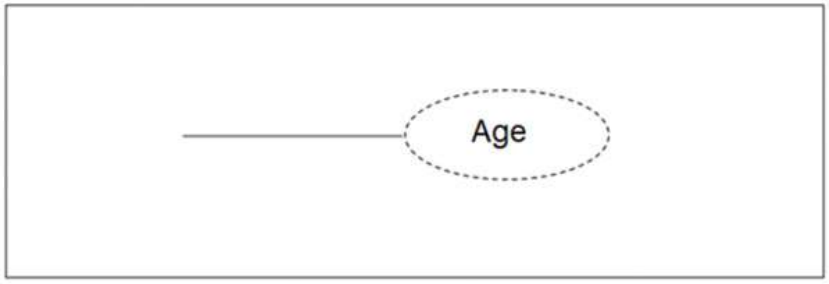
For example, Student Name can be divided as Student First Name, Student Middle Name, and Student Last Name.



Derived Attribute

As the name suggests, the derived attribute is an attribute whose value can be calculated from another attribute.

For example, Student Age can be derived from Date-of-birth of a student.



ER Diagram Example

Here's an ER Diagram for **Hospital**:

- It has three entities: Patient, Doctor and Tests.
- Age is a derived attribute for Patient Entity
- Name in the Tests entity is a Primary Key
- ID in the Doctor entity is a Primary Key
- ID in the Patient entity is a Primary Key

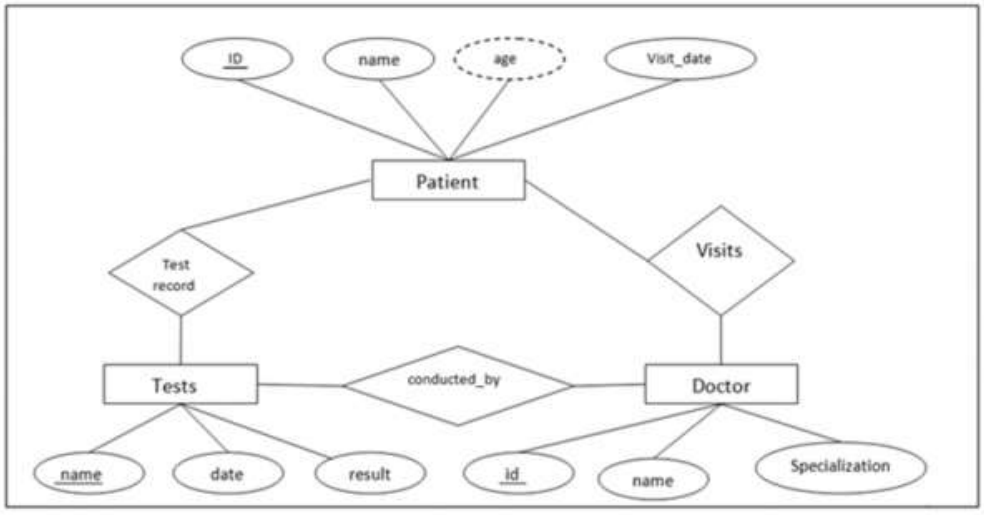
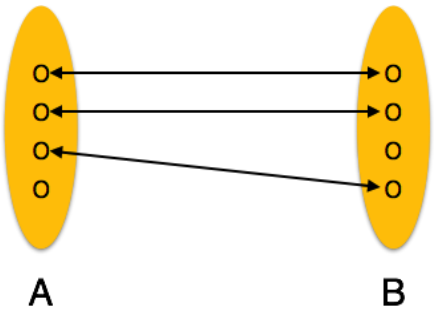


Figure 1.6: ER Diagram for Hospital

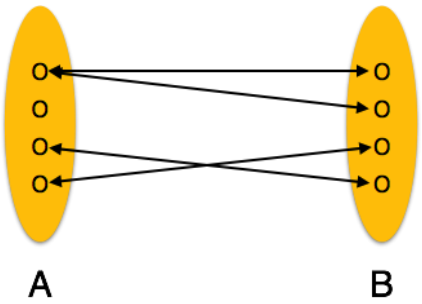
1.10.3 Mapping Cardinalities

Cardinality defines the number of entities in one entity set, which can be associated with the number of entities of other set via relationship set.

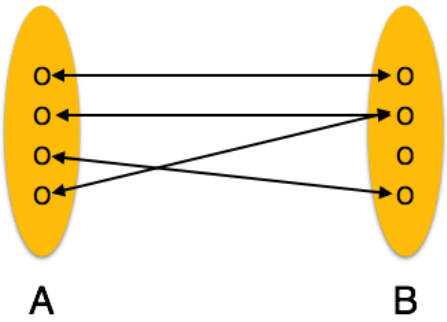
- **One-to-one** − One entity from entity set A can be associated with at most one entity of entity set B and vice versa.



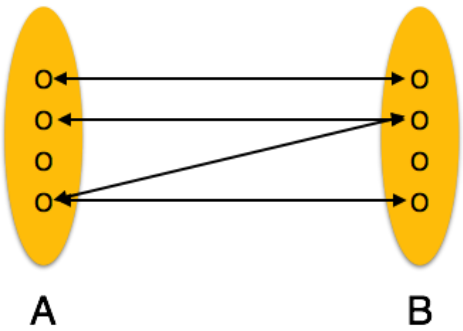
- **One-to-many** − One entity from entity set A can be associated with more than one entities of entity set B however an entity from entity set B, can be associated with at most one entity.



- **Many-to-one** − More than one entities from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A.



- **Many-to-many** − One entity from A can be associated with more than one entity from B and vice versa.



1.10.4. Keys

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, the roll_number of a student makes him/her identifiable among students.

- **Super Key** − A set of attributes (one or more) that collectively identifies an entity in an entity set.
- **Candidate Key** − A minimal super key is called a candidate key. An entity set may have more than one candidate key.
- **Primary Key** − A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.

1.10.5. Generalization and Specialization

Generalization is the process of extracting common properties from a set of entities and create a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher level entity called PERSON as shown in Figure 1. In this case, common attributes like P_NAME, P_ADD become part of higher entity (PERSON) and specialized attributes like S_FEE become part of specialized entity (STUDENT).

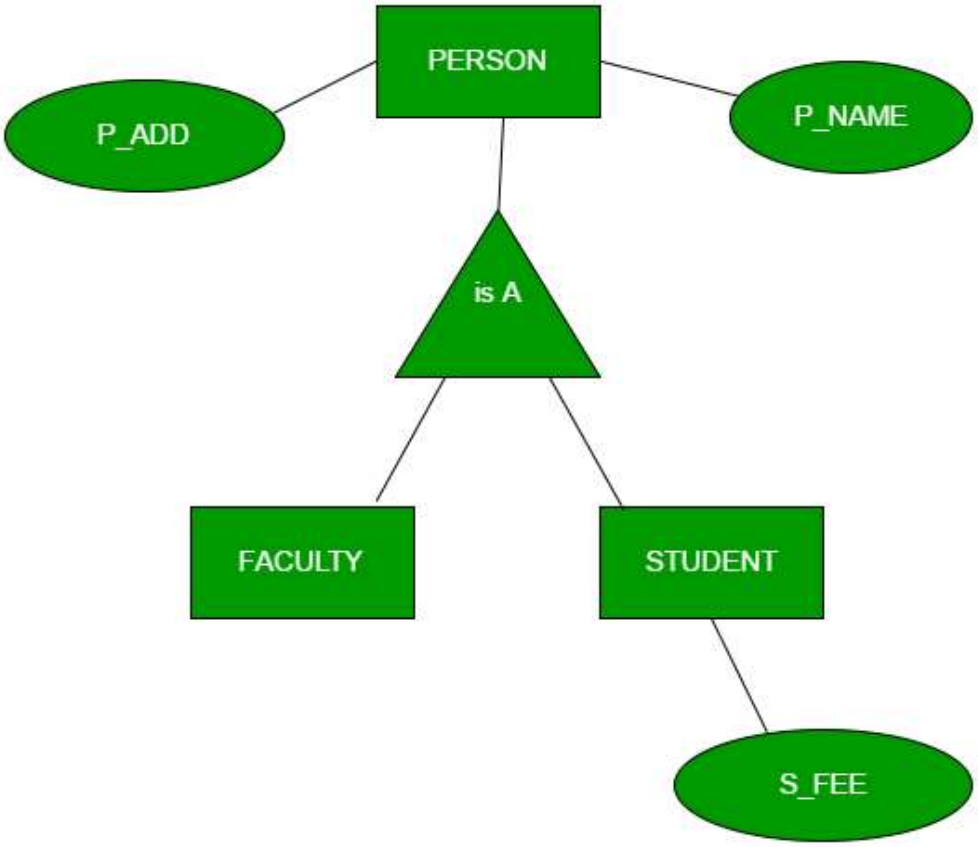


Figure 1.7: Generalization

Specialization

In specialization, an entity is divided into sub-entities based on their characteristics. It is a top-down approach where higher level entity is specialized into two or more lower level entities. For Example, EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER etc. as shown in Figure 2. In this case, common attributes like E_NAME, E_SAL etc. become part of higher entity (EMPLOYEE) and specialized attributes like TES_TYPE become part of specialized entity (TESTER).

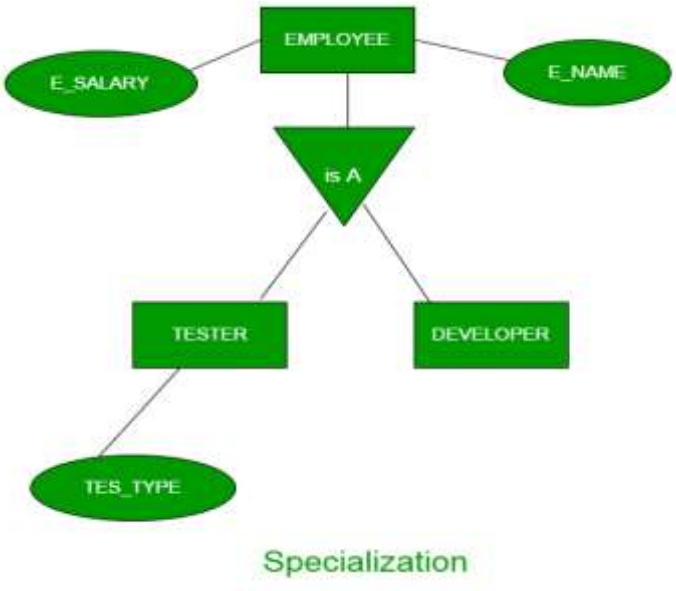


Figure 1.8: Specialization

1.10.6. Aggregation

An ER diagram is not capable of representing relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher level entity. For Example, Employee working for a project may require some machinery. So, REQUIRE relationship is needed between relationship WORKS_FOR and entity MACHINERY. Using aggregation, WORKS_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into single entity and relationship REQUIRE is created between aggregated entity and MACHINERY.

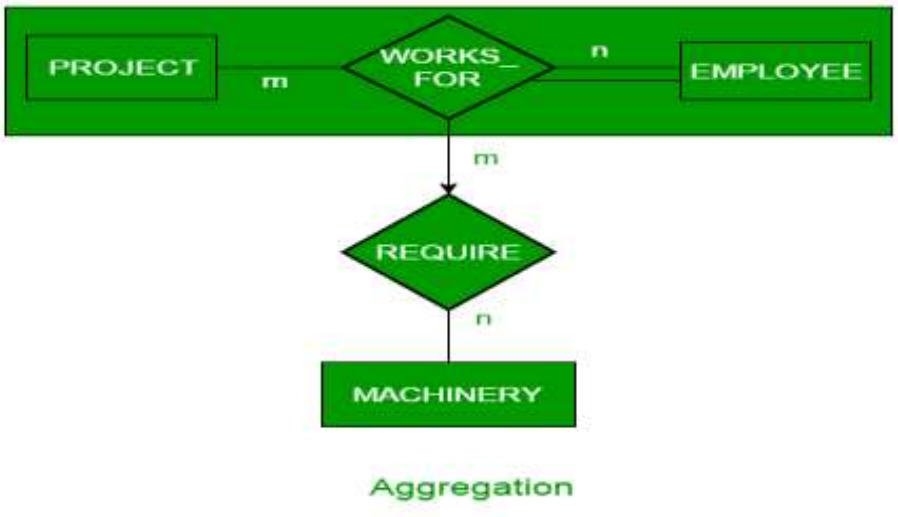


Figure 1.9: Aggregation

1.10.7. Reduction of ER diagram to Table

The database can be represented using the notations, and these notations can be reduced to a collection of tables.

In the database, every entity set or relationship set can be represented in tabular form.

The ER diagram is given below:

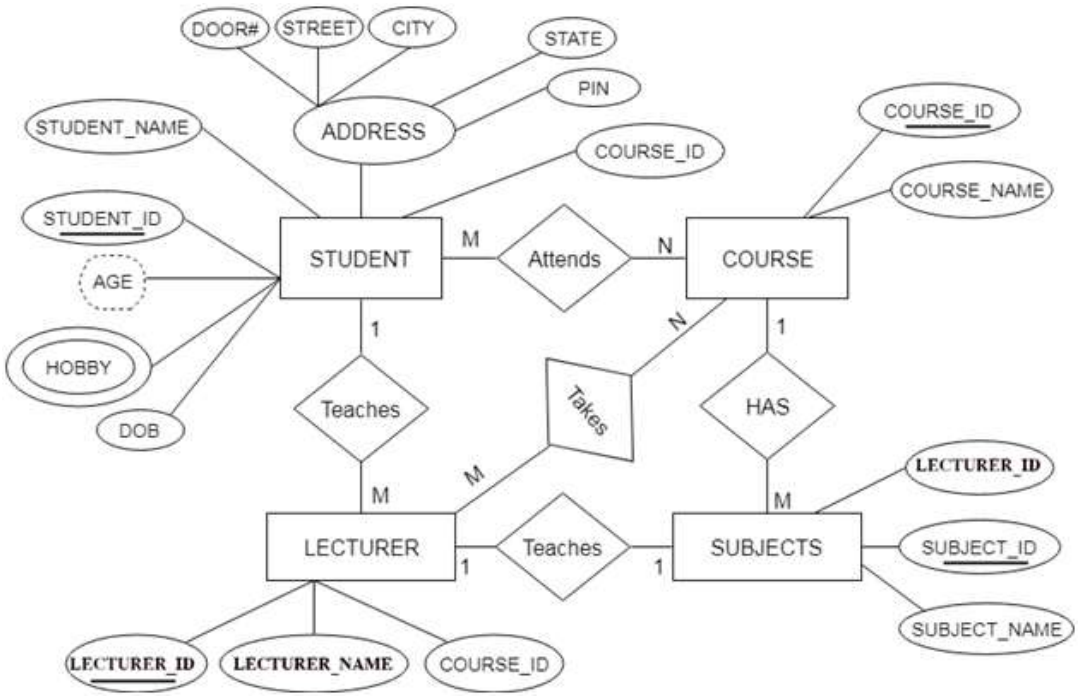


Figure 1.10: Reduction of ER Diagram to Table

There are some points for converting the ER diagram to the table:

- **Entity type becomes a table.**

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

- **All single-valued attribute becomes a column for the table.**

In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly, COURSE_NAME and COURSE_ID form the column of COURSE table and so on.

- **A key attribute of the entity type represented by the primary key.**

In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID, and LECTURE_ID are the key attribute of the entity.

- **The multivalued attribute is represented by a separate table.**

In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.

- **Composite attribute represented by components.**

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.

- **Derived attributes are not considered in the table.**

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:

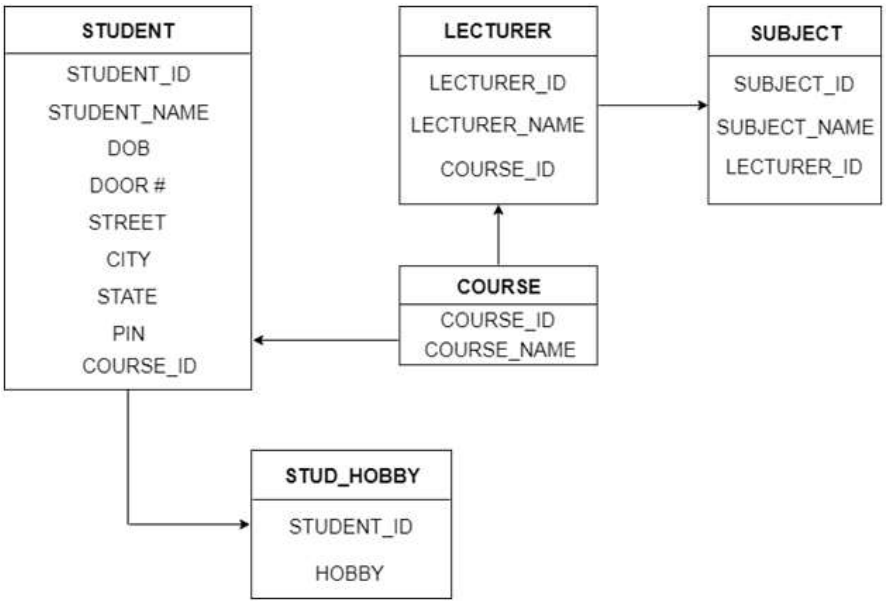


Figure 1.11: ER diagram

1.10.8. Extended ER Model

EER is a high-level data model that incorporates the extensions to the original ER model. Enhanced ERD are high level models that represent the requirements and complexities of complex database.

In addition to ER model concepts EE-R includes −

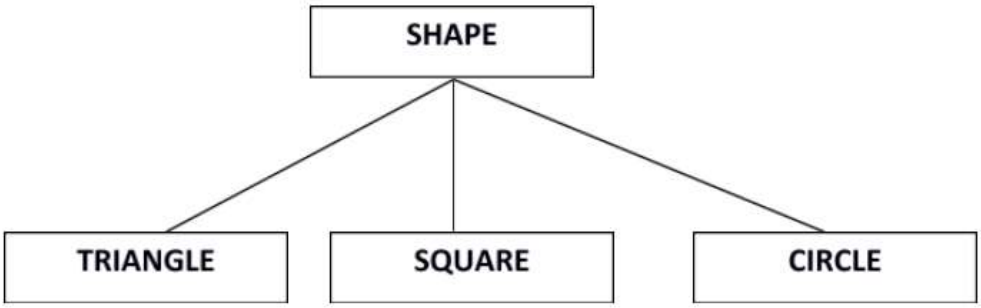
- Subclasses and Super classes.
- Specialization and Generalization.
- Category or union type.
- Aggregation.

These concepts are used to create EE-R diagrams.

Subclasses and Super class

Super class is an entity that can be divided into further subtype.

For **example** − consider Shape super class.

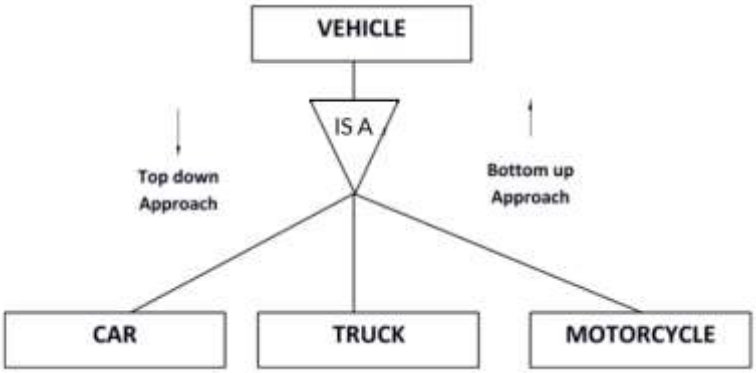


Super class shape has sub groups: Triangle, Square and Circle.

Sub classes are the group of entities with some unique attributes.Sub class inherits the properties and attributes from super class.

Specialization and Generalization

Generalization is a process of generalizing an entity which contains generalized attributes or properties of generalized entities.



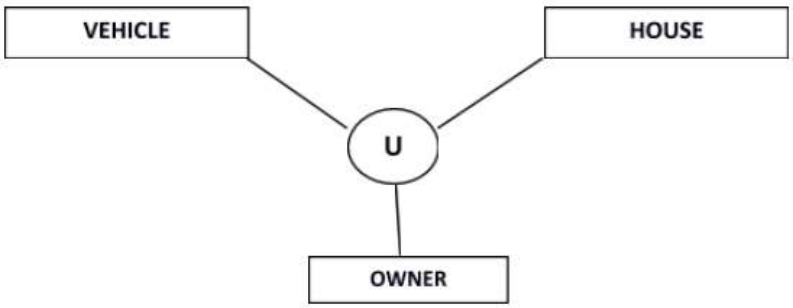
It is a Bottom up process i.e. consider we have 3 sub entities Car, Truck and Motorcycle. Now these three entities can be generalized into one super class named as Vehicle.

Specialization is a process of identifying subsets of an entity that share some different characteristic. It is a top down approach in which one entity is broken down into low level entity.

In above example Vehicle entity can be a Car, Truck or Motorcycle.

Category or Union

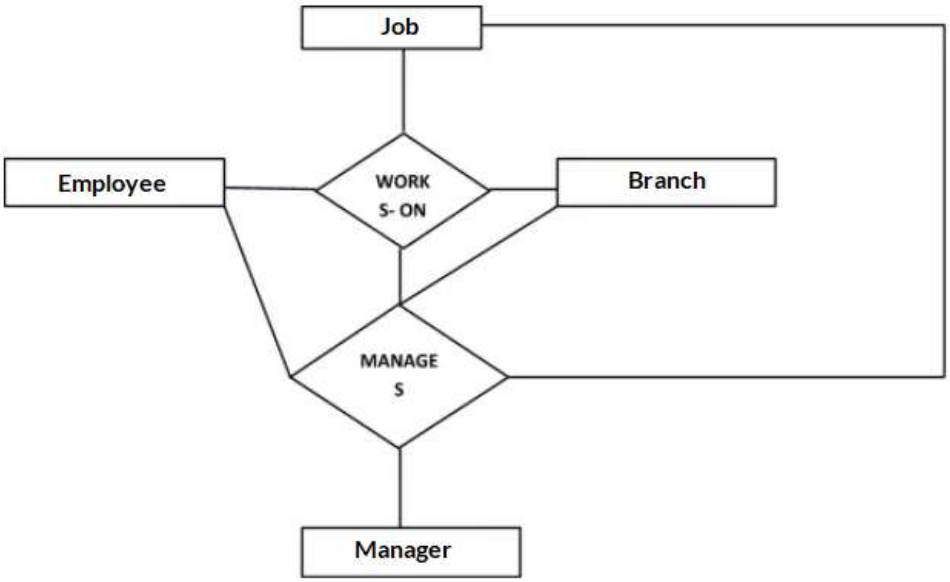
Relationship of one super or sub class with more than one super class.



Owner is the subset of two super class: Vehicle and House.

Aggregation

Represents relationship between a whole object and its component.



Consider a ternary relationship Works_On between Employee, Branch and Manager. Now the best way to model this situation is to use aggregation, So, the relationship-set, Works_On is a higher level entity-set. Such an entity-set is treated in the same manner as any other entity-set. We can create a binary relationship, Manager, between Works_On and Manager to represent who manages what tasks.

1.10.9. Relationship of higher degree

The degree of relationship can be defined as the number of occurrences in one entity that is associated with the number of occurrences in another entity.

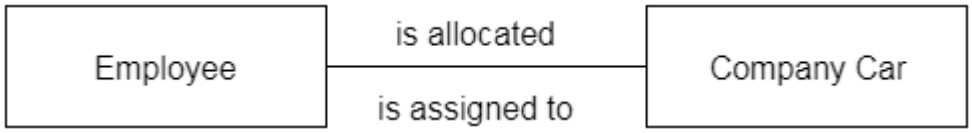
There is the three degree of relationship:

1. One-to-one (1:1)
2. One-to-many (1:M)
3. Many-to-many (M:N)

1. One-to-one

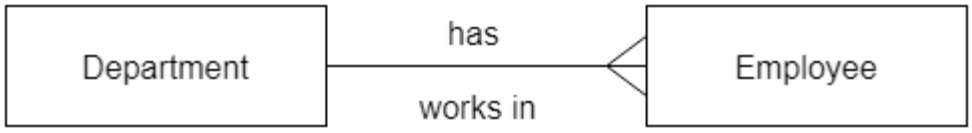
- In a one-to-one relationship, one occurrence of an entity relates to only one occurrence in another entity.

- A one-to-one relationship rarely exists in practice.
- **For example:** if an employee is allocated a company car then that car can only be driven by that employee.
- Therefore, employee and company car have a one-to-one relationship.



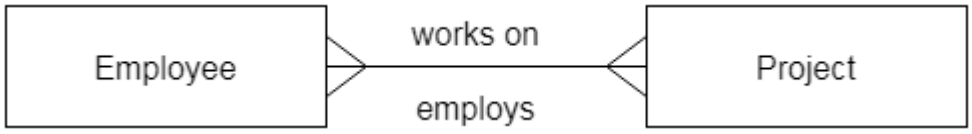
2. One-to-many

- In a one-to-many relationship, one occurrence in an entity relates to many occurrences in another entity.
- **For example:** An employee works in one department, but a department has many employees.
- Therefore, department and employee have a one-to-many relationship.



3. Many-to-many

- In a many-to-many relationship, many occurrences in an entity relate to many occurrences in another entity.
- Same as a one-to-one relationship, the many-to-many relationship rarely exists in practice.
- **For example:** At the same time, an employee can work on several projects, and a project has a team of many employees.
- Therefore, employee and project have a many-to-many relationship.



IMPORTANT QUESTIONS

1. Consider following relational schema:
   Student (s_no, s_name, s_address, class), Teacher (t_name, t_address, dept, specialization)
   Lecture_schedule (day, time, room_no, class, subject), attendance (data, subject, s_no)
   Draw E-R diagram. Identify the relationships and also indicate mapping cardinalities.
2. Describe the three-schema architecture. Why do we need mappings between schema levels?
3. Discuss the main characteristics of database approach and how it differs from traditional file system.
4. When is the concept of a weak entity type used in data modeling? Define the terms owner entity type, weak entity type, identifying relationship type and partial key.
5. What is the difference between specialization and generalization? Why we do not display the difference in schema diagram?
6. What is the Data Model? Explain Data independence with suitable diagram.
7. Explain Super key, Candidate key, Primary key and Partial key with example.
8. Describe Generalization, Specialization with example and reduce it into tables (Record need not require).
9. Draw the E-R Diagram of registration process of the student in a particular course, convert the ER diagram into tables also (Records need not require).
10. Explain all database languages in detail with examples.

MULTIPLE CHOICE QUESTIONS

1. Which of the following gives a logical structure of the database graphically?

a) Entity-relationship diagram
b) Entity diagram
c) Database diagram
d) Architectural representation

2. The entity relationship set is represented in E-R diagram as

a) Double diamonds
b) Undivided rectangles
c) Dashed lines
d) Diamond

3. The Rectangles divided into two parts represents
a) Entity set
b) Relationship set
c) Attributes of a relationship set
d) Primary key

4. Consider a directed line(->) from the relationship set advisor to both entity sets instructor and student. This indicates _____ cardinality
a) One to many
b) One to one
c) Many to many
d) Many to one

5. We indicate roles in E-R diagrams by labeling the lines that connect _____ to _____
a) Diamond , diamond
b) Rectangle, diamond
c) Rectangle, rectangle
d) Diamond, rectangle

6. An entity set that does not have sufficient attributes to form a primary key is termed a _____
a) Strong entity set
b) Variant set
c) Weak entity set
d) Variable set

7. For a weak entity set to be meaningful, it must be associated with another entity set, called the
a) Identifying set
b) Owner set
c) Neighbour set
d) Strong entity set

8. Weak entity set is represented as
a) Underline
b) Double line
c) Double diamond
d) Double rectangle

9. If you were collecting and storing information about your music collection, an album would be considered a(n) _____
a) Relation
b) Entity
c) Instance
d) Attribute

10. What term is used to refer to a specific record in your music database; for instance; information stored about a specific album?
a) Relation
b) Instance
c) Table
d) Column

11. The entity set person is classified as student and employee. This process is called _____
a) Generalization
b) Specialization
c) Inheritance
d) Constraint generalization

12. Which relationship is used to represent a specialization entity?
a) ISA
b) AIS
c) ONIS
d) WHOIS

13. The refinement from an initial entity set into successive levels of entity subgroupings represents a _____ design process in which distinctions are made explicit.
a) Hierarchy
b) Bottom-up
c) Top-down
d) Radical

14. There are similarities between the instructor entity set and the secretary entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the identifier, name, and salary attributes. This process is called
a) Commonality
b) Specialization
c) Generalization
d) Similarity

15. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has
a) Hierarchy
b) Multilevel inheritance
c) Single inheritance
d) Multiple inheritance

16. A _____ constraint requires that an entity belong to no more than one lower-level entity set.
a) Disjointness
b) Uniqueness
c) Special
d) Relational

17. Consider the employee work-team example, and assume that certain employees participate in more than one work team. A given employee may therefore appear in more than one of the team entity sets that are lower level entity sets of employee. Thus, the generalization is _____
a) Overlapping
b) Disjointness
c) Uniqueness
d) Relational

18. The completeness constraint may be one of the following: Total generalization or specialization, Partial generalization or specialization. Which is the default?
a) Total
b) Partial
c) Should be specified
d) Cannot be determined

19. Functional dependencies are a generalization of
a) Key dependencies
b) Relation dependencies
c) Database dependencies
d) None of the mentioned

20. Which of the following is another name for a weak entity?
a) Child
b) Owner
c) Dominant
d) All of the mentioned

21. Database _____ which is the logical design of the database, and the database _____ which is a snapshot of the data in the database at a given instant in time.
a) Instance, Schema
b) Relation, Schema
c) Relation, Domain
d) Schema, Instance

22. A domain is atomic if elements of the domain are considered to be _____ units.
a) Different
b) Indivisbile
c) Constant
d) Divisible

23. Which one of the following is a set of one or more attributes taken collectively to uniquely identify a record?
a) Candidate key
b) Sub key
c) Super key
d) Foreign key

24. Consider attributes ID, CITY and NAME. Which one of this can be considered as a super key?
a) NAME
b) ID
c) CITY
d) CITY, ID

25. The subset of a super key is a candidate key under what condition?
a) No proper subset is a super key
b) All subsets are super keys
c) Subset is a super key
d) Each subset is a super key

26. A _____ is a property of the entire relation, rather than of the individual tuples in which each tuple is unique.
a) Rows
b) Key
c) Attribute
d) Fields

27. Which one of the following attribute can be taken as a primary key?

a) Name
b) Street
c) Id
d) Department

28. Which one of the following cannot be taken as a primary key?
a) Id
b) Register number
c) Dept_id
d) Street

29. An attribute in a relation is a foreign key if the _____ key from one relation is used as an attribute in that relation.
a) Candidate
b) Primary
c) Super
d) Sub

30. The relation with the attribute which is the primary key is referenced in another relation. The relation which has the attribute as a primary key is called _____
a) Referential relation
b) Referencing relation
c) Referenced relation
d) Referred relation

31. The _____ is the one in which the primary key of one relation is used as a normal attribute in another relation.
a) Referential relation
b) Referencing relation
c) Referenced relation
d) Referred relation

32. A _____ integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.
a) Referential
b) Referencing
c) Specific
d) Primary

33. An _____ is a set of entities of the same type that share the same properties, or attributes.
a) Entity set
b) Attribute set
c) Relation set
d) Entity model

34. Entity is a _____
a) Object of relation
b) Present working model
c) Thing in real world
d) Model of relation

35. The descriptive property possessed by each entity set is _____
a) Entity
b) Attribute
c) Relation
d) Model

36. The function that an entity plays in a relationship is called that entity's _____
a) Participation
b) Position

c) Role

d) Instance

37. The attribute *name* could be structured as an attribute consisting of first name, middle initial, and last name. This type of attribute is called

a) Simple attribute

b) Composite attribute

c) Multivalued attribute

d) Derived attribute

38. Not applicable condition can be represented in relation entry as

a) NA

b) 0

c) NULL

d) Blank Space

39. Which of the following can be a multivalued attribute?

a) Phone_number

b) Name

c) Date_of_birth

d) All of the mentioned

40. In a relation between the entities the type and condition of the relation should be specified. That is called as_____attribute.

a) Desciptive

b) Derived

c) Recursive

d) Relative

Answer Key

1	a	**11**	b	**21**	d	**31**	c
2	d	**12**	a	**22**	b	**32**	a
3	a	**13**	c	**23**	c	**33**	a
4	b	**14**	c	**24**	b	**34**	c
5	d	**15**	d	**25**	a	**35**	b
6	c	**16**	a	**26**	b	**36**	c
7	a	**17**	a	**27**	c	**37**	b
8	c	**18**	b	**28**	d	**38**	c
9	b	**19**	a	**29**	b	**39**	a
10	d	**20**	a	**30**	c	**40**	a

UNIT 2

RELATIONAL DATA MODEL AND LANGUAGE

Relational Model (RM) represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

Some popular Relational Database management systems are:

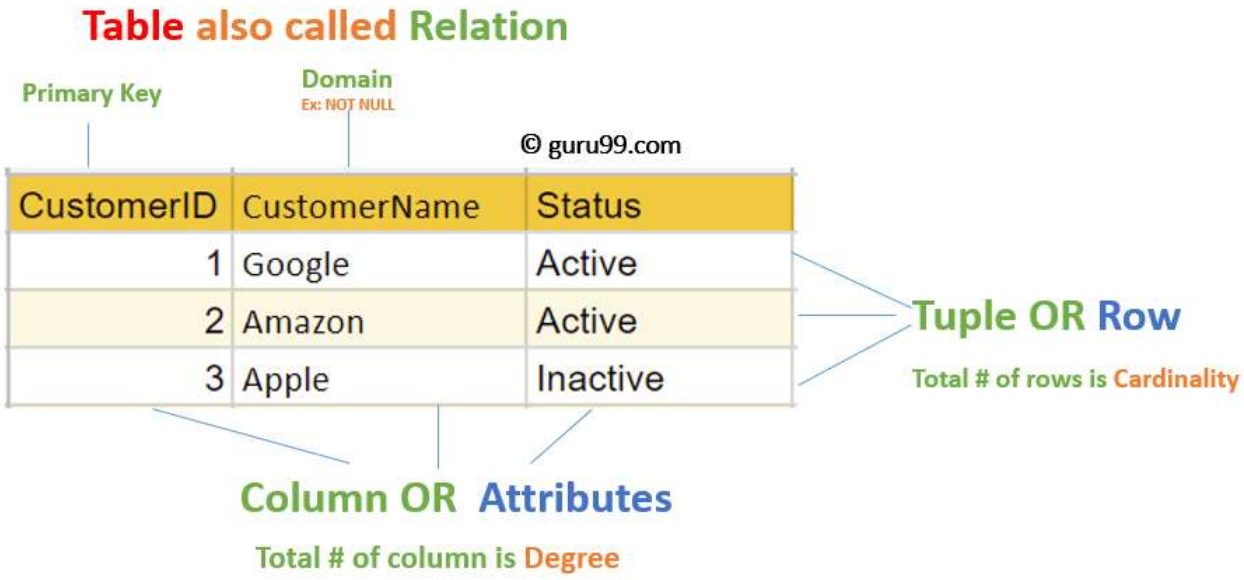
- DB2 and Informix Dynamic Server - IBM
- Oracle and RDB – Oracle
- SQL Server and Access - Microsoft

2.1. Relational Model Concepts

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student, Rollno, NAME,etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree:** The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:** Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. **Relation key** - Every row has one, two or multiple attributes, which is called relation key.

10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain

Table 2.1: Relational



2.2. Relational Integrity Constraints

Relational Integrity constraints in DBMS are referred to conditions which must be present for a valid relation. These Relational constraints in DBMS are derived from the rules in the mini-world that the database represents.

There are many types of Integrity Constraints in DBMS. Constraints on the Relational database management system is mostly divided into three main categories are:

1. Domain Constraints
2. Key Constraints
3. Referential Integrity Constraints
4. Entity Integrity Constraint

2.3. Domain Integrity Constraints

Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type.

Domain constraints specify that within each tuple, and the value of each attribute must be unique. This is specified as data types which include standard data types integers, real numbers, characters, Booleans, variable length strings, etc.

Example:

```
Create DOMAIN CustomerName
CHECK (value not NULL)
```

The example shown demonstrates creating a domain constraint such that CustomerName is not NULL

Key Constraints

An attribute that can uniquely identify a tuple in a relation is called the key of the table. The value of the attribute for different tuples in the relation has to be unique.

Example:

In the given table, CustomerID is a key attribute of Customer Table. It is most likely to have a single key for one customer, CustomerID =1 is only for the CustomerName =" Google".

Table 2.2

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

2.4. Entity integrity constraints

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

Example:

Table 2.3

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. There are two kinds of query languages − relational algebra and relational calculus.

2.5. Referential Integrity Constraints

Referential Integrity constraints in DBMS are based on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

Example: **Table 2.4**

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Customer

Billing

InvoiceNo	CustomerID	Amount
1	1	$100
2	1	$200
3	2	$150

In the above example, we have 2 relations, Customer and Billing.

Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount $300

2.6. Relational Algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

We will discuss all these operations in the following sections.

Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation.

Notation − $\sigma_p(r)$

Where **σ** stands for selection predicate and **r** stands for relation. *p* is prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

For example −

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

Output − Selects tuples from books where subject is 'database'.

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

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

$\sigma_{subject = "database" \text{ and } price = "450" \text{ or } year > "2010"}(Books)$

Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

Project Operation (∏)

It projects column(s) that satisfy a given predicate.

Notation − $\prod_{A1}, A_2, A_n (r)$

Where A_1, A_2 , A_n are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

For example −

$\prod_{subject, author} (Books)$

Selects and projects columns named as subject and author from the relation Books.

Union Operation (∪)

It performs binary union between two given relations and is defined as −

$r \cup s = \{ t \mid t \in r \text{ or } t \in s\}$

Notation − r U s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

- **r**, and **s** must have the same number of attributes.
- Attribute domains must be compatible.

- Duplicate tuples are automatically eliminated.

$\prod_{author}$ (Books) ∪ $\prod_{author}$ (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − **r** − **s**

Finds all the tuples that are present in **r** but not in **s**.

$\prod_{author}$ (Books) − $\prod_{author}$ (Articles)

Output − Provides the name of authors who have written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where **r** and **s** are relations and their output will be defined as −

r X s = { q t | q ∈ r and t ∈ s}

$\sigma_{author = 'roopam'}$(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by roopam.

Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho** ρ.

Notation − ρ_x (E)

Where the result of expression **E** is saved with name of **x**.

Additional operations are −

- Set intersection
- Assignment
- Natural join

2.7. Relational Calculus

In contrast to Relational Algebra, Relational Calculus is a non-procedural query language, that is, it tells what to do but never explains how to do it.

Relational calculus exists in two forms −

Tuple Relational Calculus (TRC)

Filtering variable ranges over tuples

Notation − {T | Condition}

Returns all tuples T that satisfies a condition.

For example −

{ T.name | Author(T) AND T.article = 'database' }

Output − Returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified. We can use Existential (∃) and Universal Quantifiers (∀).

For example −

{ R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output − The above query will yield the same result as the previous one.

Domain Relational Calculus (DRC)

In DRC, the filtering variable uses the domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation −

$\{ a_1, a_2, a_3, ..., a_n \mid P(a_1, a_2, a_3, ... ,a_n)\}$

Where a1, a2 are attributes and **P** stands for formulae built by inner attributes.

For example −

{< article, page, subject > | ∈ Ram ∧ subject = 'database'}

Output − Yields Article, Page, and Subject from the relation Ram, where subject is database.

Just like TRC, DRC can also be written using existential and universal quantifiers. DRC also involves relational operators.

The expression power of Tuple Relation Calculus and Domain Relation Calculus is equivalent to Relational Algebra.

2.8. Introduction on SQL

SQL is a standard language for accessing and manipulating databases.

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases

- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

2.8.1. Characteristics of SQL

- SQL is easy to learn.
- SQL is used to access data from relational database management systems.
- SQL can execute queries against the database.
- SQL is used to describe the data.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL allows users to set permissions on tables, procedures, and views.

Advantages of SQL

There are the following advantages of SQL:

High speed

Using the SQL queries, the user can quickly and efficiently retrieve a large amount of records from a database.

No coding needed

In the standard SQL, it is very easy to manage the database system. It doesn't require a substantial amount of code to manage the database system.

Well defined standards

Long established are used by the SQL databases that are being used by ISO and ANSI.

Portability

SQL can be used in laptop, PCs, server and even some mobile phones.

Interactive language

SQL is a domain language used to communicate with the database. It is also used to receive answers to the complex questions in seconds.

Multiple data view

Using the SQL language, the users can make different views of the database structure.

2.8.2. SQL Data Types and Literals

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use which are listed below −

Table 2.5: Exact Numeric Data Types

DATA TYPE	FROM	TO
Bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
Int	-2,147,483,648	2,147,483,647
Smallint	-32,768	32,767
Tinyint	0	255
Bit	0	1
Decimal	-10^38 +1	10^38 -1
Numeric	-10^38 +1	10^38 -1
Money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
Smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

Table 2.6: Approximate Numeric Data Types

DATA TYPE	FROM	TO
Float	-1.79E + 308	1.79E + 308

Real	-3.40E + 38	3.40E + 38

Date and Time Data Types

Table 2.7: Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

Table 2.8: Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**Char** Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	**Varchar** Maximum of 8,000 characters.(Variable-length non-Unicode data).
3	**varchar(max)** Maximum length of 2E + 31 characters, Variable-length non-Unicode data (SQL Server 2005 only).
4	**Text** Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

Table 2.9

Sr.No.	DATA TYPE & Description
1	**Nchar** Maximum length of 4,000 characters.(Fixed length Unicode)
2	**Nvarchar** Maximum length of 4,000 characters.(Variable length Unicode)
3	**nvarchar(max)** Maximum length of 2E + 31 characters (SQL Server 2005 only).(Variable length Unicode)
4	**ntext** Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types

Table 2.10: Binary Data Types

Sr.No.	DATA TYPE & Description
1	**Binary**: Maximum length of 8,000 bytes(Fixed-length binary data)
2	**Varbinary**: Maximum length of 8,000 bytes.(Variable length binary data)
3	**varbinary(max)**:Maximum length of 2E + 31 bytes (SQL Server 2005 only). (Variable length Binary data)
4	**Image**: Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Misc Data Types

Table 2.11: Misc Data Types

Sr.No.	DATA TYPE & Description
1	**sql_variant**: Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
2	**Timestamp**: Stores a database-wide unique number that gets updated every time a row gets updated
3	**Uniqueidentifier**: Stores a globally unique identifier (GUID)
4	**Xml**: Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
5	**Cursor**: Reference to a cursor object
6	**Table**: Stores a result set for later processing

2.8.3. SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.

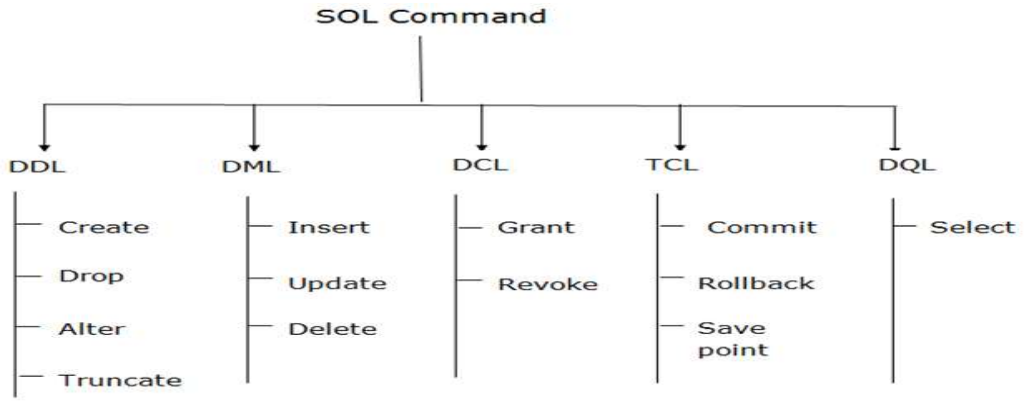


Figure 2.1: Types of SQL Commands

1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. CREATE It is used to create a new table in the database.

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

Example:

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

b. DROP: It is used to delete both the structure and record stored in the table.

Syntax

DROP TABLE ;

Example

DROP TABLE EMPLOYEE;

c. ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);

EXAMPLE

ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));

ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

d. TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE EMPLOYEE;

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. INSERT: The INSERT statement is a SQL query. It is used to insert data into the row of a table.

Syntax:

INSERT INTO TABLE_NAME

(col1, col2, col3,.... col N)

VALUES (value1, value2, value3, ... valueN);

Or

INSERT INTO TABLE_NAME

VALUES (value1, value2, value3, ... valueN);

For example:

INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");

b. UPDATE: This command is used to update or modify the value of a column in the table.

Syntax:

UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITI ON]

For example:

```
UPDATE students
SET User_Name = 'Sonoo'
WHERE Student_Id = '3'
```

c. DELETE: It is used to remove one or more row from a table.

Syntax:

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
DELETE FROM javatpoint
WHERE Author="Sonoo";
```

3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

a. Grant: It is used to give user access privileges to a database.

Example

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

b. Revoke: It is used to take back permissions from the user.

Example

```
REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

a. Commit: Commit command is used to save all the transactions to the database.

Syntax:

COMMIT;

Example:

DELETE FROM CUSTOMERS
WHERE AGE = 25;
COMMIT;

b. Rollback: Rollback command is used to undo transactions that have not already been saved to the database.

Syntax:

ROLLBACK;

Example:

DELETE FROM CUSTOMERS

```
WHERE AGE = 25;
ROLLBACK;
```

c. SAVEPOINT: It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```

5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

a. SELECT: This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

Syntax:

```
SELECT expressions
FROM TABLES
WHERE conditions;
```

For example:

```
SELECT emp_name
FROM employee
WHERE age > 20;
```

2.8.4. SQL Operators and Their Procedure

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then –

Table 2.12: SQL Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	a + b will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand.	a - b will give -10
* (Multiplication)	Multiplies values on either side of the operator.	a * b will give 200
/ (Division)	Divides left hand operand by right hand operand.	b / a will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder.	b % a will give 0

SQL Comparison Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then –

Table 2.13: SQL Comparison Operators

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators

Here is a list of all the logical operators available in SQL.

Table 2.14: SQL Logical Operators

Sr.No.	Operator & Description
1	**ALL**: The ALL operator is used to compare a value to all values in another value set.
2	**AND**: The AND operator allows the existence of multiple conditions in an SQL statement's

	WHERE clause.
3	**ANY**: The ANY operator is used to compare a value to any applicable value in the list as per the condition.
4	**BETWEEN**: The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
5	**EXISTS**: The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion.
6	**IN**: The IN operator is used to compare a value to a list of literal values that have been specified.
7	**LIKE**: The LIKE operator is used to compare a value to similar values using wildcard operators.
8	**NOT**: The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
9	**OR**: The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
10	**IS NULL**: The NULL operator is used to compare a value with a NULL value.
11	**UNIQUE**: The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

2.8.5. SQL Table

Table is a collection of data, organized in terms of rows and columns. In DBMS term, table is known as relation and row as tuple.

Table is the simple form of data storage. A table is also considered as a convenient representation of relations.

Let's see an example of an employee table:

Table 2.15: Example

Employee		
EMP_NAME	**ADDRESS**	**SALARY**
Ankit	Lucknow	15000
Raman	Allahabad	18000
Mike	New York	20000

In the above table, "Employee" is the table name, "EMP_NAME", "ADDRESS" and "SALARY" are the column names. The combination of data of multiple columns forms a row e.g. "Ankit", "Lucknow" and 15000 are the data of one row.

SQL TABLE Variable

The **SQL Table variable** is used to create, modify, rename, copy and delete tables. Table variable was introduced by Microsoft.

It was introduced with SQL server 2000 to be an alternative of temporary tables.

It is a variable where we temporary store records and results. This is same like temp table but in the case of temp table we need to explicitly drop it.

Table variables are used to store a set of records. So declaration syntax generally looks like CREATE TABLE syntax.

create table "tablename" ("column1" "data type", "column2" "data type", ... "columnN" "data type") ;

When a transaction rolled back the data associated with table variable is not rolled back.

A table variable generally uses lesser resources than a temporary variable.

Table variable cannot be used as an input or an output parameter.

2.8.6. Views in database

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

In this article we will learn about creating, deleting and updating Views.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

Creating Views

We can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2.....

FROM table_name

WHERE condition;

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

Examples:

- **Creating View from a single table:**
    - In this example we will create a View named DetailsView from the table StudentDetails.

      Query:
    - CREATE VIEW DetailsView AS
    - SELECT NAME, ADDRESS
    - FROM StudentDetails
    - WHERE S_ID < 5;

      To see the data in the View, we can query the view in the same manner as we query a table.

      SELECT * FROM DetailsView;

      Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

    - In this example, we will create a view named StudentNames from the table StudentDetails.

      Query:

      CREATE VIEW StudentNames AS

SELECT S_ID, NAME

FROM StudentDetails

ORDER BY NAME;

If we now query the view as,

SELECT * FROM StudentNames;

Output:

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

- **Creating View from multiple tables**: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:
- CREATE VIEW MarksView AS
- SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
- FROM StudentDetails, StudentMarks
- WHERE StudentDetails.NAME = StudentMarks.NAME;

To display data of View MarksView:

SELECT * FROM MarksView;

Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

Deleting Views

We have learned about creating a View, but what if a created View is not needed any more? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

DROP VIEW view_name;

view_name: Name of the View which we want to delete.

For example, if we want to delete the View **MarksView**, we can do this as:

DROP VIEW MarksView;

Updating Views

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

- We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.

Syntax:

- CREATE OR REPLACE VIEW view_name AS
- SELECT column1,coulmn2,..
- FROM table_name
- WHERE condition;

For example, if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table, we can do this as:

CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;

If we fetch all the data from MarksView now as:

SELECT * FROM MarksView;

Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

- **Inserting a row in a view**:
  We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.**Syntax**:
- INSERT INTO view_name(column1, column2 , column3,..)
- VALUES(value1, value2, value3..);
- **view_name**: Name of the View
  Example:
  In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".
  INSERT INTO DetailsView(NAME, ADDRESS)

  VALUES("Suresh","Gurgaon");

  If we fetch all the data from DetailsView now as,

  SELECT * FROM DetailsView;

  Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

- **Deleting a row from a View**:

  Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**Syntax**:

- DELETE FROM view_name
- WHERE condition;
- **view_name**:Name of view from where we want to delete rows
- **condition**: Condition to select rows

  Example:

  In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

  DELETE FROM DetailsView

  WHERE NAME="Suresh";

  If we fetch all the data from DetailsView now as,

  SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

SQL Indexes

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows −

```
DROP INDEX index_name;
```

You can check the INDEX Constraint chapter to see some actual examples on Indexes.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

2.8.7. SQL Sub Query

A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

```
SELECT column_name
```

FROM table_name

WHERE column_name expression operator

(SELECT column_name from table_name WHERE ...);

Example

Consider the EMPLOYEE table have the following records:

Table 2.16

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be:

SELECT *

FROM EMPLOYEE

WHERE ID IN (SELECT ID

FROM EMPLOYEE

WHERE SALARY > 4500);

This would produce the following result:

Table 2.17

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00

5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

2. Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

INSERT INTO table_name (column1, column2, column3....)
SELECT *
FROM table_name
WHERE VALUE OPERATOR

Example

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
INSERT INTO EMPLOYEE_BKP
SELECT * FROM EMPLOYEE
WHERE ID IN (SELECT ID
FROM EMPLOYEE);
```

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

```
UPDATE table
SET column_name = new_value
WHERE VALUE OPERATOR
(SELECT COLUMN_NAME
 FROM TABLE_NAME
WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
 WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

Table 2.18

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

```
1. DELETE FROM TABLE_NAME
2. WHERE VALUE OPERATOR
3.    (SELECT COLUMN_NAME
4.    FROM TABLE_NAME
5.    WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

```
DELETE FROM EMPLOYEE
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
WHERE AGE >= 29 );
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

Table 2.18

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

2.8.8. Aggregate functions in SQL

In database management an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

Various Aggregate Functions

1) Count()

2) Sum()

3) Avg()

4) Min()

5) Max()

Now let us understand each Aggregate function with a example:

Id	Name	Salary
1	A	80
2	B	40
3	C	60
4	D	70
5	E	60
6	F	Null

Count():

Count(*): Returns total number of records .i.e 6.

Count(salary): Return number of Non Null values over the column salary. i.e 5.

Count(Distinct Salary): Return number of distinct Non Null values over the column salary .i.e 4

Sum():

sum(salary): Sum all Non Null values of Column salary i.e., 310

sum(Distinct salary): Sum of all distinct Non-Null values i.e., 250.

Avg():

Avg(salary) = Sum(salary) / count(salary) = 310/5

Avg(Distinct salary) = sum(Distinct salary) / Count(Distinct Salary) = 250/4

Min():

Min(salary): Minimum value in the salary column except NULL i.e., 40.

Max(salary): Maximum value in the salary i.e., 80.

2.8.9. Insert, Update and Delete Operations in SQL

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.There are two basic syntaxes of the INSERT INTO statement which are shown below.

INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The **SQL INSERT INTO** syntax will be as follows −

INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);

Example

The following statements would create six records in the CUSTOMERS table.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

All the above statements would produce the following records in the CUSTOMERS table as shown below.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Populate one table using another table

You can populate the data into a table through the select statement over another table; provided the other table has a set of fields, which are required to populate the first table.

Here is the syntax −

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
```

```
FROM second_table_name
[WHERE condition];
```

SQL UPDATE

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows −

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

Example

Consider the CUSTOMERS table having the following records −

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, the CUSTOMERS table would have the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	Pune	4500.00
7	Muffy	24	Indore	10000.00

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Pune	1000.00
2	Khilan	25	Pune	1000.00
3	kaushik	23	Pune	1000.00
4	Chaitali	25	Pune	1000.00
5	Hardik	27	Pune	1000.00
6	Komal	22	Pune	1000.00
7	Muffy	24	Pune	1000.00

SQL DELETE QUERY

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows −

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

Now, the CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
```

```
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan  | 25 | Delhi     |  1500.00 |
| 3 | kaushik |  23 | Kota     |  2000.00 |
| 4 | Chaitali |  25 | Mumbai   |  6500.00 |
| 5 | Hardik  |  27 | Bhopal   |  8500.00 |
| 7 | Muffy   |  24 | Indore   | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to DELETE all the records from the CUSTOMERS table, you do not need to use the WHERE clause and the DELETE query would be as follows −

```
SQL> DELETE FROM CUSTOMERS;
```

Now, the CUSTOMERS table would not have any record.

2.8.10. Joins in SQL

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables −

CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan  | 25 | Delhi     |  1500.00 |
| 3 | kaushik |  23 | Kota     |  2000.00 |
| 4 | Chaitali |  25 | Mumbai   |  6500.00 |
| 5 | Hardik  |  27 | Bhopal   |  8500.00 |
| 6 | Komal   |  22 | MP       |  4500.00 |
| 7 | Muffy   |  24 | Indore   | 10000.00 |
```

```
+----+----------+-----+-----------+----------+
```

ORDERS Table

```
+-----+---------------------+------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+------------+--------+
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
| 101 | 2009-11-20 00:00:00 |          2 |   1560 |
| 103 | 2008-05-20 00:00:00 |          4 |   2060 |
+-----+---------------------+------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
   FROM CUSTOMERS, ORDERS
   WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.
- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

2.8.11. SET Operations in SQL

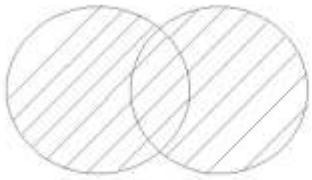
SQL supports few Set operations which can be performed on the table data. These are used to get meaningful results from data stored in the table, under different special conditions.

In this tutorial, we will cover 4 different types of SET operations, along with example:

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS

UNION Operation

UNION is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.



Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

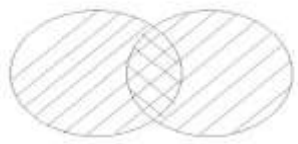
```
SELECT * FROM First

UNION

SELECT * FROM Second;
```

The result set table will look like,

ID	NAME
1	abhi
2	Adam
3	Chester

UNION ALL

This operation is similar to Union. But it also shows the duplicate rows.



Example of Union All

The **First** table,

ID	NAME
1	Abhi
2	Adam

The **Second** table,

ID	**NAME**
2	Adam
3	Chester

Union All query will be like,

```
SELECT * FROM First

UNION ALL

SELECT * FROM Second;
```

The resultset table will look like,

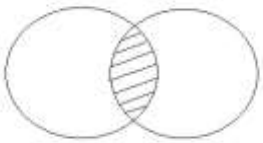
ID	**NAME**
1	Abhi

2	Adam
2	Adam
3	Chester

INTERSECT

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same.

NOTE: MySQL does not support INTERSECT operator.



Example of Intersect

The **First** table,

ID	NAME
1	Abhi
2	Adam

The **Second** table,

ID	NAME
2	Adam
3	Chester

Intersect query will be,

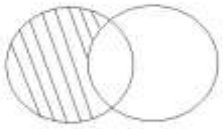
```
SELECT * FROM First

INTERSECT
```

```
SELECT * FROM Second;
```

The result set table will look like

ID	NAME
2	Adam

MINUS: The Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.



Example of Minus

The **First** table,

ID	**NAME**
1	Abhi
2	Adam

The **Second** table,

ID	**NAME**
2	Adam
3	Chester

Minus query will be,

```
SELECT * FROM First

MINUS

SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	Abhi

2.8.12. SQL Cursors

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes

S.No	Attribute & Description
1	**%FOUND**

	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
6 customers selected
PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
```

```
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
  c_id customers.id%type;
  c_name customer.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
```

```
/
```

When the above code is executed at the SQL prompt, it produces the following result −

1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

2.8.13. Triggers in SQL

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
```

```
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Trigger created.

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

Old salary: 1500

New salary: 2000

Salary difference: 500

2.8.14. PL/SQL Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named '**greetings**' can be called with the EXECUTE keyword as −

EXECUTE greetings;

The above call will display −

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

```
BEGIN
  greetings;
END;
/
```

The above call will display −

Hello World

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
```

```
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
```

```
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Square of (23): 529

PL/SQL procedure successfully completed.

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following −

findMin(x => a, y => b, z => c, m => d);

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

IMPORTANT QUESTIONS

1. **Q.1** Consider the relational database:

   EMPLOYEE (Name,Ssn, Sex, Superssn, Dno)

   DEPARTMENT (Dnumber, Dname, Dlocation)

   DEPT_LOCATION (Dnumber,Dlocation)

   PROJECT (Pname, Pnumber, Plocation, Dnum)

   WORKS_ON (Essn, Pno, Hours)

   DEPENDENT (ESSN, Dependent_name, Sex, Bdate, Relationship)

   Write the SQL expressions for the following queries

   i.Retrieve the name and address of all employees who work for research department.

   ii.Retrieve the name of employees who have no dependents

   iii.List the name of employees who have atleast one dependent.

2. Discuss the entity integrity and referential integrity constraints. Why each is considered important**?**
3. What is union compatibility? Why do union, intersection and difference operations require that the relations on which they applied be union compatible?
4. Explain tuple relational calculus and domain relational calculus by taking suitable example**.**
5. Write Relational Algebraic Expressions for the following:

   i. Department (Deptid, Dname, HOD)

   ii. Student (Rollno, Name, Year, Deptid)

   iii. Project (Pid, Pname, Duration)

   iv. Student_PROJECT (Rollno, Pid)

   Display roll numbers of those students who have not been assigned any project.

   Display student names along with their department names.

   Display names of those students who have been assigned project of less than 15 days duration**.**

6. Consider the following relational schema:

   SUPPLIER (SID, SNAME, SADDRESS)

   PARTS (PID, PNAME, COLOR)

   CATALOG (SID, PID, COST),

The catalog relation lists the prices charged for parts by suppliers:

Express the following statements into relational algebra expressions

Find the name of a supplier who supplies yellow parts.

Find the names of the suppliers who supply red part that costs less than 100 dollars.

Find the names of the suppliers who supply a red part that costs more than 100 dollars and a green part that costs less than 100 dollars.

Find the name of the supplier and id who supply green part and address is Noida.

7. Define Relational calculus and their types. Explain it with suitable examples.

8. Discuss Join and types with suitable example.

9. What do you mean by referential integrity? Explain the concept of foreign key with a suitable example?

10. What are the differences in Cartesian product and natural join operations? Explain with a suitable example.

MCQ QUESTIONS

1. Which one of the following is used to define the structure of the relation, deleting relations and relating schemas?

a) DML(Data Manipulation Langauge)

b) DDL(Data Definition Langauge)

c) Query

d) Relational Schema

2. **CREATE TABLE** employee (name **VARCHAR**, id **INTEGER**)

What type of statement is this?

a) DML

b) DDL

c) View

d) Integrity constraint

3. The basic data type char(n) is a_____length character string and varchar(n) is _____ length character.

a) Fixed, equal

b) Equal, variable

c) Fixed, variable

d) Variable, equal

4. To remove a relation from an SQL database, we use the_____command.

a) Delete

b) Purge

c) Remove

d) Drop table

5. Updates that violate_____are disallowed.

a) Integrity constraints

b) Transaction control

c) Authorization

d) DDL constraints

6. The_____clause allows us to select only those rows in the result relation of the _____clause that satisfy a specified predicate.

a) Where, from

b) From, select

c) Select, from

d) From, where

7. The query given below will not give an error. Which one of the following has to be replaced to get the desired output?

```
SELECT ID, name, dept name, salary * 1.1
WHERE instructor;
```

a) Salary*1.1

b) ID

c) Where

d) Instructor

8. Which of the following statements contains an error?

a) Select * from emp where empid = 10003;

b) Select empid from emp where empid = 10006;

c) Select empid from emp;

d) Select empid where empid = 1009 and lastname = „GELLER";

```
9. SELECT emp_name
FROM department
```

Which one of the following has to be added into the blank to select the dept_name which has Computer Science as its ending string?

a) %

b) _

c) ||

d) $

```
10. SELECT *
FROM instructor
```

To display the salary from greater to smaller and name in ascending order which of the following options should be used?

a) Ascending, Descending

b) Asc, Desc

c) Desc, Asc

d) Descending, Ascending

11. In SQL the spaces at the end of the string are removed by_____function.

a) Upper

b) String

c) Trim

d) Lower

12. The union operation automatically_____unlike the select clause.

a) Adds tuples

b) Eliminates unique tuples

c) Adds common tuples

d) Eliminates duplicate

13. For like predicate which of the following is true.

```
i) % matches zero OF more characters.
ii) _ matches exactly one CHARACTER.
```

a) i-only

b) ii-only

c) i & ii

d) None of the mentioned

14. _____clause is an additional filter that is applied to the result.

a) Select

b) Group-by

c) Having

d) Order by

15. If the attribute phone number is included in the relation all the values need not be entered into the phone number column. This type of entry is given as

a) 0

b) –

c) Null

d) Empty space

```
16. SELECT name
FROM instructor
WHERE salary IS NOT NULL;
```

a) Tuples with null value

b) Tuples with no null values

c) Tuples with any salary

d) All of the mentioned

17. The primary key must be

a) Unique

b) Not null

c) Both Unique and Not null

d) Either Unique or Not null

18. The result of_____unknown is unknown.

a) Xor

b) Or

c) And

d) Not

19. Aggregate functions are functions that take a_____as input and return a single value.

a) Collection of values

b) Single value

c) Aggregate value

d) Both Collection of values & Single value

20.
```
SELECT COUNT (_____ID)
FROM teaches
```

If we do want to eliminate duplicates, we use the keyword_____in the aggregate expression.

a) Distinct

b) Count

c) Avg

d) Primary key

21. A Boolean data type that can take values true, false, and_____

a) 1

b) 0

c) Null

d) Unknown

22. The phrase “greater than at least one” is represented in SQL by _____

a) < all

b) < some

c) > all

d) > some

23. We can test for the nonexistence of tuples in a subquery by using the _____ construct.

a) Not exist

b) Not exists

c) Exists

d) Exist

24. Aggregate functions can be used in the select list or the__clause of a select statement or subquery. They cannot be used in a_____clause.

a) Where, having

b) Having, where

c) Group by, having

d) Group by, where

25. Subqueries cannot:

a) Use group by or group functions

b) Retrieve data from a table different from the one in the outer query

c) Join tables

d) Appear in select, update, delete, insert statements.

26. The EXISTS keyword will be true if:

a) Any row in the subquery meets the condition only

b) All rows in the subquery fail the condition only

c) Both of these two conditions are met

d) Neither of these two conditions is met

27. Delete from r where P;

The above command

a) Deletes a particular tuple from the relation

b) Deletes the relation

c) Clears all entries from the relation

d) All of the mentioned

28.
```
UPDATE instructor
        salary= salary * 1.05;
```

Fill in with correct keyword to update the instructor relation.

a) Where

b) Set

c) In

d) Select

29. _____are useful in SQL update statements, where they can be used in the set clause.

a) Multiple queries

b) Sub queries

c) Update

d) Scalar subqueries

30. The problem of ordering the update in multiple updates is avoided using

a) Set

b) Where

c) Case

d) When

31. Which of the join operations do not preserve non matched tuples?

a) Left outer join

b) Right outer join

c) Inner join

d) Natural join

32. How many tables may be included with a join?

a) One

b) Two

c) Three

d) All of the mentioned

33. In SQL the statement select * from R, S is equivalent to

a) Select * from R natural join S

b) Select * from R cross join S

c) Select * from R union join S

d) Select * from R inner join S

34.
```
CREATE VIEW faculty AS
    SELECT ID, name, dept name
    FROM instructor;
```

Find the error in this query.

a) Instructor

b) Select

c) View …as

d) None of the mentioned

35. To include integrity constraint in an existing relation use :

a) Create table

b) Modify table

c) Alter table

d) Drop table

36. Foreign key is the one in which the_____of one relation is referenced in another relation.

a) Foreign key

b) Primary key

c) References

d) Check constraint

37. Data integrity constraints are used to:

a) Control who is allowed access to the data

b) Ensure that duplicate records are not entered into the table

c) Improve the quality of data entered for a specific property (i.e., table column)

d) Prevent users from changing the values stored in the table

38. Which of the following is used to input the entry and give the result in a variable in a procedure?

a) Put and get

b) Get and put

c) Out and In

d) In and out

39. The format for compound statement is

a) Begin ……. end

b) Begin atomic……. end

c) Begin ……. repeat

d) Both Begin ……. end and Begin atomic……. end

40. Triggers are supported in

a) Delete

b) Update

c) Views

d) All of the mentioned

Answer Key

1	B	11	C	21	D	31	C
2	B	12	D	22	B	32	D
3	C	13	A	23	B	33	B
4	D	14	C	24	B	34	D
5	A	15	C	25	C	35	C
6	A	16	B	26	A	36	B
7	C	17	C	27	A	37	C
8	D	18	D	28	B	38	D
9	A	19	A	29	D	39	D
10	C	20	A	30	C	40	C

UNIT-III

DATA BASE DESIGN & NORMALIZATION

When designing a DB, what is the best way to represent the real-world entities involved?

- common sense of the designer plays a big part
- here we formalize a measure of "goodness" of groupings of attributes in relations

Two levels to measure goodness:

logical level - how users interpret relation schemas - how well user understands meaning of data tuples

manipulation (storage) level - how tuples in base relations are stored and updated

Factors in designing database schemas:

1) Semantics of attributes
2) Reduce redundancy
3) Reduce null values
4) Eliminate spurious tuples

1) Semantics: meaning of the attributes in a tuple - how they relate to each other ex:

TOY database - the meaning of each relation schema is pretty straightforward

look at the TOY relation: TOY_NUM is a key (primary key) NAME

- the name of the toy

MAN_ID - a foreign key representing an implicit relationship between the TOY db and the MANUFACTURER db

- make semantics clear and easy to explain - one relation describes only one real world entity

2) Reduce redundancy:

ex: imagine that the TOY relation and the MANUFACTURER relation were all put into one relation:

TOY_MAN(TOY_NUM, NAME, MAN_NAME, ADDRESS, PHONE, SALES_CONTACT, MSRP, AGE_GROUP, NUM_IN_STOCK, SOLD_YTD)

- while this would reduce the number of relations, the number of tuples would increase because for every toy made by the same manufacturer, all of the manufacturer's data would have to be stored

TOY_MAN:

011 FARM HOUSE FISCHER PRICE 111 MAIN ST... 29.95 ...

221 EXERSAUCER FISCHER PRICE 111 MAIN ST... 45.00 .Another problem with storing data in this way is the existence of update anomalies (3 types):

- Insertion anomalies
  - every time a new tuple is added, all manufacturer data must be inserted - and must be consistent with the data that already exists for the same manufacturer in the other relations
  - how to insert a new manufacturer for which the catalog does not yet sell any toys - null values for toy info - but toy_num is a key and can't be null

- Deletion Anomalies
  - if the last toy made by a manufacturer is deleted, then the manufacturer data is lost

- Modification Anomalies
  - if value of data in a manufacturer changes, has to be changed in all tuples involving that manufacturer

Avoid redundancy that can cause these anomalies - if any must exist to accomodate particular frequently used queries, document them so that they can be maintained

3) Null values: Avoid including attributes that are likely to have null values

- nulls waste space if used too often
- multiple interpretations of null
  - does not apply
  - unknown

- known but missing - not yet recorded

If nulls must be used, make sure they apply only in rare cases.

4) Spurious tuples

- represent wrong information
- occur when joining two or more badly designed relations

ex: Take the TOY_MAN db from above and project it onto the following:

TOY_ADDR(NAME, ADDRESS) (primary key is (NAME,ADDRESS))

TOY_MAN1(TOY_NUM,MAN_NAME,ADDRESS, PHONE, SALES_CONTACT, MSRP, AGE_GROUP, NUM_IN_STOCK, SOLD_YTD)

- when these are joined together using ADDRESS, we end up with extraneous incorrect tuples - specifically:

011 EXERSAUCER FISCHER PRICE 111 MAIN ST... 29.95 ...

3.1 Functional Dependencies

Functional Dependency (FD) is a constraint that determines the relation of one attribute to another attribute in a Database Management System. Functional Dependency helps to maintain the quality of data in the database. It plays an important role to find the difference between good and bad database design.

A functional dependency is denoted by an arrow "→". The functional dependency of X on Y is represented by X → Y. Let's understand Functional Dependency in DBMS with example. **Table 3.1**

Employee number	Employee Name	Salary	City
1	Ramesh	50000	Delhi
2	Mahendra	38000	Mumbai
3	Surbhi	25000	Pune

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

Inference Rules of Functional Dependencies

We denote by F the set of functional dependencies that are specified on relation schema R

Below are the three most important rules for Functional Dependency in Database:

1. Reflexive rule –. If X is a set of attributes and Y is subset of X, then X holds a value of Y.
2. Augmentation rule: {X -> Y} |= XZ -> YZ.When x -> y holds, and c is attribute set, then ac -> bc also holds. That is adding attributes which do not change the basic dependencies.
3. Transitivity rule: {X -> Y, Y-> Z} |= X -> Z .This rule is very much similar to the transitive rule in algebra if x -> y holds and y -> z holds, then x -> z also holds. X -> y is called as functionally that determines y
4. Decomposition rule: {X -> YZ} |= X -> Y
5. Union rule: {X -> Y, X -> Z} |= X -> YZ
6. Pseudo transitive rule: {X -> Y, WY -> Z} |= WX -> Z

The set of dependencies, which we called the closure of F, can be determined from F by using only inference rules 1 through 3. Inference rules 1 through 3 are known as **Armstrong's inference rules.**

Database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then 1, 2, 3,4,5 and 6 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left hand side of some functional dependency in F and then to determine the set of all attributes that are dependent on X. Thus for each such set of attributes X, we determine the set of attributes that are functionally determined by X based on This called **the closure of X under F.**

Example:

F = {SSN -> {ENAME, BDATE, ADDRESS, DNUMBER},

DNUMBER -> {DNAME, DMGRSSN} }

can infer the following:

SSN -> {DNAME, DMGRSSN}

SSN -> SSN

DNUMBER -> DNAME

from above F

{SSN}+ = {SSN, ENAME, BDATE, ADDRESS, DNUMBER, DNAME, DMGRSSN}

{DNUMBER}+ = {DNUMBER, DNAME, DMGRSSN}

{SSN, DNUMBER} + = {SSN, ENAME, BDATE, ADDRESS, DNUMBER, DNAME, DMGRSSN}

Equivalence of Sets of Functional Dependencies

Two sets of functional dependencies G and F are equivalent if every FD in G can be inferred from F, and every FD in F can be inferred from G. That is two sets of FDs F and G over schema R are equivalent, if $F^+ = G^+$. That is, if every functional dependency of F is in G+ and every functional dependency of G is in F+, then we would say that the sets of functional dependencies F and G are equivalent. (here, F^+ and G^+ means the closure of sets of functional dependencies)

Minimal Sets of Functional Dependencies

A set of functional dependencies F is minimal if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.

2. We cannot replace any dependency X -> A in F with a dependency Y -> A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.

3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

A minimal cover of a set of dependencies, F, is a set of dependencies, G, such that:

- G is equivalent to F ($F^+ = G^+$)
- All FDs in U have the form X→A where A is a single attribute
- It is not possible to make U smaller (while preserving equivalence) by
  Deleting an FD
  Deleting an attribute from an FD (either from LHS or RHS)
- FDs and attributes that can be deleted in this way are called redundant

Computing Minimal Cover

Example: T = {ABH→CK, A→D, C→E, BGH→F, F→AD, E→F, BH→E}

step 1: Make RHS of each FD into a single attribute

- Algorithm: Use the decomposition inference rule for FDs
- Example: F→AD replaced by F→A, F→D ; ABH→ CK by ABH→C, ABH→K

step 2: Eliminate redundant attributes from LHS.

- Algorithm: If FD XB→A ∈ T (where B is a single attribute) and X→A is entailed by T, then B was unnecessary
- Example: Can an attribute be deleted from ABH→C ?
  Compute AB^+_T, AH^+_T, BH^+_T.
  Since C ∈ $(BH)^+_T$, BH→C is entailed by T and A is redundant in ABH→C.

step 3: Delete redundant FDs from T

- Algorithm: If T - {f} entails f, then f is redundant
- If f is X→A then check if A ∈ $X^+_{T-\{f\}}$
- Example: BGH→F is entailed by E→F, BH→E, so it is redundant

Note: Steps 2 and 3 cannot be reversed!!

3.2 Normal Forms

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update & deletion anomaly. It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.

Normalization is used for mainly two purposes,

- Eliminating redundant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

3.2.1 First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values. **Example**: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

Table 3.2

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390

102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

Table 3.3

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	**Lester**	**Bangalore**	**8123450987**

3.2.2 Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as **non-prime attribute**. An attribute that is part of any candidate key is known as **prime attribute**.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

Table 3.4

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non-prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this: **teacher_details table:**

Table 3.5

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

Table 3.6

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

3.2.3 Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

Table 3.7

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.
Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

Table 3.8

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Table 3.9

3.2.4 Boyce Codd normal form (BCNF)

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

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

Table 3.10

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

Functional dependencies in the table above:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys. To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

Table 3.11

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

Table 3.12

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

emp_dept_mapping table:

Table 3.13

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

Functional dependencies:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:

For first table: emp_id

For second table: emp_dept

For third table: {emp_id, emp_dept}

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

3.3 Inclusion Dependency

- Multivalued dependency and join dependency can be used to guide database design although they both are less common than functional dependencies.
- Inclusion dependencies are quite common. They typically show little influence on designing of the database.
- The inclusion dependency is a statement in which some columns of a relation are contained in other columns.
- The example of inclusion dependency is a foreign key. In one relation, the referring relation is contained in the primary key column(s) of the referenced relation.
- Suppose we have two relations R and S which was obtained by translating two entity sets such that every R entity is also an S entity.
- Inclusion dependency would be happen if projecting R on its key attributes yields a relation that is contained in the relation obtained by projecting S on its key attributes.
- In inclusion dependency, we should not split groups of attributes that participate in an inclusion dependency.
- In practice, most inclusion dependencies are key-based that is involved only keys.

3.4 Decomposition

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

Decomposition is of two types:

1. Lossless join Decomposition
2. Lossy Decomposition

3.4.1 Lossless Join Decomposition

- Consider there is a relation R which is decomposed into sub relations R1 , R2 , …. , Rn.
- This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.

- For lossless join decomposition, we always have- R1 ⋈ R2 ⋈ R3 ……. ⋈ Rn = R , where ⋈ is a natural join operator.

Example : Consider the following relation R(A , B , C)-

Table 3.14

A	B	C
1	2	1
2	5	3
3	3	3

Consider this relation is decomposed into two sub relations R1(A , B) and R2(B , C)-

- R(A,B,C)

1. R1(A,B)
2. R2(B,C)

The two sub relations are-

Table 3.15

A	B
1	2
2	5
3	3

R1(A , B)

Table 3.16

B	C
2	1
5	3
3	3

R2(B , C)

Now, let us check whether this decomposition is lossless or not.

For lossless decomposition, we must have-

R1 ⋈ R2 = R

Now, if we perform the natural join (⋈) of the sub relations R1 and R2 , we get-

Table 3.17

A	B	C
1	2	1
2	5	3
3	3	3

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition.

NOTE:

•Lossless join decomposition is also known as non-additive join decomposition.

•This is because the resultant relation after joining the sub relations is same as the decomposed relation.

•No extraneous tuples appear after joining of the sub-relations.

3.4.2 Lossy Join Decomposition:

- Consider there is a relation R which is decomposed into sub relations R1, R2, …. , Rn.
- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.
- The natural join of the sub relations is always found to have some extraneous tuples.
- For lossy join decomposition, we always have- R1 ⋈ R2 ⋈ R3 ……. ⋈ Rn ⊃ R where ⋈ is a natural join operator

Example: Consider that we have table STUDENT with three attribute roll_no , sname and department.

Student:

Table 3.18

Roll_no	Sname	Dept
111	Parimal	COMPUTER
222	Parimal	ELECTRICAL

This relation is decomposed into two relation no_name and name_dept :

No_name

Table 3.19

Roll_no	Sname
111	parimal
222	parimal

name_dept

Table 3.20

Sname	Dept
parimal	COMPUTER
parimal	ELECTRICAL

In lossy decomposition, spurious tuples are generated when a natural join is applied to the relations in the decomposition.

Table 3.21

Roll_no	Sname	Dept
111	parimal	COMPUTER
111	parimal	ELECTRICAL
222	parimal	COMPUTER
222	parimal	ELECTRICAL

The above decomposition is a bad decomposition or Lossy decomposition.

3.5 Multivalued Dependencies and Join Dependencies

A consequence of 1NF: no multivalued attributes

Now have multiple entries to record the multivalued dependency

If you have two MVDs in a relation (several dependents and several projects for an employee), all combinations of project and dependents have to be listed.

In general, in **R(A,B,C)** if each A values has associated with it a set of B values and a set of C values such that the B and C values are independent of each other, then A multi-determines B and A multi-determines C

Multi-valued dependencies occur in pairs.

Example: **JointAppoint(facId, dept, committee)** assuming a faculty member can belong to more than one department and belong to more than one committee

Table must list all combinations of values of department and committee for each facId

MVD Formal definition

Let R be a relation scheme and X and Y are subsets of R,

X ->> Y

holds if for all pairs of tuples s and t in r, such that

s[X] = t[X] there also exist tuples u and v where

s[X] = t[X] = u[X] = v[X]

s[Y] = u[Y]

t[R-XY] = u[R-XY]

s[R-XY] = v[R-XY]

t[Y] = v[Y]

Example:

Course ->> Instructor

Course ->> Text

Table 3.21

Course(Y)	Instructor(X)	Text(R-XY)
Prin of IT	Kruse	Intro to IT
Prin of IT	Wright	Intro to IT
CS1	Thomas	Intro to Java
CS1	Thomas	CS Theory

CS2	Rhodes	Java Data Structures
CS2	Rhodes	Unix
CS2	Kruse	Java Data Structures
CS2	Kruse	Unix

4NF

A relation R is in 4NF if for all MVD in relation is of the form A->>B at least one of the following hold

- A ->> B is a trivial MVD
- A is a superkey

(Course, Instructor) and (Course, Text) is 4NF decomposition

LJ Decomposition for 4NF Relations

The relation schemas S and T form a LJ decomposition for R if
(S*T) ->>(S-T) or (S*T)->>(T-S).

replace MVD for FD in BCNF Algorithm to produce a 4NF decomposition algorithm

Join Dependency

- Join decomposition is a further generalization of Multivalued dependencies.
- If the join of R1 and R2 over C is equal to relation R, then we can say that a join dependency (JD) exists.
- Where R1 and R2 are the decompositions R1(A, B, C) and R2(C, D) of a given relations R (A, B, C, D).
- Alternatively, R1 and R2 are a lossless decomposition of R.
- A JD ⋈ {R1, R2,..., Rn} is said to hold over a relation R if R1, R2,....., Rn is a lossless-join decomposition.
- The *(A, B, C, D), (C, D) will be a JD of R if the join of join's attribute is equal to the relation R.
- Here, *(R1, R2, R3) is used to indicate that relation R1, R2, R3 and so on are a JD of R.

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Dependency Preservation

If f is an FD in F, but f is not in $F_1 \cup F_2$, there are two possibilities:
$f \in (F_1 \cup F_2)^+$
If the constraints in F_1 and F_2 are maintained, f will be maintained automatically.
$f \notin (F_1 \cup F_2)^+$
f can be checked only by first taking the join of r_1 and r_2. This is costly.

Example

Schema (R, F) where

- R = {SSN, Name, Address, Hobby}
- F = {SSN → Name, Address}

can be decomposed into

- R_1 = {SSN, Name, Address} F_1 = {SSN→Name, Address}
- and R_2 = {SSN, Hobby} F_2 = { }

Since $F = F_1 \cup F_2$ the decomposition is dependency preserving

Example

Schema: (ABC; F) with F = {A → B, B→ C, C→ B}

Decomposition:

- (AC, F_1), F_1 = {A→ C} Note: A→ C not in F, but is in F^+

- (BC, F_2), F_2 = {B→ C, C→ B}

A → B ∉ (F_1 ∪ F_2), **but** A → B ∈ (F_1 ∪ F_2)$^+$.

- So F^+ = (F_1 ∪ F_2)$^+$ and thus the decompositions is still dependency preserving

3.6 Alternative Approaches to Database Design

1. We have taken the approach of starting with a single relation scheme and decomposing it.
    - One goal was lossless-join decomposition.
    - For that, we decided we needed to talk about the join of all relations on the decomposed database.
    - If we compute the natural join, we find that all tuples referring to loan number 58 disappear.
    - We call the tuples that disappear when the join is computed **dangling tuples**.
    - Formally, if are a set of relations, a tuple of relation is a dangling tuple if is not in the relation
    - Dangling tuples may occur in practical applications.
    - They represent incomplete information.
    - The relation is called a **universal relation** since it involves all the attributes in the universe defined by .
    - The only way to write a universal relation for our example is include null values.
    - Because of the difficulty in managing null values, it may be desirable to view the decomposed relations as representing the database rather than the universal relation.
    - We still might need null values if we tried to enter a loan number without a customer name, branch name, or amount.
    - In this case, a particular decomposition defines a restricted form of incomplete information that is acceptable in our database.

2. The normal forms we have defined generate good database design from the point of view of representation of incomplete information.
    - We need a loan number to represent any information in our example.
    - We do not want to store data for which the key attributes are unknown.
    - The normal forms we have defined do not allow us to do this unless we use null values.
    - Thus our normal forms allow representation of acceptable incomplete information via dangling tuples while prohibiting the storage of undesirable incomplete information.
3. Another point in our method of design is that attribute names must be unique in the universal relation.
    - We call this the **unique role assumption**.
    - If we defined the relations expressions like branch-loan loan-customer are possible but meaningless.
    - In SQL, there is no natural join operation, and so references to names are disambiguated by prefixing relation names.
    - In this case, non-uniqueness might be both convenient and allowed.
    - The unique role assumption is generally preferable, and if it is not made, special care must be taken when constructing a normalized design.

Important Questions

1. Consider the universal relation R= {A,B,C,D,E,F,G,H,I,J} and a set of functional dependencies F= { {AB→ C}, { A → DE}, { B→ F}, {F→ GH}, {D→ IJ}}.
   What is the key for R?
   Decompose R into 2NF, then 3NF relations.
2. Consider the relation R: (A, B, C, D, E, F, G, H) with following FDs: F :{ AC → G, D → EG, BC →D, CG →BD, ACD →B, CE →AG}, Find the canonical cover of F.
3. Explain dependency preserving decomposition. Given two sets F1 and F2 of FDs for a relation R (A, B, C, D, E). F1:A→ B, AB →C, D →AC, D →E, F2: A →BC, D →AE. Are F1 and F2 equivalent? Explain.
4. Suppose we have the database for an investment firm consisting of the following attributes: B (broker), O(office of the broker), I(investor), S(Stock), Q(quantity of the stock owned by an investor) and D(dividend paid by a stock), with the functional dependencies:
   a. S→ D, I→ B, IS→ Q, B→>O

   Find the key of the relational schema R=BOSQID.
   How many keys does the R have? Justify.
   Find the lossless join decomposition of R into Boyce Codd Normal Form.

5. **C**onsider the relation R(A,B,C,D,E,F,G,H,I,J) and the set of FD's
   a. F= {AB→C, A→DE, B→F, F→GH, D→IJ},

   What is the key in the given relation? Decompose R into 2NF and 3NF.
6. Is every 3rd NF is in BCNF, if not why? If so, give any example of 3rd NF but not in BCNF.
7. What is Canonical Cover? Determine canonical cover of an FD set
   a. {A→BCD, B→CDA, C→ABD}
8. What is a Functional Dependency? Describe Inference rules for Functional Dependencies (FDs). If we have a initial set of attributes (A,G) and given FD are A→ B, A→C, CG→ H,CG→ I,B→ H, Find the relation.
9. Explain Lossless join Decomposition with example. Why is it desirable to have a decomposition to be dependency preserving?
10. Define Normalization. List the types of Normal form. Describe multivalued dependency with a suitable example.

Multiple choice questions

1. In the normal form, a composite attribute is converted to individual attributes.

a) First

b) Second

c) Third

d) Fourth

2. A table on the many side of a one to many or many to many relationships must:

a) Be in Second Normal Form (2NF)

b) Be in Third Normal Form (3NF)

c) Have a single attribute key d) Have a composite key

3. Tables in second normal form (2NF):

a) Eliminate all hidden dependencies

b) Eliminate the possibility of a insertion anomalies c) Have a composite key

d) Have all non key fields depend on the whole primary key

4. Which-one ofthe following statements about normal forms is FALSE?

a) BCNF is stricter than 3 NF

b) Lossless, dependency -preserving decomposition into 3 NF is always possible

c) Loss less, dependency – preserving decomposition into BCNF is always possible d) Any relation with two attributes is BCNF

5. Functional Dependencies are the types of constraints that are based on

a) Key

b) Key revisited

c) Superset key

d) None of the mentioned

6. Which is a bottom-up approach to database design that design by examining the relationship between attributes:

a) Functional dependency

b) Database modeling

c) Normalization

d) Decomposition

7. Which forms are based on the concept of functional dependency:

a) 1NF

b) 2NF

c) 3NF

d) 4NF

8. Empdt1(empcode, name, street, city, state, pincode).

For any pincode, there is only one city and state. Also, for given street, city and state, there is just one pincode. In normalization terms, empdt1 is a relation in

a) 1 NF only

b) 2 NF and hence also in 1 NF

c) 3NF and hence also in 2NF and 1NF

d) BCNF and hence also in 3NF, 2NF and 1NF

9. We can use the following three rules to find logically implied functional dependencies. This collection of rules is called

a) Axioms

b) Armstrong's axioms

c) Armstrong d) Closure

10. Which of the following is not Armstrong's Axiom?

a) Reflexivity rule

b) Transitivity rule

c) Pseudotransitivity rule

d) Augmentation rule

11 The relation employee(ID,name,street,Credit,street,city,salary) is decomposed into

employee1 (ID, name)

employee2 (name, street, city, salary)

This type of decomposition is called

a) Lossless decomposition

b) Lossless-join decomposition

c) All of the mentioned

d) None of the mentioned

12. The union operation automatically unlike the select clause.

a) Adds tuples

b) Eliminates unique tuples

c) Adds common tuples

d) Eliminates duplicate

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

A->BC A->B

This can be combined as

a) A->BC

b) A->B

c) B->C

d) None of the mentioned

14. Consider a relation R(A,B,C,D,E) with the following functional dependencies:

ABC -> DE and

D -> AB

The number of superkeys of R is:

a) 2

b) 7

c) 10

d) 12

15. Suppose relation R(A,B,C,D,E) has the following functional dependencies:

A -> B B -> C BC -> A A -> D

E -> A

D -> E

Which of the following is not a key?

a) A

b) E

c) B, C

d) D

16. A relation is in if an attribute of a composite key is dependent on an attribute of other composite key.

a) 2NF

b) 3NF

c) BCNF

d) 1NF

17. What are the desirable properties of a decomposition a) Partition constraint

b) Dependency preservation

c) Redundancy

d) Security

18. R (A,B,C,D) is a relation. Which of the following does not have a lossless join dependency preserving BCNF decomposition?

a) A->B, B->CD

b) A->B, B->C, C->D

c) AB->C, C->AD

d) A->BCD

19. The functional dependency can be tested easily on the materialized view, using the constraints .a) Primary key

b) Null

c) Unique

d) Both Null and Unique

20. Which normal form is considered adequate for normal relational database design?

a) 2NF

b) 5NF

c) 4NF

d) 3NF

21. A table has fields F1, F2, F3, F4, and F5, with the following functional dependencies:

F1->F3

F2->F4 (F1,F2)->F5

in terms of normalization, this table is in

a) 1NF

b) 2NF

c) 3NF

d) None of the mentioned

22. Let R(A,B,C,D,E,P,G) be a relational schema in which the following FDs are known to hold:

AB->CD DE->P

C->E

P->C B->G

The relation schema R is a) in BCNF

b) in 3NF, but not in BCNF

c) in 2NF, but not in 3NF

d) not in 2NF

23. The main task carried out in the is to remove repeating attributes to separate tables.

a) First Normal Form

b) Second Normal Form

c) Third Normal Form

d) Fourth Normal Form

24. If a multivalued dependency holds and is not implied by the corresponding functional dependency, it usually arises from one of the following sources.

a) A many-to-many relationship set

b) A multivalued attribute of an entity set

c) A one-to-many relationship set

d) Both A many-to-many relationship set and A multivalued attribute of an entity set

25. In which of the following, a separate schema is created consisting of that attribute and the primary key of the entity set.

a) A many-to-many relationship set

b) A multivalued attribute of an entity set

c) A one-to-many relationship set

d) None of the mentioned

26. Fifth Normal form is concerned with a) Functional dependency

b) Multivalued dependency

c) Join dependency

d) Domain-key

27. In 2NF

a) No functional dependencies (FDs) exist

b) No multivalued dependencies (MVDs) exist

c) No partial FDs exist

d) No partial MVDs exist

28. What is the best way to represent the attributes in a large database?

a) Relational-and

b) Concatenation

c) Dot representation

d) All of the mentioned

29. Consider the schema R(S,T,U,V) and the dependencies S→T, T→U, U→V, V→S. Let R= {R1,R2} such that R1∩R2=Φ. Then the decomposition is :

A. not in 2NF

B. in 2NF but not in 3NF

C. in 3NF but not in 2NF

D. in both 2NF and 3NF

30. R(A,B,C,D) is a relation, Which of the following does not have a lossless join dependency preserving BCNF decomposition

A. A->B, B->CD

B. A->B, B->C,C->D

C. AB->C, C->AD

D. A->BCD

Answer Key

1	A	11	D	21	A
2	D	12	D	22	D
3	A	13	A	23	A
4	C	14	C	24	D
5	A	15	C	25	B
6	C	16	B	26	C
7	C	17	B	27	C
8	B	18	D	28	B
9	B	19	D	29	B
10	C	20	D	30	C

UNIT-IV

TRANSACTION PROCESSING CONCEPT

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations.

Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited, but the savings account were not credited.

Collections of operations that form a single logical unit of work are called transactions. A database system must ensure proper execution of transactions despite failures either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total money might see the checking-account balance before it is debited by the fundstransfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

4.1 Transaction System

A transaction is a program including a collection of database operations, executed as a logical unit of data processing. The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data. It is an atomic process that is either performed into completion entirely or is not performed at all. A transaction involving only data retrieval without any data update is called read-only transaction.

Each high level operation can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks −

- **read_item()** − reads data item from storage to main memory.
- **modify_item()** − change value of item in the main memory.
- **write_item()** − write the modified value from main memory to storage.

Database access is restricted to read_item() and write_item() operations. Likewise, for all transactions, read and write forms the basic database operations.

Transaction Operations

The low level operations performed in a transaction are −

- **begin_transaction** − A marker that specifies start of transaction execution.
- **read_item or write_item** − Database operations that may be interleaved with main memory operations as a part of transaction.
- **end_transaction** − A marker that specifies end of transaction.
- **commit** − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
- **rollback** − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

Transaction States

A transaction may go through a subset of five states, active, partially committed, committed, failed and aborted.

- **Active** − The initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations.
- **Partially Committed** − The transaction enters this state after the last statement of the transaction has been executed.
- **Committed** − The transaction enters this state after successful completion of the transaction and system checks have issued commit signal.
- **Failed** − The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail.
- **Aborted** − This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began.

The following state transition diagram depicts the states in the transaction and the low level transaction operations that causes change in states.

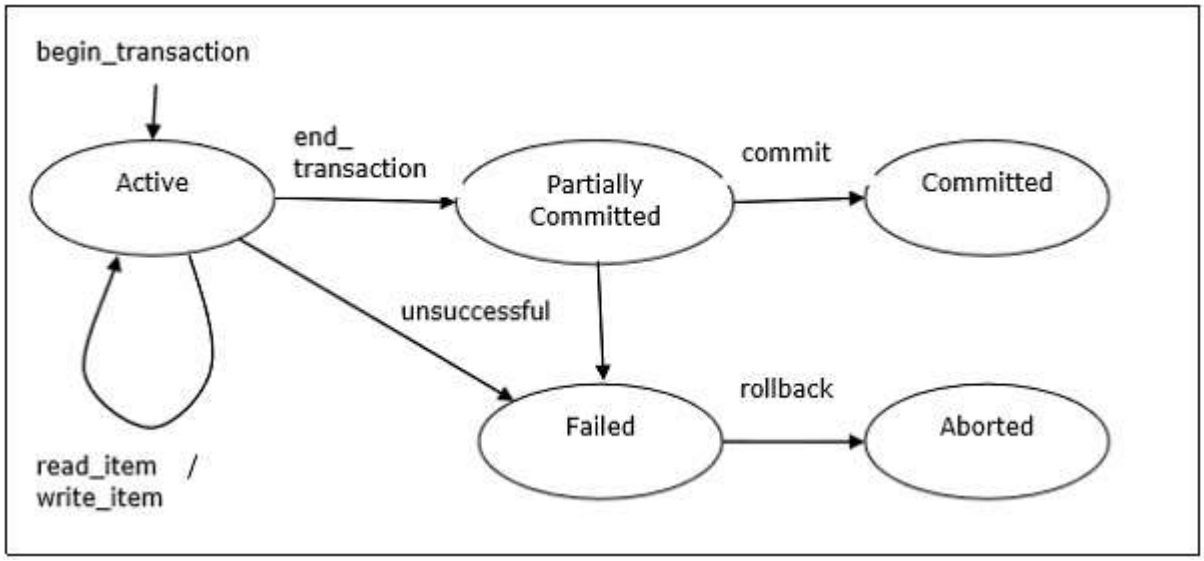


Figure 4.1: Transaction State Diagram

Desirable Properties of Transactions

Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** − This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.
- **Consistency** − A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database.
- **Isolation** − A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.
- **Durability** − If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those

accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.

Transactions access data using two operations:

- read(X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- write(X), which transfers the data item X from the the local buffer of the transaction that executed the write back to the database.

In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored in memory and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. We shall return to this subject in Recovery System.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as

Ti: read(A);

A := A − 50;

write(A);

read(B);

B := B + 50;

write(B).

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

- Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints.
- Atomicity: Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti, a

failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, wenote that the sum A + B is no longer preserved. Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such astate an inconsistent state. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account A is $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are. The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the transaction-management component.

- Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the recovery-management component. The transaction-management component and the recovery-management component are closely related.

- Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially that is, one after the other. However, concurrent execution of transactions provides significant performance benefits . Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

We discuss the problems caused by concurrently executing transactions . The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to astate that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation . Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component, whichwe discuss later, in Concurrency Control.

Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- Improved throughput and resource utilization. Atransaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

Reduced waiting time. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes. We study concurrency-control schemes in Concurrency Control; for now, we focus on the concept of correct concurrent execution.

Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as

T1: read(A);

A := A − 50;

write(A);

read(B);

B := B + 50;

write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as

T2: read(A);

temp := A * 0.1;

A := A − temp;

write(A);

read(B);

B := B + temp;

write(B).

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears in Figure 4.2 . In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts A and B that is, the sum A + B is preserved after the execution of both transactions.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Figure 4.2 Schedule 1—a serial schedule in which T1 is followed by T2.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure 4.3. Again, as expected, the sum A + B is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

The execution sequences just described are called schedules. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction T1, the instruction write(A) must appear before the instruction read(B), in any valid schedule. In the following discussion, we shall refer to the first execution sequence (T1 followed by T2) as schedule 1, and to the second execution sequence (T2 followed by T1) as schedule 2. These schedules are serial: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist n! different valid serial schedules.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Figure 4.3 Schedule 2—a serial schedule in which T2 is followed by T1.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction

for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of n transactions is much larger than n!.

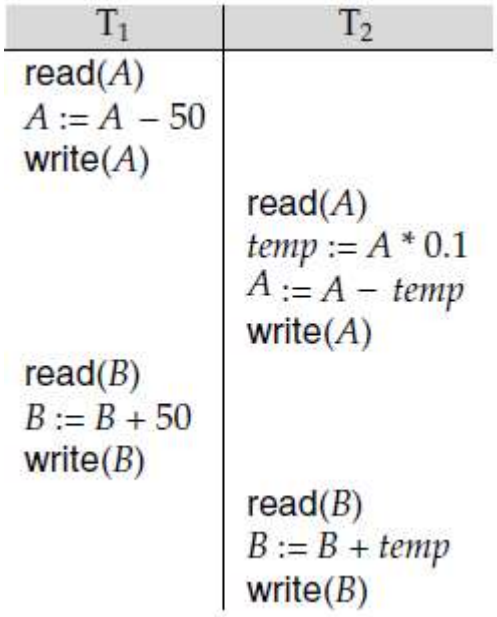
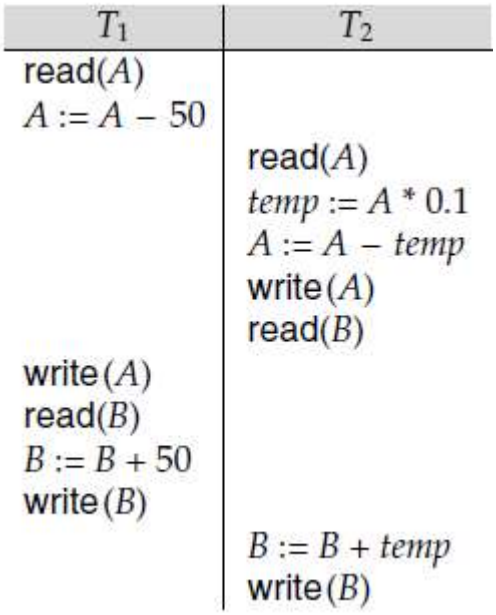


Figure 4.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule.

4.2 Serializability of Schedules

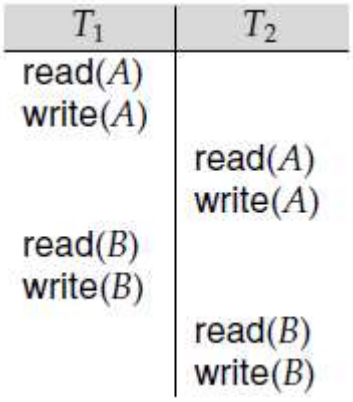
The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not



T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Figure 4.5 Schedule 4—a concurrent schedule.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: read and write. We thus assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules.



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 4.6 Schedule 3 —showing only the read and write instructions.

4.4 Conflict Serializability & View Serializability

Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions Ti and Tj , respectively ($i \neq j$). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.
2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus, the order of Ii and Ij matters.
3. Ii = write(Q), Ij = read(Q). The order of Ii and Ij matters for reasons similar to those of the previous case.
4. Ii = write(Q), Ij = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both Ii and Ij are read instructions does the relative order of their execution not matter. We say that Ii and Ij conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3, in Figure The write(A) instruction of T1 conflicts with the read(A) instruction of T2.However, the write(A) instruction of T2 does not conflict with the read(B) instruction of T1, because the two instructions access different data items.

Let Ii and Ij be consecutive instructions of a schedule S. If Ii and Ij are instructions of different transactions and Ii and Ij do not conflict, then we can swap the order of Ii and Ij to produce a new schedule S'. We expect S to be equivalent to S', since all instructions appear in the same order in both schedules except for Ii and Ij, whose order does not matter.

Since the write(A) instruction of T2 in schedule 3 of Figure. does not conflict with the read(B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.We continue to swap non conflicting instructions:

- Swap the read(B) instruction of T1 with the read(A) instruction of T2.
- Swap the write(B) instruction of T1 with the write(A) instruction of T2.
- Swap the write(B) instruction of T1 with the read(A) instruction of T2.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure 4.7 Schedule 6—schedule 5 after swapping of a pair of instructions.

The final result of these swaps, schedule 6 of Figure 4.8 is a serial schedule. Thus, we have shown that schedule 5 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent. In our previous examples, schedule 1 is not conflict equivalent to schedule 2.However, schedule 1 is conflict equivalent to schedule 3, because the read(B) and write(B) instruction of T1 can be swapped with the read(A) and write(A) instruction of T2.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of Figure 4.8,it consists of only the significant operations (that is, the read and write) of transactions T3 and T4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <T3,T4> or the serial schedule <T4,T3>.

It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction T5, which transfers $10 from account B to account A. Let schedule 8 be as defined in Figure 4.9. We claim that schedule 8 is not conflict equivalent to the serial schedule <T1,T5>, since, in schedule 8, the write(B) instruction of T5 conflicts with the read(B) instruction of T1.

Thus, we cannot move all the instructions of T1 before those of T5 by swapping consecutive nonconflicting instructions. However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule <T1,T5> are the same $960 and $2040, respectively.

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 4.7 Schedule 6—a serial schedule that is equivalent to schedule 3.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figure 4.8 Schedule 7.

We can see from this example that there are less stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule <T1,T5>, it must analyze the computation performed by T1 and T5, rather than just the read and write operations. In general, such analysis is hard to implement and is computationally expensive. However, there are other definitions of schedule equivalence based purely on the read and write operations.

View Serializability

In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

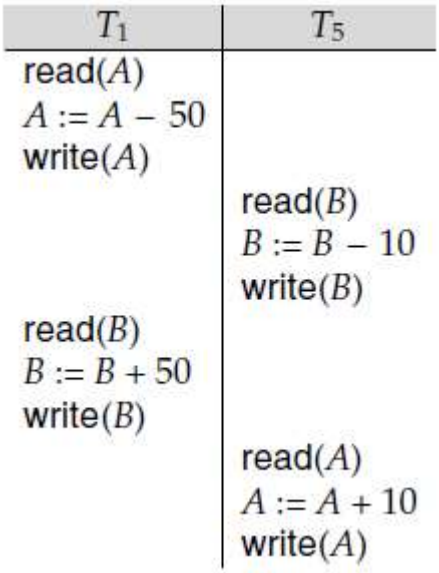


Figure 4.8 Schedule 8.

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be view equivalent if three conditions are met:

1. For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S', also read the initial value of Q.
2. For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj , then the read(Q) operation of transaction Ti must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction Tj .
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is view serializable if it is view equivalent to a serial schedule. As an illustration, suppose that we augment schedule 7 with transaction T6, and obtain schedule 9 in Figure 4.9. Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule <T3, T4, T6>, since the one read(Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions T4 and T6 perform write(Q) operations without having performed a read(Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serializable schedule that is not conflict serializable.

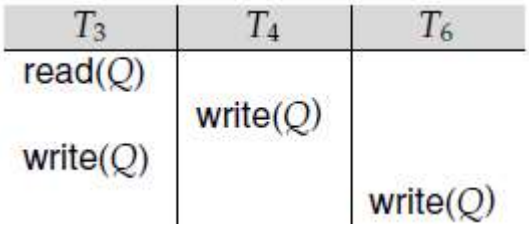


Figure 4.9 Schedule 9—a view-serializable schedule.

4.4 Testing for Serializability

When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable. To do that, we must first understand how to determine, given a particular schedule S, whether the schedule is serializable. We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule S. We construct a directed graph, called a precedence graph, from S. This graph consists of a pair G = (V, E), where V is a set

of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges Ti →Tj for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q).
2. Ti executes read(Q) before Tj executes write(Q).
3. Ti executes write(Q) before Tj executes write(Q).

Figure 4.10 Precedence graph for (a) schedule 1 and (b) schedule 2.

If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S' equivalent to S, Ti must appear before Tj . For example, the precedence graph for schedule 1 in Figure a contains the single edge T1 → T2, since all the instructions of T1 are executed before the first instruction of T2 is executed. Similarly, Figure b shows the precedence graph for schedule 2 with the single edge T2 → T1, since all the instructions of T2 are executed before the first instruction of T1 is executed.

The precedence graph for schedule 4 appears in Figure It contains the edge T1 →T2, because T1 executes read(A) before T2 executes write(A). It also contains the edge T2 → T1, because T2 executes read(B) before T1 executes write(B). If the precedence graph for S has a cycle, then schedule S is not conflict serializable.

If the graph contains no cycles, then the schedule S is conflict serializable. A serializability order of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph. There are, in general, several possible linear orders that can be obtained through a topological sorting. For example, the graph of Figure a has the two acceptable linear orderings shown in Figures b and c.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the

order of n2 operations, where n is the number of vertices in the graph (that is, the number of transactions). Thus, we have a practicalscheme for determining conflict serializability.

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 indeed do not contain cycles. The precedence graph for schedule 4 , on the other hand, contains a cycle, indicating that thisschedule is not conflict serializable. Testing for view serializability is rather complicated. In fact, it has been shown that the problem of testing for view serializability is itself NP-complete. Thus, almost certainly there exists no efficient algorithm to test for view serializability. See the bibliographical notes for references on testing for view serializability. However, concurrency-control schemes can still use sufficient conditions for view serializability. That is, if the sufficient conditions are satisfied, the schedule is view serializable, but there may be view-serializable schedules that do not satisfy the sufficient conditions.

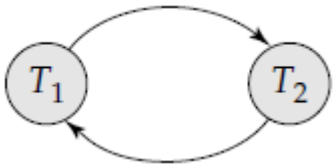


Figure 4.11 Precedence graph for schedule 4.

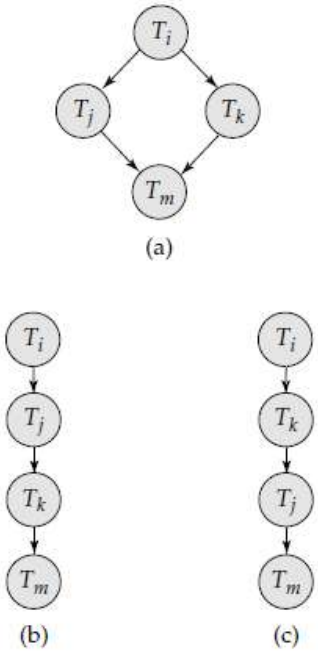


Figure 4.12 Illustration of topological sorting.

4.5 Recoverability, Recovery from Transaction Failure

So far, we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution. If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Recoverable Schedules

Consider schedule 11 in Figure 4.13, in which T9 is a transaction that performs only one instruction: read(A). Suppose that the system allows T9 to commit immediately after executing the read(A) instruction. Thus, T9 commits before T8 does. Now suppose that T8 fails before it commits. Since T9 has read the value of data item A written by T8, we must abort T9 to ensure transaction atomicity. However, T9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T8.

Schedule 11, with the commit happening immediately after the read(A) instruction, is an example of a non-recoverable schedule, which should not be allowed. Most database system require that all schedules be recoverable. A recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj .

Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti, we may have to roll back several transactions. Such situations occur if transactions have read data written by Ti. As an illustration, consider the partial schedule of Figure 4.14. Transaction T10 writes a value of A that is read by transaction T11.

Transaction T11 writes a value of A that is read by transaction T12. Suppose that, at this point, T10 fails. T10 must be rolled back. Since T11 is dependent on T10, T11 must be rolled back. Since T12 is dependent on T11, T12 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Figure 4.13 Schedule 11.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Figure 4.14 Schedule 12.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascadeless schedules. Formally, a cascadeless schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj . It is easy to verify that every cascadeless schedule is also recoverable.

Recovery from Transaction Failure

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Transactions, are preserved. An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information.

1.Transaction failure: There are two types of errors that may cause a transaction to fail:

a. Logical error. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
b. System error. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

2. System crash: There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the fail-stop assumption. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

3.Disk failure: A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database.

Recovery and Atomicity

Consider again our simplified banking system and transaction Ti that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of Ti, after output(BA) has taken place, but before output(BB) was executed, where BA and BB denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- Re execute Ti. This procedure will result in the value of A becoming $900, rather than $950. Thus, the system enters an inconsistent state.
- Do not re execute Ti. The current system state has values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by Ti. However, if Ti performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures. There are two ways to perform such outputs; we shall assume that transactions are executed serially; in other words, only a single transaction is active at a time.

4.6 Log-Based Recovery

The most widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database. There are several types of log records. An update log record describes a single database write. It has these fields:

- Transaction identifier is the unique identifier of the transaction that performed the write operation.
- Data-item identifier is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- Old value is the value of the data item prior to the write.
- New value is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

- <Ti start>. Transaction Ti has started.

- <Ti, Xj, V1, V2>. Transaction Ti has performed a write on data item Xj . Xj had value V1 before the write, and will have value V2 after the write.
- <Ti commit>. Transaction Ti has committed.
- <Ti abort>. Transaction Ti has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created. Observe that the log contains a complete record of all database activity. As a result, the volume of data stored in the log may become unreasonably large.

Deferred Database Modification

The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits. Recall that a transaction is said to be partially committed once the final action of the transaction has been executed.

The version of the deferred-modification technique that we describe in this section assumes that transactions are executed serially. When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction Ti proceeds as follows. Before Ti starts its execution, a record <Ti start> is written to the log. A write(X) operation by Ti results in the writing of a new record to the log. Finally, when Ti partially commits, a record <Ti commit> is written to the log.

When transaction Ti partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state.

Observe that only the new value of the data item is required by the deferred modification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field. To illustrate, reconsider our simplified banking system. Let T0 be a transaction that transfers $50 from account A to account B:

T0: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).

Let T1 be a transaction that withdraws $100 from account C:

T1: read(C);
C := C − 100;
write(C).

Suppose that these transactions are executed serially, in the order T0 followed by T1, and that the values of accounts A, B, and C before the execution took place were $1000, $2000, and $700, respectively. The portion of the log containing the relevant information on these two transactions appears below. There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of T0 and T1. One such order appears in Figure. Note that the value of A is changed in the database only after the record <T0, A, 950> has been placed in the log.

<T0 start>
<T0 , A, 950>
<T0 , B, 2050>
<T0 commit>
<T1 start>
<T1 , C, 600>
<T1 commit>

Portion of the database log corresponding to T0 and T1.

Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

- redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

The set of data items updated by Ti and their respective new values can be found in the log. The redo operation must be idempotent; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction Ti needs to be redone if and only if the log contains both the record <Ti start> and the record <Ti commit>. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

As an illustration, let us return to our banking example with transactions T0 and T1 executed one after the other in the order T0 followed by T1. Figure shows the log that results from the complete execution of T0 and T1. Let us suppose that the system crashes before the completion of the transactions, so that we can see how the recovery technique restores the database to a consistent state. Assume that the crash occurs just after the log record for the step

write(B)

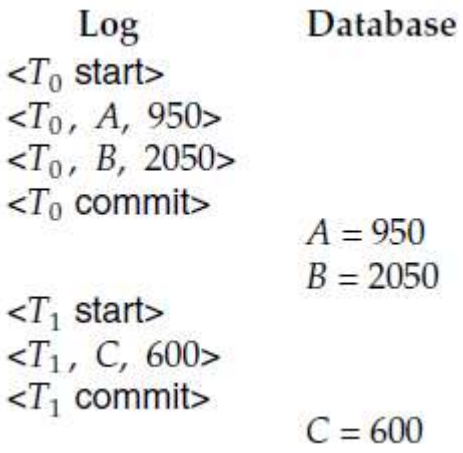


Figure 4.15 State of the log and database corresponding to T0 and T1.

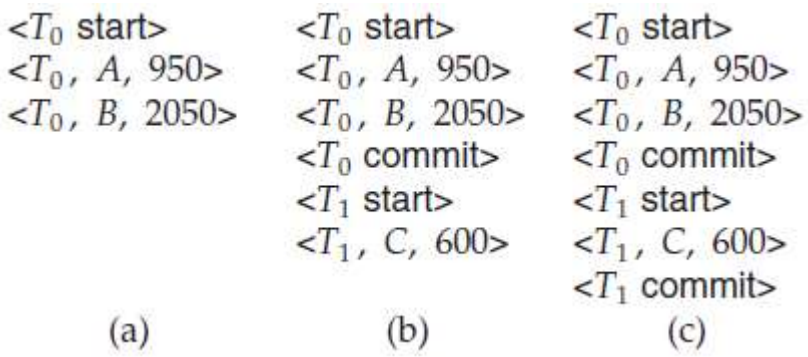


Figure 4.16 The same log shown at three different times.

of transaction T0 has been written to stable storage. The log at the time of the crash appears in Figure a. When the system comes back up, no redo actions need to be taken, since no commit record appears in the log. The values of accounts A and B remain $1000 and $2000, respectively. The log records of the incomplete transaction T0 can be deleted from the log.

Now, let us assume the crash comes just after the log record for the step:

write(C) of transaction T1 has been written to stable storage. In this case, the log at the time of the crash is as in Figure b. When the system comes back up, the operation redo(T0) is performed, since the record

<T0 commit>

appears in the log on the disk. After this operation is executed, the values of accounts A and B are $950 and $2050, respectively. The value of account C remains $700. As before, the log records of the incomplete transaction T1 can be deleted from the log. Finally, assume that a crash occurs just after the log record

<T1 commit>

is written to stable storage. The log at the time of this crash is as in Figure c. When the system comes back up, two commit records are in the log: one for T0 and one for T1. Therefore, the system must perform operations redo(T0) and redo(T1), in the order in which their commit records appear in the log. After the system executes these operations, the values of accounts A, B, and C are $950, $2050, and $600, respectively. Finally, let us consider a case in which a second system crash occurs during recovery from the first crash. Some changes may have been made to the database as a result of the redo

operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record

<Ti commit>

found in the log, the the system performs the operation redo(Ti). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

Immediate Database Modification

The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called uncommitted modifications. In the event of a crash or a transaction failure, the system must use the old-value field of the log records described in restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration.

Before a transaction Ti starts its execution, the system writes the record <Ti start> to the log. During its execution, any write(X) operation by Ti is preceded by the writing of the appropriate new update record to the log. When Ti partially commits, the system writes the record <Ti commit> to the log.

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage.We therefore require that, before execution of an output(B) operation, the log records corresponding to B be written onto stable storage.

As an illustration, let us reconsider our simplified banking system, with transactions T0 and T1 executed one after the other in the order T0 followed by T1. The portion of the log containing the relevant information concerning these two transactions appears in below shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T0 and T1.

<T0 start>
<T0 , A, 1000, 950>
<T0 , B, 2000, 2050>
<T0 commit>

<T1 start>

<T1 , C, 700, 600>

<T1 commit>

Portion of the system log corresponding to T0 and T1.

4.7 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints. During execution, the system maintains the log, using one of the two techniques In addition, the system periodically performs checkpoints, which requirethe following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>. Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction Ti that committed prior to the checkpoint. For such a transaction, the <Ti commit> record appears in the log before the<checkpoint> record. Any database modifications made by Ti must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti. This observation allows us to refine our previous recovery schemes. (We continue to assume that transactions are run serially.) After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction Ti that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first<checkpoint> record (since we are searching backward, the record found is the

final<checkpoint> record in the log); then it continues the search backward until it finds the next <Ti start> record. This record identifies a transaction Ti.

Once the system has identified transaction Ti, the redo and undo operations need to be applied to only transaction Ti and all transactions Tj that started executing after transaction Ti. Let us denote these transactions by the set T. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on the modification technique being used. For the immediate-modification technique, the recovery operations are:

- For all transactions Tk in T that have no <Tk commit> record in the log, execute undo(Tk).
- For all transactions Tk in T such that the record <Tk commit> appears in the log, execute redo(Tk). Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed.

  As an illustration, consider the set of transactions {T0, T1, . . ., T100} executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction T67. Thus, only transactions T67, T68, . . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

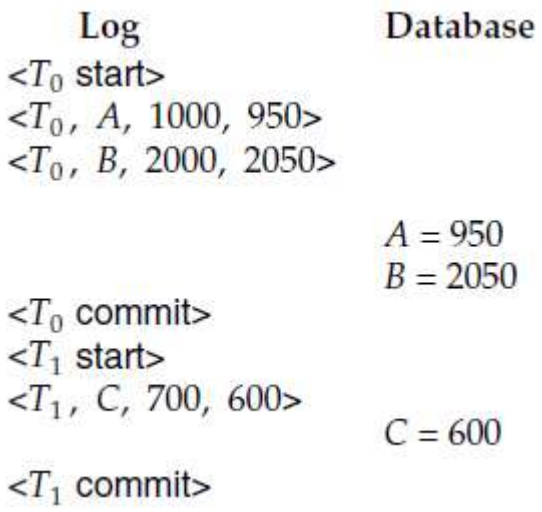


Figure 4.17 State of system log and database corresponding to T0 and T1.

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

- undo(Ti) restores the value of all data items updated by transaction Ti to the old values.
- redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

The set of data items updated by Ti and their respective old and new values can be found in the log. The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process. After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

- Transaction Ti needs to be undone if the log contains the record <Ti start>, but does not contain the record <Ti commit>.
- Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>.

As an illustration, return to our banking example, with transaction T0 and T1 executed one after the other in the order T0 followed by T1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 4.18. First, let us assume that the crash occurs just after the log record for the step

write(B)

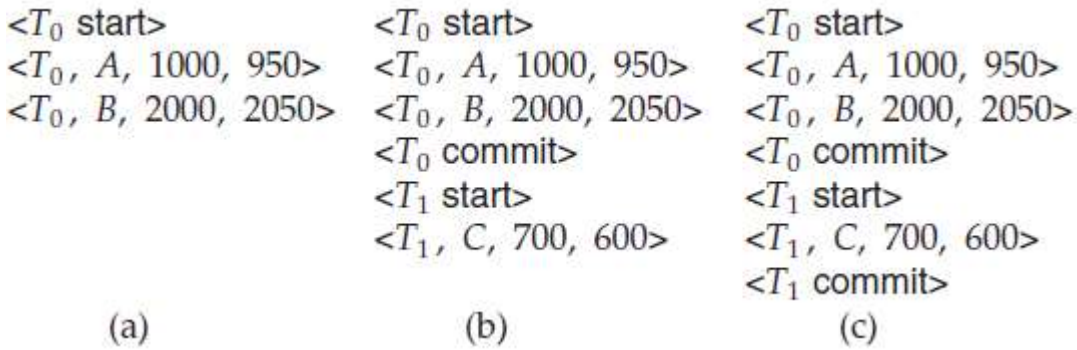


Figure 4.18 The same log, shown at three different times.

of transaction T0 has been written to stable storage. When the system comes back up, it finds the record <T0 start> in the log, but no corresponding <T0 commit> record. Thus, transaction T0 must be undone, so an undo(T0) is performed. As a result, the values in accounts A and B (on the disk) are restored to $1000 and $2000, respectively. Next, let us assume that the crash comes just after the log record for the step

write(C)

of transaction T1 has been written to stable storage When the system comes back up, two recovery actions need to be taken. The operation undo(T1) must be performed, since the record <T1 start> appears in the log, but there is no record<T1 commit>. The operation redo(T0) must be performed, since the log contains both the record <T0 start> and the record <T0 commit>. At the end of the entire recovery procedure, the values of accounts A, B, and C are $950, $2050, and $700, respectively. Note that the undo(T1) operation is performed before the redo(T0). In this example, the same outcome would result if the order were reversed. However, the order of doing undo operations first, and then redo operations, is important for the recovery algorithm.

Finally, let us assume that the crash occurs just after the log record<T1 commit> has been written to stable storage. When the system comes back up, both T0 and T1 need to be redone, since the records <T0 start> and <T0 commit> appear in the log, as do the records <T1 start> and <T1 commit>. After the system performs the recovery procedures redo(T0) and redo(T1), the values in accounts A, B, and C are $950, $2050, and $600, respectively.

4.8 Deadlock Handling

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions {T_0, T_1, T_2, ..., Tn}. T_0 needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can

create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available

but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction T_i requests for a lock on an item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.

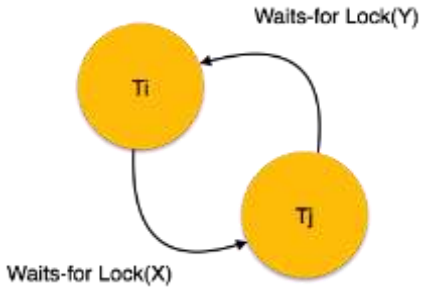


Figure 4.19 Cycle in a Graph

Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

4.9 Distributed Databases

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Distributed Database Management System

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

Factors Encouraging DDBMS

The following factors encourage moving over to DDBMS −

- **Distributed Nature of Organizational Units** − Most organizations in the current times are subdivided into multiple units that are physically distributed over the globe. Each unit requires its own set of local data. Thus, the overall database of the organization becomes distributed.
- **Need for Sharing of Data** − The multiple organizational units often need to communicate with each other and share their data and resources. This demands common databases or replicated databases that should be used in a synchronized manner.
- **Support for Both OLTP and OLAP** − Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) work upon diversified systems which may have common data. Distributed database systems aid both these processing by providing synchronized data.
- **Database Recovery** − One of the common techniques used in DDBMS is replication of data across different sites. Replication of data automatically helps in data recovery if database in any site is damaged. Users can access data from other sites while the damaged site is being reconstructed. Thus, database failure may become almost inconspicuous to users.
- **Support for Multiple Application Software** − Most organizations use a variety of application software each with its specific database support. DDBMS provides a uniform functionality for using the same data among different platforms.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development − If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

More Reliable − In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response − If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost − In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

Adversities of Distributed Databases

Following are some of the adversities associated with distributed databases.

- **Need for complex and expensive software** − DDBMS demands complex and often expensive software to provide data transparency and co-ordination across the several sites.
- **Processing overhead** − Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Data integrity** − The need for updating data in multiple sites pose problems of data integrity.
- **Overheads for improper data distribution** − Responsiveness of queries is largely dependent upon proper data distribution. Improper data distribution often leads to very slow response to user requests.

4.9.1 Distributed Data Storage

There are 2 ways in which data can be stored on different sites. These are:

1. Replication

In this approach, the entire relation is stored redundantly at 2 or more sites. If the entire database is available at all sites, it is a fully redundant database. Hence, in replication, systems maintain copies of data.

This is advantageous as it increases the availability of data at different sites. Also, now query requests can be processed in parallel.

However, it has certain disadvantages as well. Data needs to be constantly updated. Any change made at one site needs to be recorded at every site that relation is stored or else it may lead to inconsistency. This is a lot of overhead. Also, concurrency control becomes way more complex as concurrent access now needs to be checked over a number of sites.

2. Fragmentation

In this approach, the relations are fragmented (i.e., they're divided into smaller parts) and each of the fragments is stored in different sites where they're required. It must be made sure that the fragments are such that they can be used to reconstruct the original relation (i.e, there isn't any loss of data).

Fragmentation is advantageous as it doesn't create copies of data, consistency is not a problem.

Fragmentation of relations can be done in two ways:

Horizontal fragmentation – Splitting by rows – The relation is fragmented into groups of tuples so that each tuple is assigned to at least one fragment.

Vertical fragmentation – Splitting by columns – The schema of the relation is divided into smaller schemas. Each fragment must contain a common candidate key so as to ensure lossless join.

4.9.2 Concurrency Control in Distributed Systems

In this section, we will see how the above techniques are implemented in a distributed database system.

Distributed Two-phase Locking Algorithm

The basic principle of distributed two-phase locking is same as the basic two-phase locking protocol. However, in a distributed system there are sites designated as lock managers. A lock manager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.

Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types −

- **Centralized two-phase locking** − In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.
- **Primary copy two-phase locking** − In this approach, a number of sites are designated as lock control centers. Each of these sites has the responsibility of managing a defined

set of locks. All the sites know which lock control center is responsible for managing lock of which data table/fragment item.

- **Distributed two-phase locking** − In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its local site. The location of the lock manager is based upon data distribution and replication.

Distributed Timestamp Concurrency Control

In a centralized system, timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site's local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site's clock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager. During transaction, a transaction manager sends a lock request to the site's scheduler. The scheduler puts the request to the corresponding queue in increasing timestamp order. Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first.

Conflict Graphs

Another method is to create conflict graphs. For this transaction classes are defined. A transaction class contains two set of data items called read set and write set. A transaction belongs to a particular class if the transaction's read set is a subset of the class' read set and the transaction's write set is a subset of the class' write set. In the read phase, each transaction issues its read requests for the data items in its read set. In the write phase, each transaction issues its write requests.

A conflict graph is created for the classes to which active transactions belong. This contains a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and denotes conflicts within the class. A horizontal edge connects two nodes across two classes and denotes a write-write conflict among different classes. A diagonal edge connects two nodes across two classes and denotes a write-read or a read-write conflict among two classes.

The conflict graphs are analyzed to ascertain whether two transactions within the same class or across two different classes can be run in parallel.

Distributed Optimistic Concurrency Control Algorithm

Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm. For this extension, two rules are applied −

Rule 1 − According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

Rule 2 − According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

4.9.3 Directory Systems

In the pre computerization days, organizations would create physical directories of employees and distribute them across the organization. In general, a directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement. In the world of physical telephone directories, directories that satisfy lookups in the forward direction are called white pages, while directories that satisfy lookups in the reverse direction are called yellow pages

Directory Access Protocols

Several directory access protocols have been developed to provide a standardized way of accessing data in a directory. The most widely used among them today is the Lightweight Directory Access Protocol (LDAP).

The reasons for using a specialized protocol for accessing directory information:

• First, directory access protocols are simplified protocols that cater to a limited type of access to data. They evolved in parallel with the database access protocols.

• Second, and more important, directory systems provide a simple mechanism to name objects in a hierarchical fashion, similar to file system directory names, which can be used in a distributed directory system to specify what information is stored in each of the directory servers.

LDAP: Lightweight Directory Access Protocol

A directory system is implemented as one or more servers, which service multiple clients. Clients use the application programmer interface defined by directory system to communicate with the directory servers. Directory access protocols also define a data model and access control.

The X.500 directory access protocol, defined by the International Organization for Standardization (ISO), is a standard for accessing directory information. However, the protocol is rather complex, and is not widely used. The Lightweight Directory Access Protocol (LDAP) provides many of the X.500 features, but with less complexity, and is widely used.

LDAP Data Model

In LDAP directories store entries, which are similar to objects. Each entry must have a distinguished name (DN), which uniquely identifies the entry. A DN is in turn made up of a sequence of relative distinguished names (RDNs).

Data Manipulation

Unlike SQL, LDAP does not define either a data-definition language or a data manipulation language. However, LDAP defines a network protocol for carrying out data definition and manipulation. Users of LDAP can either use an application programming interface, or use tools provided by various vendors to perform data definition and manipulation. LDAP also defines a file format called LDAP Data Interchange Format (LDIF) that can be used for storing and exchanging information.

Distributed Directory Trees

Information about an organization may be split into multiple DITs, each of which stores information about some entries. A node in a DIT may contain a referral to another node in another DIT. Referrals are the key component that help organize a distributed collection of directories into an integrated system. When a server gets a query on a DIT, it may return a referral to the client, which then issues a query on the referenced DIT. Access to the referenced DIT is transparent, proceeding without the

user's knowledge. Alternatively, the server itself may issue the query to the referred DIT and return the results along with locally computed results

The hierarchical naming mechanism used by LDAP helps break up control of information across parts of an organization. The referral facility then helps integrate all the directories in an organization into a single virtual directory

IMPORTANT QUESTIONS

1. What do you mean by deadlock? Explain different deadlock techniques.
2. What are the problem faced when concurrent transactions are executed in an Uncontrolled manner? Give an example and explain.
3. What is a schedule (history)? Define the concepts of recoverable, cascade less, and Strict schedules, and compare them in terms of their recoverability.
4. What do you mean by Transaction? Explain ACID properties of transaction with suitable example.
5. Discuss the Recovery Technique from transaction failure and their types with example.
6. What is a Schedule? What are the problems associated with concurrent schedule? Explain with suitable examples.
7. What is Serializability? Explain view and conflict serializability.
8. List all possible schedules for transactions T1 and T2 and determine which is conflict serializable and which are not.

	T1	T2
a.		
b.	read_item(X);	read_item(X);
c.	X: =X-N;	X: =X+M;
d.	Write_item(X);	write_item(X);
e.	Read_item(Y);	
f.	Y: =Y+N;	
g.	Write_item(Y);	

9. What is log based recovery? Explain the difference between deferred and immediate database modification. How checkpoint reduce the overheads in database modification?
10. Explain the role of check pointing in detail.

MULTIPLE CHOICE QUESTIONS

1. Collections of operations that form a single logical unit of work are called _____

a) Views

b) Networks

c) Units

d) Transactions

2. The "all-or-none" property is commonly referred to as _____

a) Isolation

b) Durability

c) Atomicity

d) None of the mentioned

3. Which of the following is a property of transactions?

a) Atomicity b)
Durability

c) Isolation

d) All of the mentioned

4. Execution of translation in isolation preserves the _____ of a database

a) Atomicity

b) Consistency

c) Durability

d) All of the mentioned

5. Which of the following systems is responsible for ensuring durability?

a) Recovery system

b) Atomic system

c) Concurrency control system

d) Compiler system

6. A transaction that has not been completed successfully is called as _____

a) Compensating transaction

b) Aborted transaction

c) Active transaction

d) Partially committed transaction

7. The execution sequences in concurrency control are termed as _____

a) Serials

b) Schedules

c) Organizations

d) Time tables

8. The scheme that controls the interaction between executing transactions is called as _____

a) Concurrency control scheme

b) Multiprogramming scheme

c) Serialization scheme

d) Schedule scheme

9. I and J are _____ if they are operations by different transactions on the same data item, and at least one of them is a write operation.

a) Conflicting

b) Overwriting

c) Isolated

d) Durable

10. If a schedule S can be transformed into a schedule S' by a series of swaps of non- conflicting instructions, then S and S' are

a) Non conflict equivalent

b) Equal

c) Conflict equivalent

d) Isolation equivalent

11 A schedule is _____ if it is conflict equivalent to a serial schedule.

a) Conflict serializable

b) Conflicting

c) Non serializable

d) None of the mentioned

12. The set of _____ in a precedence graph consists of all the transactions participating in the schedule

a) Vertices

b) Edges

c) Directions

d) None of the mentioned

13. A _____of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

a) Serializability order

b) Direction graph

c) Precedence graph

d) Scheduling scheme

14. Which of the following is the most expensive method?

a) Timestamping

b) Plain locking

c) Predicate locking

d) Snapshot isolation

15. A transaction that performs only one operation is called as a _____

a) Partial schedule

b) Complete schedule

c) Dependent schedule

d) Independent schedule

16. The phenomenon in which one failure leads to a series of transaction rollbacks is called as __

a) Cascading rollback

b) Cascadeless rollback

c) Cascade cause

d) None of the mentioned

17. A _____ is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the commit operation of Tj

a) Partial schedule

b) Dependent schedule

c) Recoverable schedule

d) None of the mentioned

18. The average time for a transaction to be completed after it has been submitted is called as __

a) Minimum response time

b) Average response time

c) Average reaction time

d) Minimum reaction time

19. If a schedule is equivalent to a serial schedule, it is called as a _____

a) Serializable schedule

b) Equivalent schedule

c) Committed schedule

d) None of the mentioned

20. Which of the following is not a type of a schedule?

a) Partial schedule

b) Dependent schedule

c) Recoverable schedule

d) None of the mentioned

Answer Key

1	d	**11**	a
2	c	**12**	a
3	d	**13**	a
4	b	**14**	c
5	a	**15**	a
6	b	**16**	a
7	b	**17**	c
8	a	**18**	b
9	a	**19**	a
10	c	**20**	a

UNIT-V

CONCURRENCY CONTROL TECHNIQUES

5.1 Concurrency Control

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Problems with Concurrency control

- **Lost Updates** occur when multiple transactions select the same row and update the row based on the value selected
- Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (**dirty read**)
- **Non-Repeatable Read** occurs when a second transaction is trying to access the same row several times and reads different data each time.
- **Incorrect Summary issue** occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

Need of Concurrency control in database system

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

5.2 Locking Techniques for Concurrency Control

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

- Lock-Based Protocols
- Two Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Deadlock

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

Strict Two-Phase Locking Method

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over.

Centralized 2PL

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

Primary copy 2PL

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

Distributed 2PL

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

5.3 Timestamp-based Protocols for Concurrency Control

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj . To implement this scheme, we associate with each data item Q two timestamp values:

• W-timestamp(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.

• R-timestamp(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

This protocol operates as follows:

1. Suppose that transaction Ti issues read(Q).

a. If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction Ti issues write(Q).

a. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

b. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back. c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti). If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Example:

Suppose there are there transactions T1, T2, and T3.

T1 has entered the system at time 0010

T2 has entered the system at 0020

T3 has entered the system at 0030

Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

Starvation is possible if the same transaction is restarted and continually aborted.

Thomas' Write Rule

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

5.4 Validation Based Protocol

Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.

The Validation based Protocol is performed in the following three phases:

1. Read Phase
2. Validation Phase
3. Write Phase

Read Phase

In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.

Validation Phase

In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.

Write Phase

In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.

To perform the validation test, we need to know when the various phases of transactions Ti took place. We shall, therefore, associate three different timestamps with transaction Ti:

1. Start(Ti), the time when Ti started its execution.

2. Validation(Ti), the time when Ti finished its read phase and started its validation phase.

3. Finish(Ti), the time when Ti finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(Ti).

The validation test for transaction Tj requires that, for all transactions Ti with TS(Ti) < TS(Tj), one of the following two conditions must hold:

1. Finish(Ti) < Start(Tj). Since Ti completes its execution before Tj started, the serializability order is indeed maintained.

2. The set of data items written by Ti does not intersect with the set of data items read by Tj , and Ti completes its write phase before Tj starts its validation phase (Start(Tj) < Finish(Ti) < Validation(Tj)). This condition ensures that the writes of Ti and Tj do not overlap. Since the writes of Ti do not affect the read of Tj , and since Tj cannot affect the read of Ti, the serializability order is indeed maintained.

This validation scheme is called the optimistic concurrency control scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end

5.5 Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if Ti could issue a single lock request to lock the

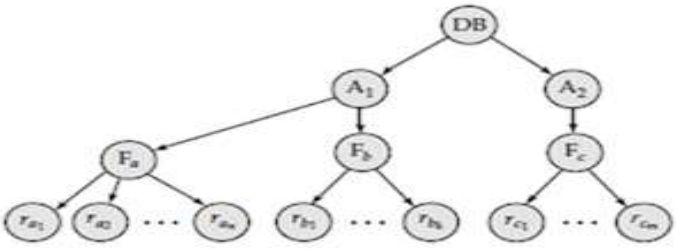


Figure:5.1 Granularity Hierarchy

entire database. On the other hand, if transaction Tj needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of above Figure, which consists of four levels of nodes. The highest level represents the entire database. Below it is nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction Ti gets an **explicit lock** on file Fc of Figure 16.16, in exclusive mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of Fc explicitly.

Suppose that transaction Tj wishes to lock record rb6 of file Fb. Since Ti has locked Fb explicitly, it follows that rb6 is also locked (implicitly). But, when Tj issues a lock request for rb6 , rb6 is not explicitly locked! How does the system determine whether Tj can lock rb6 ? Tj must traverse the tree from the root to record rb6 . If any node in that path is locked in an incompatible mode, then Tj must be delayed.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 5.2: Compatibility Matrix

Suppose now that transaction Tk wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that Tk should not succeed in locking the root node, since Ti is currently holding a lock on part of the tree (specifically, on file Fb). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, de- feats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **inten- tion lock modes**. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, Q—must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in above figure.

The **multiple-granularity locking protocol**, which ensures serializability, is this:

Each transaction Ti can lock a node Q by following these rules:

1. It must observe the lock-compatibility function of figure

2. It must lock the root of the tree first, and can lock it in any mode.

3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.

4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.

5. It can lock a node only if it has not previously unlocked any node (that is, Ti is two phase).

6. It can unlock a node Q only if it currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in top- down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure 16.16 and these trans- actions:

• Suppose that transaction T18 reads record ra2 in file Fa. Then, T18 needs to lock the database, area A1, and Fa in IS mode (and in that order), and finally to lock ra2 in S mode.

• Suppose that transaction T19 modifies record ra9 in file Fa. Then, T19 needs to lock the database, area A1, and file Fa in IX mode, and finally to lock ra9 in X mode.

• Suppose that transaction T20 reads all the records in file Fa. Then, T20 needs to lock the database and area A1 (in that order) in IS mode, and finally to lock Fa in S mode.

• Suppose that transaction T21 reads the entire database. It can do so after locking the database in S mode.

We note that transactions T18, T20, and T21 can access the database concurrently. Transaction T19 can execute concurrently with T18, but not with either T20 or T21.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

• Short transactions that access only a few data items

• Long transactions that produce reports from an entire file or set of files There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. See the bibliographical notes for additional references. Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol. There are techniques to reduce

deadlock frequency in the multiple-granularity protocol, and also to eliminate dead- lock entirely. These techniques are referenced in the bibliographical notes.

5.6 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For ex- ample, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In multiversion concurrency control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrency- control manager selects one of the versions of Q to be read. The concurrency-control

scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

Multiversion Timestamp Ordering

The most common transaction ordering technique used by multiversion schemes is timestamping. With each transaction Ti in the system, we associate a unique static timestamp, denoted by TS(Ti). The database system assigns this timestamp before the transaction starts execution, as described in Section 16.2.

With each data item Q, a sequence of versions <Q1, Q2,.. ., Qm> is associated. Each version Qk contains three data fields:

• Content is the value of version Qk .

• W-timestamp(Qk) is the timestamp of the transaction that created version Qk .

• R-timestamp(Qk) is the largest timestamp of any transaction that successfully read version Qk .

A transaction—say, Ti—creates a new version Qk of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by Ti. The system initializes the W-timestamp and R-timestamp to TS(Ti). It updates the R-timestamp value of Qk whenever a transaction Tj reads the content of Qk , and R-timestamp(Qk) < TS(Tj).

The multiversion timestamp-ordering scheme presented next ensures serializ- ability. The scheme operates as follows. Suppose that transaction Ti issues a read(Q) or write(Q) operation. Let Qk denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(Ti).

1. If transaction Ti issues a read(Q), then the value returned is the content of version Qk .

2. If transaction Ti issues write(Q), and if TS(Ti) < R-timestamp(Qk), then the sys- tem rolls back transaction Ti. On the other hand, if TS(Ti) = W-timestamp(Qk), the system overwrites the contents of Qk ; otherwise it creates a new version of Q.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is “too late” in doing a write. More precisely, if Ti attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

Versions that are no longer needed are removed according to the following rule.

Suppose that there are two versions, Qk and Qj , of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system.

Then, the older of the two versions Qk and Qj will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait. In typical database systems, where reading is a more frequent operation than is writing, this advantage may be of major practical significance.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. This multiversion timestamp-ordering scheme does not ensure

recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp-ordering scheme, to make it recoverable and cascadeless.

Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**.

Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp- ordering protocol for performing reads. Thus, when a read-only transaction Ti issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than TS(Ti).

When an update transaction reads an item, it gets a shared lock on the item, and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item, and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value ∞, a value greater than that of any possible timestamp.

When the update transaction Ti completes its actions, it carries out commit processing: First, Ti sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, Ti increments ts-counter by 1. Only one update transaction is allowed to perform commit processing at a time.

As a result, read-only transactions that start after Ti increments ts-counter will see the values updated by Ti, whereas those that start before Ti increments ts-counter will see the value before the updates by Ti. In either case, read-only transactions never need to wait for locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering.

Suppose there are two versions, Qk and Qj , of a data item, and that both versions have a timestamp less than the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions Qk and Qj will not be used again and can be deleted.

Multiversion two-phase locking or variations of it are used in some commercial database systems.

5.7 Recovery with Concurrent Transaction

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

Interaction with concurrency control:

In this scheme, the recovery scheme depends greatly on the concurrency control scheme that is used. So, to rollback a failed transaction, we must undo the updates performed by the transaction.

Transaction rollback:

- In this scheme, we rollback a failed transaction by using the log.
- The system scans the log backward a failed transaction, for every log record found in the log the system restores the data item.

Checkpoints:

- Checkpoints is a process of saving a snapshot of the applications state so that it can restart from that point in case of failure.
- Checkpoint is a point of time at which a record is written onto the database form the buffers.
- Checkpoint shortens the recovery process.
- When it reaches the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till the next checkpoint and so on.
- The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed.

Restart recovery:

- When the system recovers from a crash, it constructs two lists.

- The undo-list consists of transactions to be undone, and the redo-list consists of transaction to be redone.
- The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

.

5.8 Case Study of Oracle

Operational Data Analytics Insurance firms succeed through their ability to identify and quantify risks facing their clients. They are under constant and increasing pressure to rapidly consider every available quantifiable factor to develop profiles of insurance risk. To this end, insurers collect a vast amount of operational data about policy holders and insured objects. While extremely valuable, this operational data must often wait to be coaxed into a traditional data warehouse format, for even later assessment by an analyst. This case study discusses how Mobiliar restructured their multi-database data warehouse environment to streamline risk analysis using operational data.

The ultimate goal for organizations, like Mobiliar, that want to streamline their analytics is being able to analyze their data in real time—without having a negative impact on OLTP [online transaction processing] performance and without having to wait for the classic ETL [extract, transform, and load] process to load transaction data into the data warehouse.

Astonishing Proof of Concept (PoC) Results

Oracle Database In-Memory has a unique dual format (rows and columns) that maintains the transactional data in both row and columnar format in memory, enabling real-time analytics to be performed immediately across all transactions, thereby eliminating delays and reliance on transforming transactions into a data mart, data warehouse or other analytic store for examination. To prove that Oracle Database In-Memory could truly allow Mobiliar to use their operational data for real-time analytics, the database team at Mobiliar set up a proof of concept to test different analytical scenarios. Key to the proof of concept for Mobiliar was choosing scenarios that represented typical business cases for the insurance company.

About Oracle Database In-Memory Oracle Database In-Memory transparently accelerates analytic queries by orders of magnitude, enabling real-time business decisions. It dramatically accelerates data warehouses and mixed workload OLTP environments. The unique "dual-format" approach

automatically maintains data in both the existing Oracle row format for OLTP operations, and in a new purely in-memory column format optimized for analytical processing. Both formats are simultaneously active and transactionally consistent. Embedding the column store into Oracle Database ensures it is fully compatible with ALL existing features, and requires absolutely no changes in the application layer.

Important Questions

Q.1: a) Describe major problems associated with concurrent processing with examples.

b) What is the role of locks in avoiding these problems.

c) What is phantom phenomenon

Q.2: What do you mean by multiple granularities? Explain in detail.

Q.3: Define deadlock. Explain deadlock recovery and preverntion techniques

Q.4: Explain multiversion concurrency control in detail.

Q.5: Explain the working of various time stamping protocols for concurrency control

Q.6: Explain the difference between two phase commit protocol and three phase commit protocol.

Q.7: What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.

Q.8: What are the problems encountered in distributed DBMS while considering concurrency control and recovery?

Q 9: Distinguish between data replication and data fragmentation.

Q 10: Explain in detail Validation Based Protocol.

MULTIPLE CHOICE QUESTIONS

1.If a transaction has obtained a _____ lock, it can read but cannot write on the item

a) Shared mode

b) Exclusive mode

c) Read only mode

d) Write only mode

2.If a transaction has obtained a _____ lock, it can both read and write on the item

a) Shared mode

b) Exclusive mode

c) Read only mode

d) Write only mode

3.A transaction can proceed only after the concurrency control manager _____ the lock to the transaction

a) Grants

b) Requests

c) Allocates

d) None of the mentioned

4.If a transaction can be granted a lock on an item immediately in spite of the presence of another mode, then the two modes are said to be _____

a) Concurrent

b) Equivalent

c) Compatible

d) Executable

5.A transaction is made to wait until all _____ locks held on the item are released

a) Compatible

b) Incompatible

c) Concurrent

d) Equivalent

6.State true or false: It is not necessarily desirable for a transaction to unlock a data item immediately after its final access

a) True

b) False

7.The situation where no transaction can proceed with normal execution is known as _____

a) Road block

b) Deadlock

c) Execution halt

d) Abortion

8.The protocol that indicates when a transaction may lock and unlock each of the data items is called as _____

a) Locking protocol

b) Unlocking protocol

c) Granting protocol

d) Conflict protocol

9.If a transaction Ti may never make progress, then the transaction is said to be _____

a) Deadlocked

b) Starved

c) Committed

d) Rolled back

10.The two phase locking protocol consists which of the following phases?

a) Growing phase

b) Shrinking phase

c) More than one of the mentioned

d) None of the mentioned

11.If a transaction may obtain locks but may not release any locks then it is in _____ phase

a) Growing phase

b) Shrinking phase

c) Deadlock phase

d) Starved phase

12.If a transaction may release locks but may not obtain any locks, it is said to be in _____ phase

a) Growing phase

b) Shrinking phase

c) Deadlock phase

d) Starved phase

13.Which of the following cannot be used to implement a timestamp

a) System clock

b) Logical counter

c) External time counter

d) None of the mentioned

14.A logical counter is _____ after a new timestamp has been assigned

a) Incremented

b) Decremented

c) Doubled

d) Remains the same

15.W-timestamp(Q) denotes?

a) The largest timestamp of any transaction that can execute write(Q) successfully

b) The largest timestamp of any transaction that can execute read(Q) successfully

c) The smallest timestamp of any transaction that can execute write(Q) successfully

d) The smallest timestamp of any transaction that can execute read(Q) successfully

16.R-timestamp(Q) denotes?

a) The largest timestamp of any transaction that can execute write(Q) successfully

b) The largest timestamp of any transaction that can execute read(Q) successfully

c) The smallest timestamp of any transaction that can execute write(Q) successfully

d) The smallest timestamp of any transaction that can execute read(Q) successfully

17.A _____ ensures that any conflicting read and write operations are executed in timestamp order

a) Organizational protocol

b) Timestamp ordering protocol

c) Timestamp execution protocol

d) 802-11 protocol

18.The default timestamp ordering protocol generates schedules that are

a) Recoverable

b) Non-recoverable

c) Starving

d) None of the mentioned

19.Which of the following timestamp based protocols generates serializable schedules?

a) Thomas write rule

b) Timestamp ordering protocol

c) Validation protocol

d) None of the mentioned

20.State true or false: The Thomas write rule has a greater potential concurrency than the timestamp ordering protocol

a) True

b) False

21.In timestamp ordering protocol, suppose that the transaction Ti issues read(Q) and TS(Ti)<W-timestamp(Q), then

a) Read operation is executed

b) Read operation is rejected

c) Write operation is executed

d) Write operation is rejected

22.In timestamp ordering protocol, suppose that the transaction Ti issues write(Q) and TS(Ti)<W-timestamp(Q), then

a) Read operation is executed

b) Read operation is rejected

c) Write operation is executed

d) Write operation is rejected

23.The _____ requires each transaction executes in two or three different phases in its lifetime

a) Validation protocol

b) Timestamp protocol

c) Deadlock protocol

d) View protocol

24.During _____ phase, the system reads data and stores them in variables local to the transaction.

a) Read phase

b) Validation phase

c) Write phase

d) None of the mentioned

25.During the _____ phase the validation test is applied to the transaction

a) Read phase

b) Validation phase

c) Write phase

d) None of the mentioned

26.During the _____ phase, the local variables that hold the write operations are copied to the database

a) Read phase

b) Validation phase

c) Write phase

d) None of the mentioned

27.Read only operations omit the _____ phase

a) Read phase

b) Validation phase

c) Write phase

d) None of the mentioned

28.Which of the following timestamp is used to record the time at which the transaction started execution?

a) Start(i)

b) Validation(i)

c) Finish(i)

d) Write(i)

29.Which of the following timestamps is used to record the time when a transaction has finished its read phase?

a) Start(i)

b) Validation(i)

c) Finish(i)

d) Write(i)

30.Which of the following timestamps is used to record the time when a database has completed its write operation?

a) Start(i)

b) Validation(i)

c) Finish(i)

d) Write(i)

31.State true or false: Locking and timestamp ordering force a wait or rollback whenever a conflict is detected.

a) True

b) False

32.State true or false: We determine the serializability order of validation protocol by the validation ordering technique

a) True

b) False

33.In a granularity hierarchy the highest level represents the

a) Entire database

b) Area

c) File

d) Record

34.If a node is locked in an intention mode, explicit locking is done at a lower level of the tree. This is called

a) Intention lock modes

b) Explicit lock

c) Implicit lock

d) Exclusive lock

35.If a node is locked in _____ then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.

a) Intention lock modes

b) Intention-shared-exclusive mode

c) Intention-exclusive (IX) mode

d) Intention-shared (IS) mode

36.This validation scheme is called the _____ scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

a) Validation protocol

b) Validation-based protocol

c) Timestamp protocol

d) Optimistic concurrency-control

37.The file organization which allows us to read records that would satisfy the join condition by using one block read is

a) Heap file organization

b) Sequential file organization

c) Clustering file organization

d) Hash files organization

38.DBMS periodically suspends all processing and synchronizes its files and journals through the use of

a) Checkpoint facility

b) Backup facility

c) Recovery manager

d) Database change log

39.The extent of the database resource that is included with each lock is called the level of

a) Impact

b) Granularity

c) Management

d) DBMS control

40.A condition that occurs when two transactions wait for each other to unlock data is known as a(n)

a) Shared lock

b) Exclusive lock

c) Binary lock

d) Deadlock

Answer Key

1	A	11	A	21	B	31	A
2	B	12	B	22	D	32	B
3	A	13	C	23	A	33	A

4	C	14	A	24	A	34	A
5	A	15	A	25	B	35	C
6	A	16	B	26	C	36	A
7	B	17	B	27	C	37	C
8	A	18	B	28	A	38	A
9	B	19	A	29	B	39	B
10	C	20	A	30	C	40	D

LAST YEAR AKTU QUESTION PAPER SOLUTION

Section-A

1. **Attempt all questions in brief**

a) What is Relational Algebra?

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

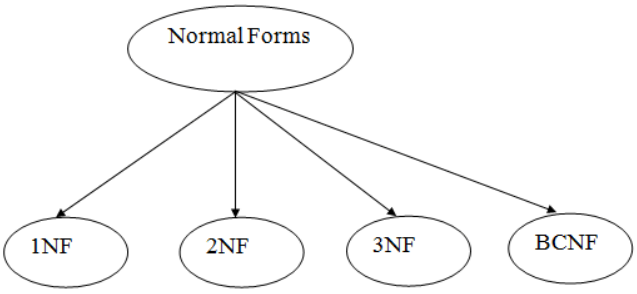
b) Explain normalisation. What is normal form?

Normalization

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Types of Normal Forms

There are the four types of normal forms:

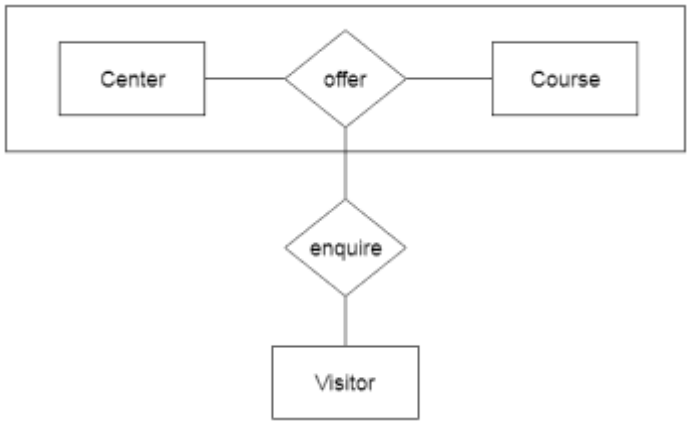


Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
4NF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

c) What do you mean by aggregation?

In aggregation, the relation between two entities is treated as a single entity. In aggregation, relationship with its corresponding entities is aggregated into a higher level entity.

For example: Center entity offers the Course entity act as a single entity in the relationship which is in a relationship with another entity visitor. In the real world, if a visitor visits a coaching center then he will never enquiry about the Course only or just about the Center instead he will ask the enquiry about both.

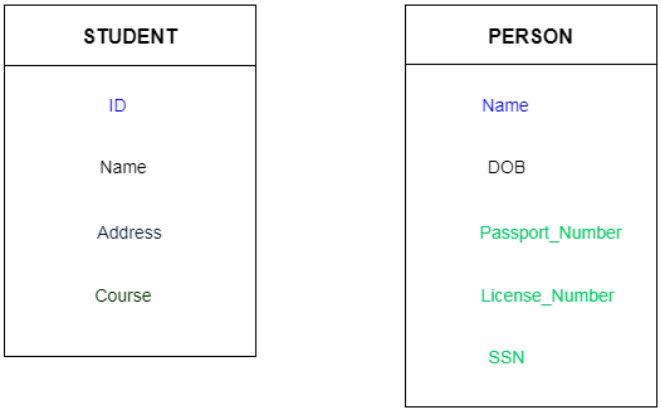


d)Define super key, candidate key, primary key and foreign key.

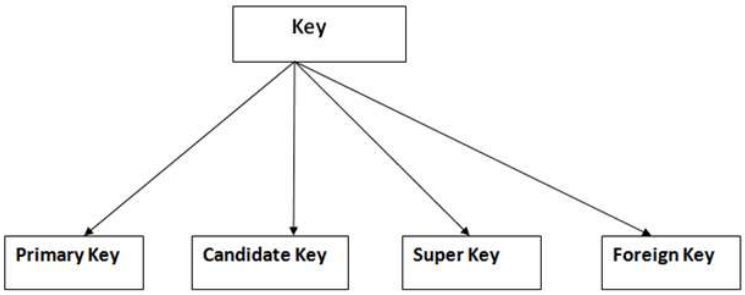
Keys

- Keys play an important role in the relational database.
- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

For example: In Student table, ID is used as a key because it is unique for each student. In PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.

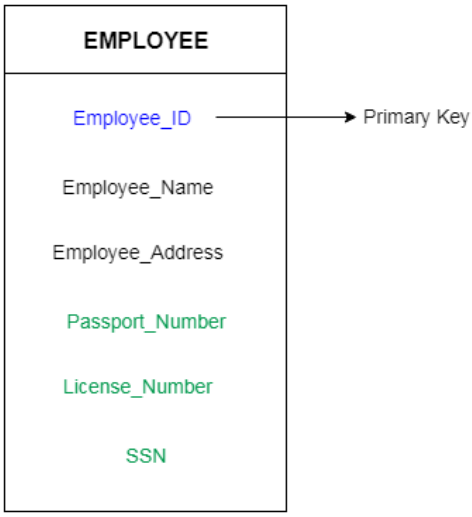


Types of key:



1. Primary key

- It is the first key which is used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys as we saw in PERSON table. The key which is most suitable from those lists become a primary key.
- In the EMPLOYEE table, ID can be primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary key since they are also unique.
- For each entity, selection of the primary key is based on requirement and developers.

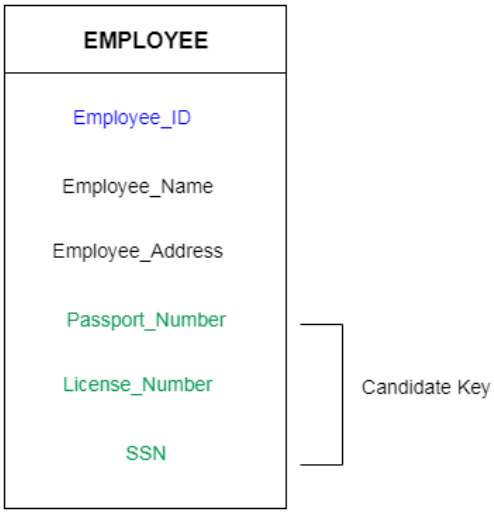


2. Candidate key

- A candidate key is an attribute or set of an attribute which can uniquely identify a tuple.

- The remaining attributes except for primary key are considered as a candidate key. The candidate keys are as strong as the primary key.

For example: In the EMPLOYEE table, id is best suited for the primary key. Rest of the attributes like SSN, Passport_Number, and License_Number, etc. are considered as a candidate key.



3. Super Key

Super key is a set of an attribute which can uniquely identify a tuple. Super key is a superset of a candidate key.

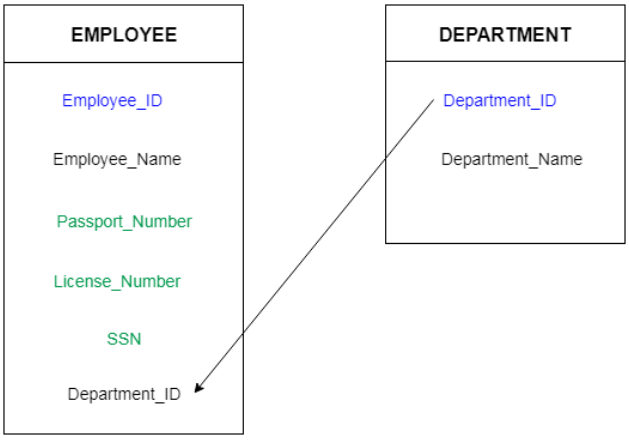
For example: In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME) the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.

The super key would be EMPLOYEE-ID, (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

4. Foreign key

- Foreign keys are the column of the table which is used to point to the primary key of another table.

- In a company, every employee works in a specific department, and employee and department are two different entities. So we can't store the information of the department in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id as a new attribute in the EMPLOYEE table.
- Now in the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.



e) What is strong and weak entity set?

Strong Entity

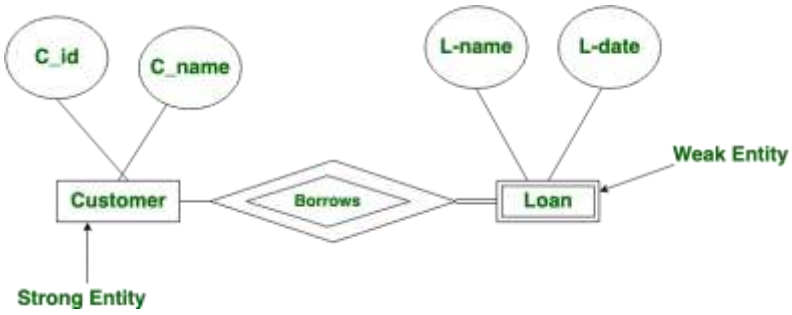
A strong entity is not dependent of any other entity in the schema. A strong entity will always have a primary key. Strong entities are represented by a single rectangle. The relationship of two strong entities is represented by a single diamond.

Various strong entities, when combined together, create a strong entity set.

Weak Entity

A weak entity is dependent on a strong entity to ensure the its existence. Unlike a strong entity, a weak entity does not have any primary key. It instead has a partial discriminator key. A weak entity is represented by a double rectangle.

The relation between one strong and one weak entity is represented by a double diamond.



Difference between Strong and Weak Entity:

	Strong Entity	Weak Entity
1.	Strong entity always has primary key.	While weak entity has partial discriminator key.
2.	Strong entity is not dependent of any other entity.	Weak entity is depend on strong entity.
3.	Strong entity is represented by single rectangle.	Weak entity is represented by double rectangle.
4.	Two strong entity's relationship is represented by single diamond.	While the relation between one strong and one weak entity is represented by double diamond.
5.	Strong entity have either total participation or not.	While weak entity always has total participation.

f) What do you mean by conflict serializable schedule?

Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

Conflicting Operations

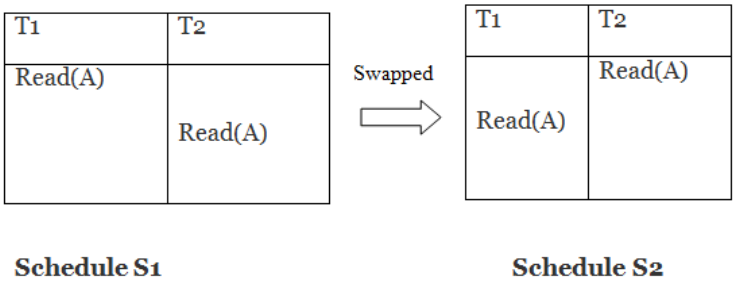
The two operations become conflicting if all conditions satisfy:

1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

Example:

Swapping is possible only if S1 and S2 are logically equal.

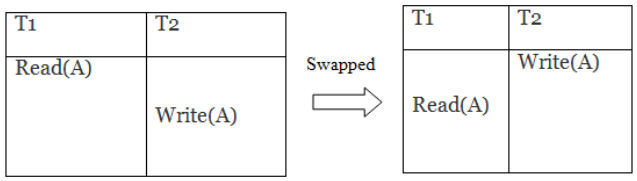
1. T1: Read(A) T2: Read(A)



Schedule S1 **Schedule S2**

Here, S1 = S2. That means it is non-conflict.

2. T1: Read(A) T2: Write(A)



Schedule S1 **Schedule S2**

Here, S1 ≠ S2. That means it is conflict.

Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

Example:

Non-serial schedule

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule S1

Serial Schedule

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Schedule S2

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

T1	**T2**
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)

	Write(B)

Since, S1 is conflict serializable.

g) Define concurrency control.

Concurrency Control

- In the concurrency control, the multiple transactions can be executed simultaneously.
- It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

Problems of concurrency control

Several problems can occur when concurrent transactions are executed in an uncontrolled manner. Following are the three problems in concurrency control.

1. Lost updates
2. Dirty read
3. Unrepeatable read

1. Lost update problem

- When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.
- If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

Example:

Transaction-X	Time	Transaction-Y
——	t1	——
Read A	t2	——
——	t3	Read A
Update A	t4	——
——	t5	Update A
——	t6	——

Here,

- At time t2, transaction-X reads A's value.
- At time t3, Transaction-Y reads A's value.
- At time t4, Transactions-X writes A's value on the basis of the value seen at time t2.
- At time t5, Transactions-Y writes A's value on the basis of the value seen at time t3.
- So at time T5, the update of Transaction-X is lost because Transaction y overwrites it without looking at its current value.
- Such type of problem is known as Lost Update Problem as update made by one transaction is lost here.

2. Dirty Read

- The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.
- A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

Example:

Transaction-X	Time	Transaction-Y
——	t1	——
——	t2	Update A
Read A	t3	——
——	t4	Rollback
——	t5	——

- At time t2, transaction-Y writes A's value.
- At time t3, Transaction-X reads A's value.
- At time t4, Transactions-Y rollbacks. So, it changes A's value back to that of prior to t1.
- So, Transaction-X now contains a value which has never become part of the stable database.
- Such type of problem is known as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

3. Inconsistent Retrievals Problem

- Inconsistent Retrievals Problem is also known as unrepeatable read. When a transaction calculates some summary function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.
- A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

Example:

Suppose two transactions operate on three accounts.

Account-1	Account-2	Account-3
Balance = 200	Balance = 250	Balance = 150

Transaction-X	Time	Transaction-Y
——	t1	——
Read Balance of Acc-1 sum <-- 200 Read Balance of Acc-2	t2	——
Sum <-- Sum + 250 = 450	t3	——
——	t4	Read Balance of Acc-3
——	t5	Update Balance of Acc-3 150 --> 150 - 50 --> 100
——	t6	Read Balance of Acc-1
——	t7	Update Balance of Acc-1 200 --> 200 + 50 --> 250
Read Balance of Acc-3	t8	COMMIT
Sum <-- Sum + 250 = 550	t9	——

- Transaction-X is doing the sum of all balance while transaction-Y is transferring an amount 50 from Account-1 to Account-3.
- Here, transaction-X produces the result of 550 which is incorrect. If we write this produced result in the database, the database will become an inconsistent state because the actual sum is 600.
- Here, transaction-X has seen an inconsistent state of the database.

Concurrency Control Protocol

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1. Lock based protocol
2. Time-stamp protocol
3. Validation based protocol

Section-B

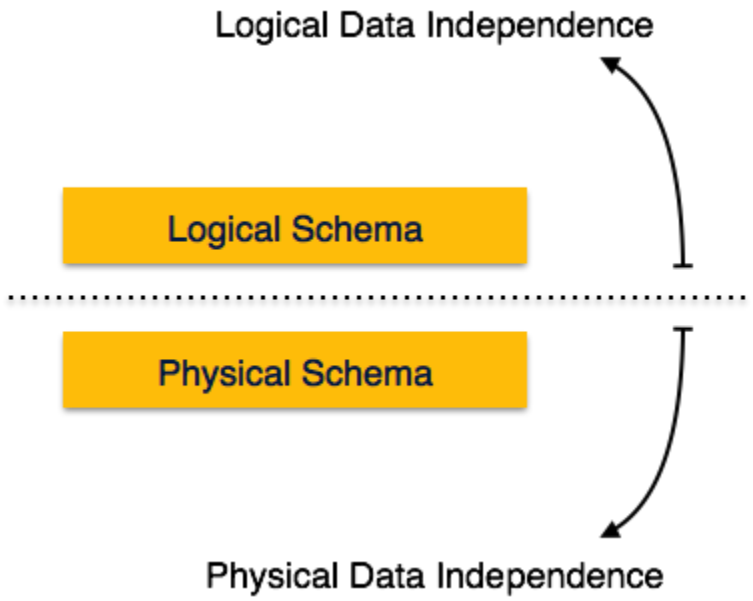
2. Attempt any three of the following:

a) Explain data independence and its types.

If a database system is not multi-layered, then it becomes difficult to make any changes in the database system. Database systems are designed in multi-layers as we learnt earlier.

Data Independence

A database system normally contains a lot of data in addition to users' data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.



Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation. Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

Physical Data Independence

All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data. For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

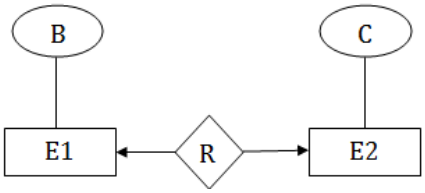
b) Describe mapping constraint with its types.

Mapping Constraints

- A mapping constraint is a data constraint that expresses the number of entities to which another entity can be related via a relationship set.
- It is most useful in describing the relationship sets that involve more than two entity sets.
- For binary relationship set R on an entity set A and B, there are four possible mapping cardinalities. These are as follows:
    1. One to one (1:1)
    2. One to many (1:M)
    3. Many to one (M:1)
    4. Many to many (M:M)

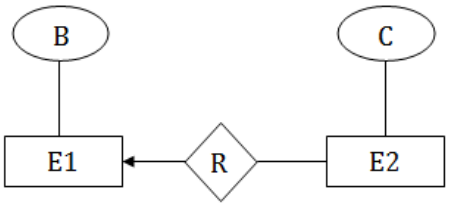
One-to-one

In one-to-one mapping, an entity in E1 is associated with at most one entity in E2, and an entity in E2 is associated with at most one entity in E1.



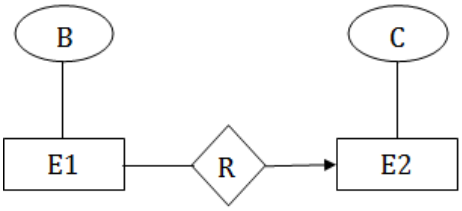
One-to-many

In one-to-many mapping, an entity in E1 is associated with any number of entities in E2, and an entity in E2 is associated with at most one entity in E1.



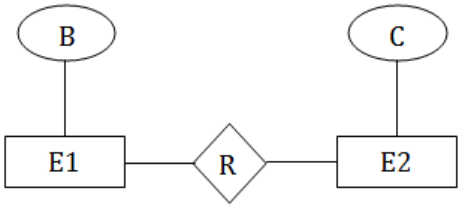
Many-to-one

In one-to-many mapping, an entity in E1 is associated with at most one entity in E2, and an entity in E2 is associated with any number of entities in E1.



Many-to-many

In many-to-many mapping, an entity in E1 is associated with any number of entities in E2, and an entity in E2 is associated with any number of entities in E1.



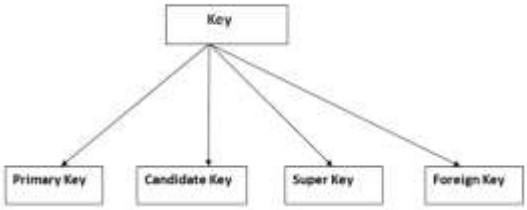
c) Define keys. Explain various types of keys

Keys

- Keys play an important role in the relational database.
- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

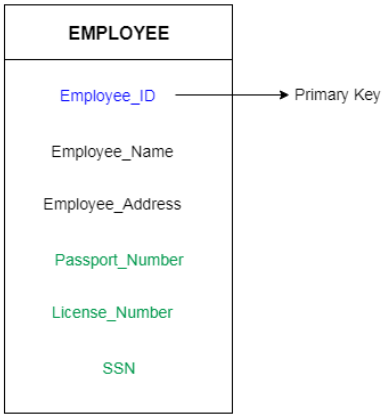
For example: In Student table, ID is used as a key because it is unique for each student. In PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.

Types of key:



1. Primary key

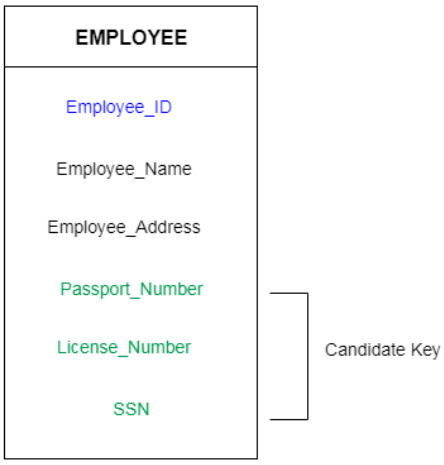
- It is the first key which is used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys as we saw in PERSON table. The key which is most suitable from those lists become a primary key.
- In the EMPLOYEE table, ID can be primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary key since they are also unique.
- For each entity, selection of the primary key is based on requirement and developers.



2. Candidate key

- A candidate key is an attribute or set of an attribute which can uniquely identify a tuple.
- The remaining attributes except for primary key are considered as a candidate key. The candidate keys are as strong as the primary key.

For example: In the EMPLOYEE table, id is best suited for the primary key. Rest of the attributes like SSN, Passport_Number, and License_Number, etc. are considered as a candidate key.



3. Super Key

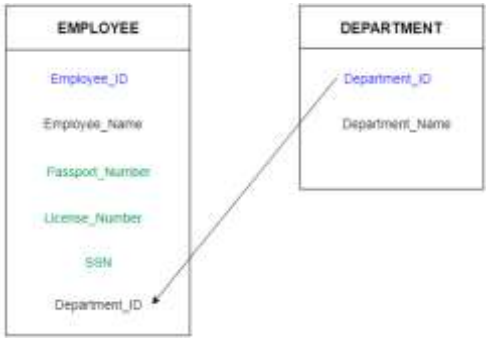
Super key is a set of an attribute which can uniquely identify a tuple. Super key is a superset of a candidate key.

For example: In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME) the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.

The super key would be EMPLOYEE-ID, (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

4. Foreign key

- Foreign keys are the column of the table which is used to point to the primary key of another table.
- In a company, every employee works in a specific department, and employee and department are two different entities. So we can't store the information of the department in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id as a new attribute in the EMPLOYEE table.
- Now in the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.



d) Explain the phantom phenomena. Discuss the timestamp protocol that avoids the phantom phenomena.

The so-called **phantom problem** occurs within a transaction when the same query produces different sets of rows at different times. For example, if a SELECT is executed twice, but returns a row the second time that was not returned the first time, the row is a "**phantom**" row.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

- If W_TS(X) >TS(Ti) then the operation is rejected.
- If W_TS(X) <= TS(Ti) then the operation is executed.
- Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Advantages and Disadvantages of TO protocol:

- TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade- free.

e) What are distributed database? List advantage and disadvantage of data replication and data fragmentation.

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Data Replication

Data replication is the process of storing separate copies of the database at two or more sites. It is a popular fault tolerance technique of distributed databases.

Advantages of Data Replication

- **Reliability** − In case of failure of any site, the database system continues to work since a copy is available at another site(s).
- **Reduction in Network Load** − Since local copies of data are available, query processing can be done with reduced network usage, particularly during prime hours. Data updating can be done at non-prime hours.
- **Quicker Response** − Availability of local copies of data ensures quick query processing and consequently quick response time.

- **Simpler Transactions** − Transactions require less number of joins of tables located at different sites and minimal coordination across the network. Thus, they become simpler in nature.

Disadvantages of Data Replication

- **Increased Storage Requirements** − Maintaining multiple copies of data is associated with increased storage costs. The storage space required is in multiples of the storage required for a centralized system.
- **Increased Cost and Complexity of Data Updating** − Each time a data item is updated, the update needs to be reflected in all the copies of the data at the different sites. This requires complex synchronization techniques and protocols.
- **Undesirable Application – Database coupling** − If complex update mechanisms are not used, removing data inconsistency requires complex co-ordination at application level. This results in undesirable application – database coupling.

Some commonly used replication techniques are −

- Snapshot replication
- Near-real-time replication
- Pull replication

Fragmentation

Fragmentation is the task of dividing a table into a set of smaller tables. The subsets of the table are called **fragments**. Fragmentation can be of three types: horizontal, vertical, and hybrid (combination of horizontal and vertical). Horizontal fragmentation can further be classified into two techniques: primary horizontal fragmentation and derived horizontal fragmentation.

Fragmentation should be done in a way so that the original table can be reconstructed from the fragments. This is needed so that the original table can be reconstructed from the fragments whenever required. This requirement is called “reconstructiveness.”

Advantages of Fragmentation

- Since data is stored close to the site of usage, efficiency of the database system is increased.

- Local query optimization techniques are sufficient for most queries since data is locally available.
- Since irrelevant data is not available at the sites, security and privacy of the database system can be maintained.

Disadvantages of Fragmentation

- When data from different fragments are required, the access speeds may be very high.
- In case of recursive fragmentations, the job of reconstruction will need expensive techniques.
- Lack of back-up copies of data in different sites may render the database ineffective in case of failure of a site.

Vertical Fragmentation

In vertical fragmentation, the fields or columns of a table are grouped into fragments. In order to maintain reconstructiveness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

For example, let us consider that a University database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

Now, the fees details are maintained in the accounts section. In this case, the designer will fragment the database as follows −

```
CREATE TABLE STD_FEES AS
  SELECT Regd_No, Fees
  FROM STUDENT;
```

Horizontal Fragmentation

Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructiveness. Each horizontal fragment must have all columns of the original base table.

For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows −

```
CREATE COMP_STD AS
  SELECT * FROM STUDENT
  WHERE COURSE = "Computer Science";
```

Hybrid Fragmentation

In hybrid fragmentation, a combination of horizontal and vertical fragmentation techniques are used. This is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

Hybrid fragmentation can be done in two alternative ways −

- At first, generate a set of horizontal fragments; then generate vertical fragments from one or more of the horizontal fragments.
- At first, generate a set of vertical fragments; then generate horizontal fragments from one or more of the vertical fragments.

SECTION-C

3. Attempt any one part of the following:

a) Define Join. Explain different types of join.

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

Consider the two tables below:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

The simplest Join is INNER JOIN.

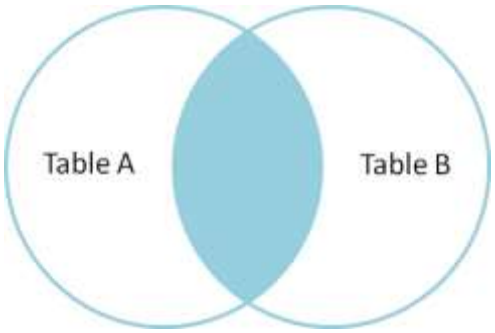
1. **INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.

   Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

   table1: First table.
   table2: Second table
   matching_column: Column common to both the tables.

   Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



Example Queries(INNER JOIN)

- This query will show the names and age of students enrolled in different courses.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

2. **LEFT JOIN**: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

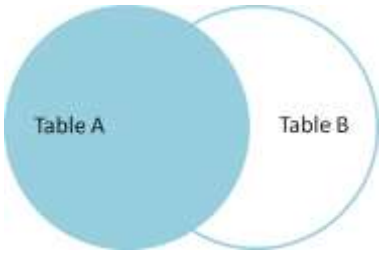
ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.



Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

3. **RIGHT JOIN**: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

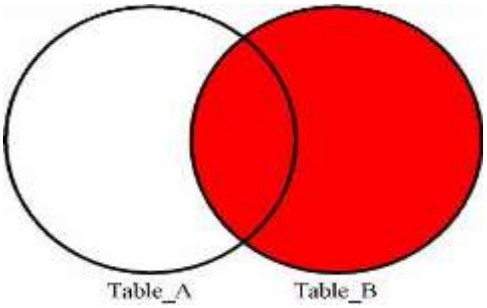
```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are same.



Example Queries(RIGHT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

4. **FULL JOIN:** FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
```

FROM table1

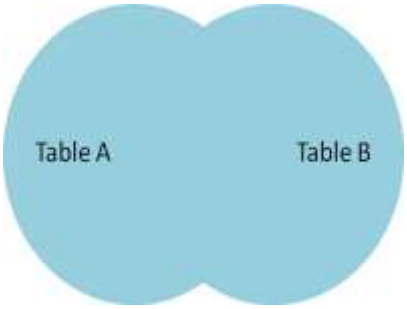
FULL JOIN table2

ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table

matching_column: Column common to both the tables.



Example Queries(FULL JOIN):

SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

FULL JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	9
NULL	10
NULL	11

b) Discuss the following terms (i) DDL command (ii) DML Command

SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. CREATE It is used to create a new table in the database.

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

Example:

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

b. DROP: It is used to delete both the structure and record stored in the table.

Syntax

DROP TABLE ;

Example

DROP TABLE EMPLOYEE;

c. ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);

EXAMPLE

ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

d. TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE EMPLOYEE;

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. INSERT: The INSERT statement is a SQL query. It is used to insert data into the row of a table.

Syntax:

```
INSERT INTO TABLE_NAME
(col1, col2, col3,.... col N)
VALUES (value1, value2, value3, .... valueN);
```

Or

```
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3, .... valueN);
```

For example:

```
INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");
```

b. UPDATE: This command is used to update or modify the value of a column in the table.

Syntax:

UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITI ON]

For example:

```
UPDATE students
SET User_Name = 'Sonoo'
WHERE Student_Id = '3'
```

c. DELETE: It is used to remove one or more row from a table.

Syntax:

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
DELETE FROM javatpoint
WHERE Author="Sonoo";
```

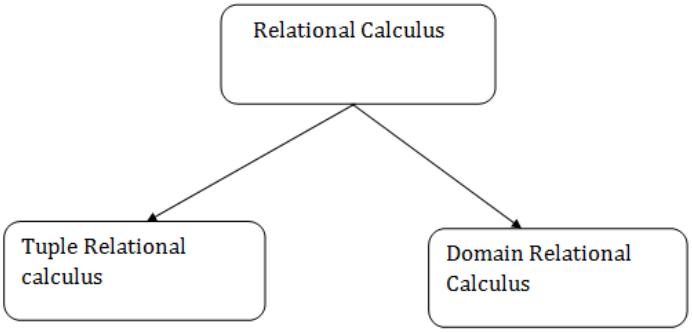
4. **Attempt any one part of the following:**

a) What is tuple relational calculas and domain relational calculas?

Relational Calculus

- Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results.
- The relational calculus tells what to do but never explains how to do.

Types of Relational calculus:



1. Tuple Relational Calculus (TRC)

- The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation.
- The result of the relation can have one or more tuples.

Notation:

{T | P (T)} or {T | Condition (T)}

Where

T is the resulting tuples

P(T) is the condition used to fetch T.

For example:

{ T.name | Author(T) AND T.article = 'database' }

OUTPUT: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

For example:

{ R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output: This query will yield the same result as the previous one.

2. Domain Relational Calculus (DRC)

- The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes.
- Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not).
- It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.

Notation:

{ a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where

a1, a2 are attributes
P stands for formula built by inner attributes

For example:

{< article, page, subject > | ∈ javatpoint ∧ subject = 'database'}

Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

b) Describe the following terms: (i) Multivalued dependency (ii) Trigger

Multivalued Dependency

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.

- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.

Example: Suppose there is a bike manufacturer company which produces two colors(white and black) of each model every year.

BIKE_MODEL	MANUF_YEAR	COLOR
M2011	2008	White
M2001	2008	Black
M3001	2013	White
M3001	2013	Black
M4006	2017	White
M4006	2017	Black

Here columns COLOR and MANUF_YEAR are dependent on BIKE_MODEL and independent of each other.

In this case, these two columns can be called as multivalued dependent on BIKE_MODEL. The representation of these dependencies is shown below:

1. BIKE_MODEL → → MANUF_YEAR
2. BIKE_MODEL → → COLOR

This can be read as "BIKE_MODEL multidetermined MANUF_YEAR" and "BIKE_MODEL multidetermined COLOR".

Trigger

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).

- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
```

```
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
```

```
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Trigger created.

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

Old salary:
New salary: 7500
Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

Old salary: 1500
New salary: 2000
Salary difference: 500

Ques5(a): What do you mean by ACID properties of a transaction? Explain in details.

Ans 5(a): Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** − This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.
- **Consistency** − A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database.
- **Isolation** − A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.
- **Durability** − If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

Example: Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as

Ti: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

- *Consistency:* The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints.

- *Atomicity:* Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, wenote that the sum A + B is no longer preserved. Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such astate an inconsistent state. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account A is $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are. The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled bya component called the transaction-management component.
- Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.
- Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

Ques:5(b): Discuss about Deadlock Prevention Scheme.

Ans:5(b): Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

Ques:6Attempt any one part of the following.

(a) What is Concurrency Control? Why it is needed in database system?

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Need of Concurrency control in database system

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

(b). Consider the following relational DATABASE. Give an expression in SQL for each of the following queries. Underline records are primary key.

employee (**person name**, street, city)

works (**person name**, company-name, salary)

company (**company name**, city)

manages (**person name**, manager-name)

Give an expression in SQL for each of the following queries:

(b).

i) **Find the names of all employees who works for the ABC Bank**

select employee. person name from employee, works

where employee. person name= company. person name and company name=’ABC’;

ii) **Find the names of all employees who live in the same city and on the same street as do their managers.**

select p. person name from employee p, employee r, manages m

where p. person name = m. person name and m. manager name = r. person name

and p. street = r. street and p.city = r.city;

iii) **Find the names, street address, and cities of residence for all employees who work for ‘ABC Bank ' and earn more than 7,000 per annum.**

select e. person name, employee. Street, employee. City from employee, works

where e. person name =works. person name and company name = 'ABC Bank' and salary > 7000);

iv) **Find the names of all employees in the database who earn more than every employee of XYZ.**

select person name from works

where salary > all (select salary from works where company name =’XYZ’;

v) **Give all employees of First Bank Corporation a 7 percent.**

update works set salary = salary *1 .07

where company name = ’ABC’;

vi) **Delete all tuples in the works relation for employees of ABC**

delete works

where company-name = 'ABC';

vii) **Find the names of all employees in the database who live in the same cities as the companies for which they work**.

select e. person name from employee e, works w, company c

where e. person name = w. person name and e. city = c. city and and w. company name = c. company name;

Ques:7:

(a)Explain Directory System.

Directory Systems

In the pre computerization days, organizations would create physical directories of employees and distribute them across the organization. In general, a directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement. In the world of physical telephone directories, directories that satisfy lookups in the forward direction are called white pages, while directories that satisfy lookups in the reverse direction are called yellow pages

Directory Access Protocols

Several directory access protocols have been developed to provide a standardized way of accessing data in a directory. The most widely used among them today is the Lightweight Directory Access Protocol (LDAP).

The reasons for using a specialized protocol for accessing directory information:

• First, directory access protocols are simplified protocols that cater to a limited type of access to data. They evolved in parallel with the database access protocols.

• Second, and more important, directory systems provide a simple mechanism to name objects in a hierarchical fashion, similar to file system directory names, which can be used in a distributed directory system to specify what information is stored in each of the directory servers.

LDAP: Lightweight Directory Access Protocol

A directory system is implemented as one or more servers, which service multiple clients. Clients use the application programmer interface defined by directory system to communicate with the directory servers. Directory access protocols also define a data model and access control.

The X.500 directory access protocol, defined by the International Organization for Standardization (ISO), is a standard for accessing directory information. However, the protocol is rather complex, and is not widely used. The Lightweight Directory Access Protocol (LDAP) provides many of the X.500 features, but with less complexity, and is widely used.

(b) Give the following queries in the relational algebra using the relational schema

Student (id, name)

enrolledIn (id, code)

subject (code, lecturer)

i). What are the names of students enrolled in cs3020?

Π name(cs3020=code(student $\bowtie$ enrolledIn))

ii). Which subjects is Hector taking?

code(σ name=Hector(student $\bowtie$ enrolledIn))

iii). Who teaches cs1500?

lecturer(σ code=cs1500(subject))

iv). Who teaches cs1500 or cs3020?

lecturer(σ code=cs1500 OR code=cs3020(subject))

v). Who teaches at least two different subjects?

π lecturer(R.lecturer = S.lecturer AND R.code < > S.code($R \bowtie S$))

vi). What are the names of students in cs1500 or cs3010?

π name(σ code=cs1500(student $\bowtie$ enrolledIn)) π name(σ code=cs3010(student $\bowtie$ enrolledIn))

vii). What are the names of students in both cs1500 and cs1200?

π name(code=cs1500(student $\bowtie$ enrolledIn)) $\cap$ π name(code=cs3010(student $\bowtie$ enrolledIn))