Index Korth

Chapter 15 Concurrency Control

Chapter 16 Recovery System

Red => Important

Black => Less Important

Introduction to DBMS

- ***Data*** is the raw and isolated facts about an entity(record).
- ***Information*** is processed, useful or meaningful data.
- ***Database*** is the collection of similar/inter-related data relevant for an enterprise.

Database Management System (DBMS)

- It is the software which is used to create, retrieve, manipulate and delete data from a database.
- It's primary goal is to manage data *conveniently* and *efficiently*.
- Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information among various users.
- Also, DBMS should ensure data security against system crashes and unauthorized data access.

Applications

- Enterprise Information
  - Sales: customers, products, purchases
  - Accounting: payments, receipts, assets
  - Human Resources: Information about employees, salaries, payroll taxes.
- Manufacturing: management of production, inventory, orders, supply chain.
- Banking and finance
  - customer information, accounts, loans, and banking transactions.
  - Credit card transactions
  - Finance: sales and purchases of financial instruments (e.g., stocks and bonds; storing real-time market data
- Universities: registration, grades
- Airlines: reservations, schedules
- Telecommunication: records of calls, texts, and data usage, generating monthly bills, maintaining balances on prepaid calling cards
- Web-based services
  - Online retailers: order tracking, customized recommendations
  - Online advertisements
- Document databases
- Navigation systems: For maintaining the locations of varies places of interest along with the exact routes of roads, train systems, buses, etc.

File Processing System

It is conventional storing mechanism which stores permanent records in various files, and different application programs are required to extract, add, delete appropriate records and files. They were used for small firms or where data is very less.

Disadvantages of File Processing System (Advantages or Need of Database Management Systems)

1. ***Data redundancy:*** Same information may be duplicated in several places in different files.

2. ***Data consistency****:* Various copies of the same data may no longer agree i.e. modification to data in a file does not agree that all it's copies will be modified accordingly.

3. ***Difficulty in accessing data****:* This system do not allow needed data to be retrieved in a convenient and efficient manner. It needs to write a new program to carry out each new task.

4. ***Data isolation****:* Since data is scattered in various files and in different formats, writing new application programs to retreive

appropriate data is difficult.

5. ***Data security problems****:* Not every user of database system should be able to access or modify all the data, but applying such security restrictions in this system is difficult.

6. ***Atomicity Problems****:* If any system failure occurs, then data should be restored to the consistent data prior to the failure. Data modification should be *atomic* in nature, i.e. it must happen entirely or not at all. This is difficult to ensure in such file systems.
   Example: Transfer of funds from one account to another should either complete or not happen at all.

7. ***Concurrent - Access Anomalies****:* Many users may need to access the data concurrently which may lead to data inconsistency. Hence different application programs need to be in coordination to avoid any anomalies and system must maintain supervision in some form. This supervision by system is difficult to implement.
   Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time

8. ***Integrity Problems****:* Data values stored in database must satisfy certain *consistency constraints*. Developers enforce these constraints by adding appropriate code in various application programs. However, it is difficult to change the programs to enforce the addition of new constraints. Integrity constraints (e.g., account balance > 0) become *buried* in program code rather than being stated explicitly.

Differences between Database Management Systems and File-Processing Systems

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 deleting some information or updating some information.

Disadvantage of DBMS

- **Size:** It occupies large disk space and large memory to run efficiently.
- **Cost:** DBMS requires a high-speed data processor and larger memory to run DBMS software, so it is costly.
- **Complexity:** DBMS creates additional complexity and requirements.

View of Data

A major purpose of a database system is to provide users with an abstract view of the data.

- ***Data abstraction:*** It is hiding the complexity of data structures to represent data in the database from users through three levels of abstraction, to achieve *data independence*. Data Independence means users and data should not directly interact with each other. The user should be at a different level and the data should be present at some other level.

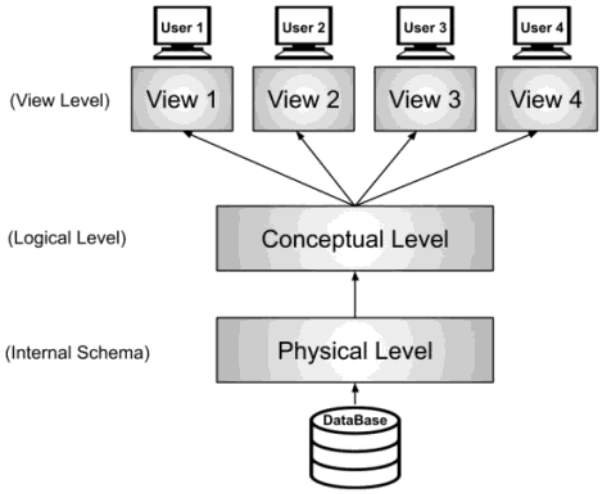
  a. **Physical Level:** It is the lowest level of abstraction which describes *how* data records are actually stores. It describes the complex low-level data structures.

  b. **Logical Level:** It is the higher level of abstraction which describes *what* data are stored in the database and the relationships among them in a simple structure. It is used by the *database administrators* (who decides what information to keep in database).

     This simple structure of logical level comprises of complex physical level structures which need to be abstracted from users of logical level. This is known as ***physical data independence***. Any change in the physical location of tables and indexes should not affect the conceptual level or external view of data. This data independence is easy to achieve and implemented by most of the DBMS.

  c. **View Level:** It is the highest level of abstraction which describes only part of the entire databse. Many users do not need all of the information stored in database (at logical level) like details of data types. View level can also hide information from users for security purposes like employee's salary. Thus view level helps in simple interaction between users and system, which may vary from user to user.

     The data at conceptual level schema and external level schema must be independent, this is known as ***conceptual data independence***. This means a change in conceptual schema should not affect external schema. e.g.; Adding or deleting attributes of a table should not affect the user's view of the table. But this type of independence is difficult to achieve as compared to physical data independence because the changes in conceptual schema are reflected in the user's view.



Levels of Data Abstraction

- ***Data models***: It is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical and view levels.

  Types of Data Models

  a. **Relational Model:** It uses a collection of tables to represent both data and relationships among data. Each table, known as *relations*, has multiple columns with unique names known as *attributes*, and entries in rows known as *tuples*.

  b. **Entity Relationship(ER) Model:** It uses a collection of basic real world objects (indistinguishable from other objects) called *entities* and *relationships* among these objects.

c. **Object Based Data Model:** It is extension of ER model with features like encapsulation, functions and object entity. Thus it a combination of object-oriented data model and relational data model.

d. **Semistructured Data Model:** It permits the specification of data where individual data items of the same tuple may have different sets of attributes. Eg) *XML* (Extensible Markup Language)

- ***Database Schema & Database Instance***

  The collection of information stored in the database at a particular moment is called an ***database instance***. Thus it is the snapshot of the database at a particular moment. It is also called the *database state* i.e. whenever we insert, delete or modify the value of the data item in the record, databases changes from one state or instance to other.

  The overall design of the database is called the ***database schema***. It displays the record types(entity), names of data items(attribute) and the relations among the files. It defines how the data is organized and how the relations among them are associated.

  Contents of database instance may change with time as the database is updated but the schema of a database does not generally change. Analogous to a program written in a programming language, database schema corresponds to the variable declarations whereas instances corresponds to the variables' value at particular instant.

  There are several types of database schemas based on the level of abstraction:

  a. **Physical Database Schema** describes the overall *physical* structure of database design i.e. how the data will be actually stored in a secondary storage.

  b. **Logical Database Schema** describes just the *logical* structure (or logical representation) of the database design. This schema defines all the logical constraints that need to be applied on the data stored. For eg in Relational Database, it defines tables, views, and integrity constraints.

  c. **Subschemas** describes the different views of database at the view level. A database may have several schemas at view level. It Identifies subset of areas, sets, records, data names defined in database for a particular user.

  Programmers construct application programs using the logical schema. Since the physical schemea is hidden behind the logical schema, it cannot be changed without affecting the application programs. Hence, application programs exhibit ***physical data independence*** if they do not depend on physical schema and need not be rewritten if physical schema changes.

Important Note:

Based on the type of data model used like relational model or E-R model, database schema can be of various types like ***Relational Database Schema***, E-R database schema, etc.

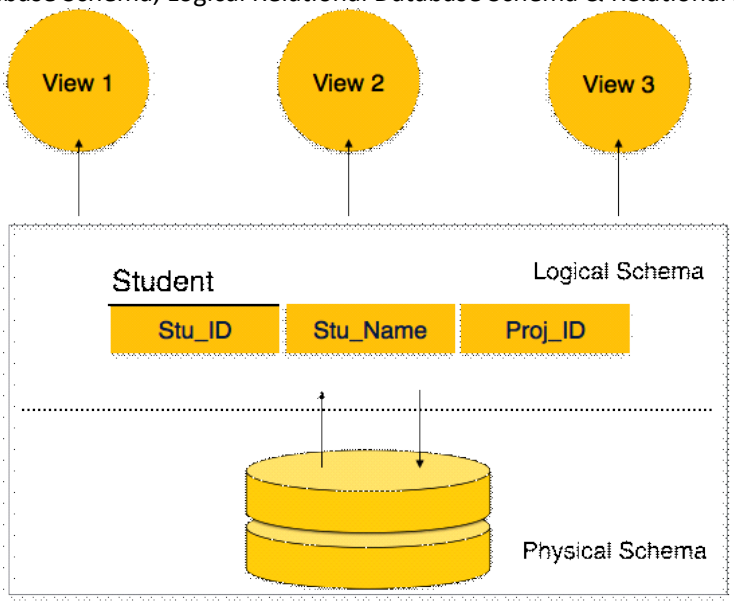
Similarly, database instance can be of various types based on data model used like ***Relational Database Instance***, E-R Database Instance, etc.

- schema: *instructor* (*ID, name, dept_name, salary*)
- Instance:

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

Relational Database Schema vs Relational Database Instance

For Relational Database Schema, the several types of database schemas based on level of abstraction are Physical Relational Database Schema, Logical Relational Database Schema & Relational Subschemas:



Types of Relational Database Schema based on Level of Abstraction

Important Note:

There is a difference between ***Relation Schema & Database Schema*** (or Relation Instance & Database Instance)
{ With Respect To Relational Model }

Relation schema: A set of table attributes is called a relation schema. Relation schema is also known as table schema. Relation schema defines name of "a" table, set of its column names (attributes) , the data types associated with each column.

Example)
Emp (emp_id, emp_name, age)

Database schema: A database schema is the collection of "one or more" relation schemas for a whole database. Database schema is really a collection of meta-data.

Example)
Emp (emp_id, emp_name, age)
Dept (d_id, d_name)

- If the database consists of only one relation (only one table), then the relation schema and the database schema

represents same.

Database Languages

1. **Data Definition Language (DDL)**

DDL is used to specify a database schema by a set of definitions and additional properties of the data. DDL gets some input instructions (or statements) and generates some output.

This output is placed in ***data dictionary,*** which is a special type of table that contains *metadata* (data about data), and can only be accessed and updated by database system itself.

DDL provides facilities to specify certain ***consistency (or integrity) constraints*** to the data values (like account balance > 0):

a. *Domain Constraints*: Domain of possible values must be associated with every attribute (like integer types, character types, data/time types, etc).

a. *Referential Integrity*: It ensures that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation.

b. *Assertions*: They are conditions that the database must always satisfy. It includes domain constraints and referential integrity constraints also. For eg) "Every Department must have 5 courses in each semester."

a. *Authorization*: It is the differentiation among the users for the type of access permissions on various data values. For Eg) read authorization, insert authorization, update authorization and delete authorization.

*Note**: Data Storage & Definition Language (SDL)*** is a special type of DDL whch specify the storage structure and access methods used by the database system by a set of statements.

2. **Database Manipulation Language (DML)**

DML is a language for accessing and updating the data organized by the appropriate data model. It is used for retreival, insertion, deletion and modification of information in the database.

Since DML is used for data retreival, It is also known as ***data query language (DQL)***. Query language is a language in which user request information from the database. Query language are higher level language than standard programming languages.

There are two types of DML:

a. ***Procedural DML***: It require a user to specify *what* data are needed and *how* to get those data. The user instructs the system to perform a sequence of operations on the database to compute the desired result.
Example) ***Relational Algebra***: It consists of a set of operations that take collection of relations as input and produce a single relation as their result.

a. ***Declarative DML (or Non-Procedural DML)***: It require a user to specify what data are needed without specifying the procedure of how to get those data.
Example) ***Relational Calculus*** (Tuple Relational Calculus and Domain Relational Calculus): It uses predicate logic to define the result without giving any specific algebraic procedure for obtaining that result.

Note: Query languages used today include elements of both procedural and non-procedural approaches. The languages which still follow only either procedural or declarative DML are known as ***pure query languages***. Example) Relational Algebra, Tuple Relational Calculus and Domain Relational Calculus.

Database Design

The process of designing the general structure of the database:

a. ***Logical Design***: Deciding on the database schema. Database design requires that we find a "good" collection of relation schemas.
   i) Business decision: What attributes should we record in the database?
   ii) Computer Science decision: What relation schemas should we have and how should the attributes be distributed among the various relation schemas?

b. ***Physical Design***: Deciding on the physical layout of the database

A database system is partitioned into modules or ***subsystems*** that deal with each of the responsibilities of the overall system. The functional components of a database system can be divided into:

1. **Storage Manager**

- A program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data
- The storage manager components include:
  - Authorization and integrity manager
  - Transaction manager
  - File manager
  - Buffer manager
- The storage manager implements several data structures as part of the physical system implementation:
  - Data files -- store the database itself
  - Data dictionary -- stores metadata about the structure of the database, in particular the schema of the database.
  - Indices -- can provide fast access to data items. A database index provides pointers to those data items that hold a particular value.

1. **Query Processor**

The query processor components include:

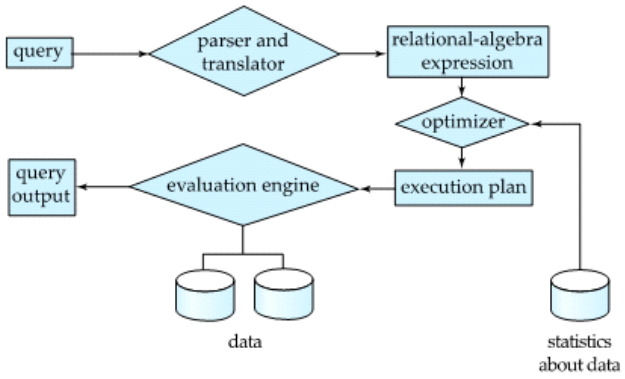
- DDL interpreter -- interprets DDL statements and records the definitions in the data dictionary.
- DML compiler -- translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
  - The DML compiler performs query optimization; that is, it picks the lowest cost evaluation plan from among the various alternatives.
- Query evaluation engine -- executes low-level instructions generated by the DML compiler.

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



2. Transaction Management

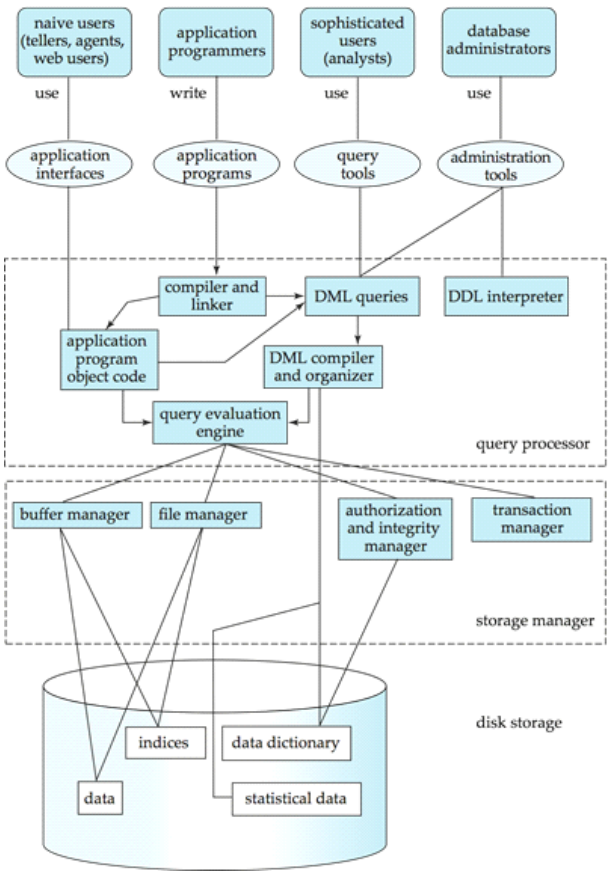
- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Architecture

Different Types of architecture of a database system are:

- Centralized databases
  - One to a few cores, shared memory
- Client-server,
  - One server machine executes work on behalf of multiple client machines.
- Parallel databases
  - Many core shared memory
  - Shared disk
  - Shared nothing
- Distributed databases
  - Geographical distribution
  - Schema/data heterogeneity

Note: For a database system on a ***network*** of connected users, ***client machines*** are those on which remote databases work while ***server machines*** are those on which the database system runs.



Database System Architecture Structure

Database Applications are partitioned into two or three parts:

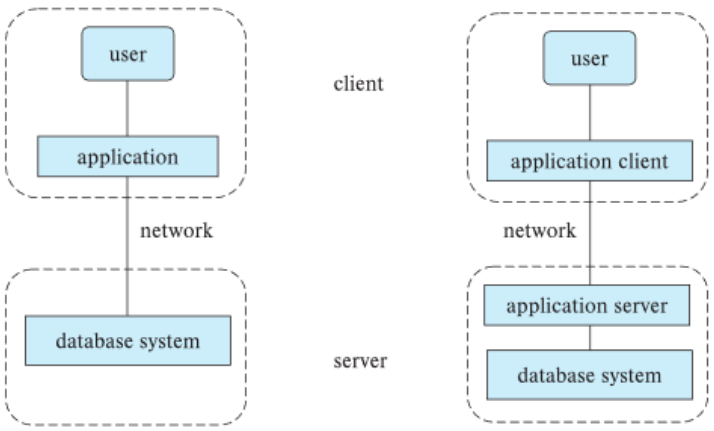
- ***Two-tier architecture***
  - The application resides at the client machine, where it invokes database system functionality at the server machine.
  - Two tier architecture is similar to a basic ***client-server model***. The application at the client end directly communicates

with the database at the server side. API's are used for this interaction.
  - The server side is responsible for providing query processing and transaction management functionalities. On the client side, the user interfaces and application programs are run. The application on the client side establishes a connection with the server side in order to communicate with the DBMS.
  - An advantage of this type is that maintenance and understanding is easier, compatible with existing systems. However this model gives poor performance when there are a large number of users.

- ***Three-tier architecture***
  - In this type, there is another layer between the client and the server. The client machine acts as a front end and does not contain any direct database calls.
  - The client does not directly communicate with the server. Instead, it interacts with an application server which further communicates with the database system and then the query processing and transaction management takes place.
  - This intermediate layer acts as a medium for exchange of partially processed data between server and client. Three-tier architecture is more appropriate for large applications that run on World Wide Web.

Advantages:
  - Enhanced scalability due to distributed deployment of application servers. Now,individual connections need not be made between client and server.
  - Data Integrity is maintained. Since there is a middle layer between client and server, data corruption can be avoided/removed.
  - Security is improved. This type of model prevents direct interaction of the client with the server thereby reducing access to unauthorized data.

Disadvantages:
Increased complexity of implementation and communication. It becomes difficult for this sort of interaction to take place due to presence of middle layers.



(a) Two-tier architecture (b) Three-tier architecture

Difference Between Two-Tier And Three-Tier Database Architecture

S.NO	Two-Tier Database Architecture	Three-Tier Database Architecture
1	It is a Client-Server Architecture.	It is a Web-based application.
2	In two-tier, the application logic is either buried inside the user interface on the client or within the database on the server (or both).	In three-tier, the application logic or process resides in the middle-tier, it is separated from the data and the user interface.
3	Two-tier architecture consists of two layers : Client Tier and Database (Data Tier).	Three-tier architecture consists of three layers : Client Layer, Business Layer and Data Layer.
4	It is easy to build and maintain.	It is complex to build and maintain.

5	Two-tier architecture runs slower.	Three-tier architecture runs faster.
6	It is less secured as client can communicate with database directly.	It is secured as client is not allowed to communicate with database directly.
7	It results in performance loss whenever the users increase rapidly.	It results in performance loss whenever the system is run on Internet but gives more performance than two-tier architecture.
8	Example – Contact Management System created using MS-Access or Railway Reservation System, etc.	Example – Designing registration form which contains text box, label, button or a large website on the Internet, etc.

Database Users & Administrators

People who work with a database are categorized as database users or database administrators.

Database Users

There are various types of users on the way they interact with the database system:

a. *Naïve User*: Unsophisticated users who interact with system by invoking one of the application programs that have been already written.

b. *Application Programmers*: They write application programs and develop user interfaces using tools known as ***Rapid Application Development (RAID)*** tools.

c. *Sophisticated Users*: They interact with the system without writing programs. They interact using the database query languages or data analysis softwares.

a. *Specialized Users*: Sophisticated users who write database applications that do not fit into traditional data-processing framework like computer-aided design systems, systems storing data with complex data types, etc.

Database Administrator (DBA)

A person who have central control of both the data and the programs that access those data over the system is called a database administrator.

Functions of a DBA include:

- Schema definition
- Storage structure and access-method definition
- Schema and physical-organization modification
- Granting of authorization for data access
- Routine maintenance
- Periodically backing up the database
- Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required
- Monitoring jobs running on the database

OLAP vs OLTP

Online Analytical Processing (OLAP)
Online Analytical Processing consists of a type of software tools that are used for data analysis for business decisions. OLAP provides an environment to get insights from the database retrieved from multiple database systems at one time.

Examples – Any type of Data warehouse system is an OLAP system. Uses of OLAP are as follows:

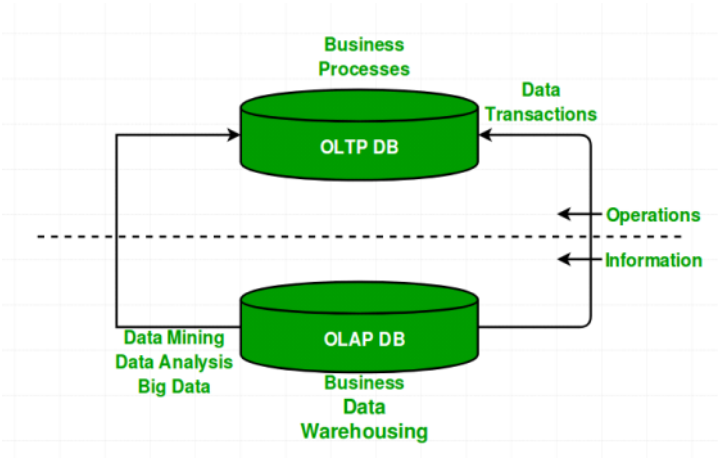
- Spotify analyzed songs by users to come up with the personalized homepage of their songs and playlist.
- Netflix movie recommendation system.

Online transaction processing (OLTP)

Online transaction processing provides transaction-oriented applications in a 3-tier architecture. OLTP administers day to day transaction of an organization.

Examples – Uses of OLTP are as follows:

- ATM center is an OLTP application.
- OLTP handles the ACID properties during data transaction via the application.
- It's also used for Online banking, Online airline ticket booking, sending a text message, add a book to the shopping cart.



Comparisons of OLAP vs OLTP –

OLAP (Online analytical processing)	OLTP (Online transaction processing)
Consists of historical data from various Databases.	Consists only operational current data.
It is subject oriented. Used for Data Mining, Analytics, Decision making,etc.	It is application oriented. Used for business tasks.
The data is used in planning, problem solving and decision making.	The data is used to perform day to day fundamental operations.
It reveals a snapshot of present business tasks.	It provides a multi-dimensional view of different business tasks.
Large amount of data is stored typically in TB, PB	The size of the data is relatively small as the historical data is archived. For ex MB, GB
Relatively slow as the amount of data involved is large. Queries may take hours.	Very Fast as the queries operate on 5% of the data.
It only need backup from time to time as compared to OLTP.	Backup and recovery process is maintained religiously
This data is generally managed by CEO, MD, GM.	This data is managed by clerks, managers.
Only read and rarely write operation.	Both read and write operations.

Relational Database Model

- A ***relational database*** consists of a collection of tables each with a unique name. It is "relational" because the values within each table are related to each other.
- Since a table is a collection of relationships, it is also known as a ***relation***.
- Row (or ***record***) in a table represents a ***relationship*** among a set of values. A ***tuple*** is a sequence or list of values. The order in which tuples appear in a relation is irrelevant because relation is a "*set*" of tuples. A relationship between n values is represented mathematically by an ***n-tuple*** of values, i.e. a tuple of n values, which corresponds to a row in a table.
- ***Relation instance*** refers to a specific instance of a relation i.e. containing a specific set of rows, whereas ***Relation Schema*** consists of a list of attributes and their corresponding domains.
- Each table has multiple columns with unique names known as ***attributes*** or ***fields***.
- For each attribute of a relation, there is a set of permitted values called the ***domain*** of that attribute. For all relations, domains of all attributes should be made ***atomic***. A domain is atomic if elements of the domain are considered to be indivisible units. The most common method for defininig domain of an attribute is defining datatype of the column.
- **Arity/Degree** refers to the numer of columns in the table whereas **cardinality** refers to the number of rows in the table.
- **Null Value**: It is a special value that signifies that the value is *unknown* or does not exist. Null values can cause various difficulties when we access or update database, which should be eliminated.

- $A_1, A_2, \ldots, A_n$ are *attributes*

- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

  Example:

  $instructor = (ID, name, dept\_name, salary)$

- Formally, given sets $D_1, D_2, \ldots D_n$ a **relation *r*** is a subset of

  $D_1 \times D_2 \times \ldots \times D_n$

  Thus, a relation is a set of *n*-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

Properties of Relational tables:

- Cells contains atomic values
- Values in a column are of the same kind
- Each row is unique
- No two tables can have the same name in a relational schema.
- Each column has a unique name
- The sequence of rows is insignificant
- The sequence of columns is insignificant.

Keys

Need of Keys

- Keys help you to identify any row of data in a table. In a real-world application, a table could contain thousands of records. Moreover, the records could be duplicated. Keys ensure that you can uniquely identify a table record despite these challenges.
- Allows you to establish a relationship between and identify the relation between tables
- Help you to enforce identity and integrity in the relationship.

Types of Keys

1. **Candidate Key:**
   - The minimal set of attribute which can uniquely identify a tuple is known as candidate key.
   - The value of Candidate Key is unique and non-null for every tuple.
   - There can be more than one candidate key in a relation.
   - The candidate key can be simple (having only one attribute) or composite as well.
   - There should be atleast one candidate key with NOT NULL constraint.

2. **Super Key:**
   - The set of attributes which can uniquely identify a tuple is known as Super Key.
   - Let X be a set of attributes in a relation R, if closure of X (X^+) determines all attributes of R, then X is said to be super key of R.
   - Adding zero or more attributes to candidate key generates super key. Hence, a candidate key is a super key but vice versa is not true always.
   - There should be atleast one super key with NOT NULL constraint.
   - If there are n attributes, then there can be atmax (2^n -1) super keys possible (all subset of attributes except empty subset). The biggest super key contains all attributes of the relation.

3. **Primary Key:**
   - Primary key is a candidate key that is most appropriate to become the main key for any table.
   - It is chosen by the database administrator (DBA). That candidate key should be chosen as primary key whose attributes are never or rarely changed.
   - It is a key that can uniquely identify each record in a table.
   - The primary key field cannot be null i.e. not null field values.
   - Two rows can't have the same primary key value i.e. unique field values.
   - The value in a primary key column can never be modified or updated if any foreign key refers to that primary key.

Primary Key ⊆ Candidate Key ⊆ Super Key

4. **Composite & Compound Key:**
   - Key that consists of two or more attributes that uniquely identify any record in a table is called Composite key.
   - But the attributes which together form the Composite key are not a key independentely or individually, i.e. individually uniqueness is not guaranteed.
   - The *difference* between compound and the composite key is that any part of the compound key can be a foreign key, but the composite key may or maybe not a part of the foreign key.

5. **Secondary or Alternate Key**
   - The candidate key which are not selected as primary key are known as secondary keys or alternative keys. They are used to speed up the search and retreival.

6. **Foreign Key**
   - Foreign key is a column that creates a relationship between two tables.
   - The purpose of Foreign keys is to maintain ***data referential integrity*** and allow navigation between two different instances of an entity.
   - It acts as a cross-reference between two tables as it references the primary key of another table.
   - The relation which is being referenced is called ***referenced relation*** and the corresponding attribute is called ***referenced attribute***.
   - The relation which refers to the referenced relation is called ***referencing relation*** and the corresponding attribute is called ***referencing attribute***.
   - The referenced attribute of the referenced relation should be the primary key for it.
   - Foreign Key can be NULL as well as may contain duplicate tuples i.e. it need not follow *uniqueness constraint*.

- **Non-Key Attributes**
  - Non-key attributes are the attributes or fields of a table, other than candidate key attributes/fields in a table.

- **Prime Attributes & Non-Prime attributes**
  - Prime attributes are attributes present in the primary key.
  - Non-prime Attributes are attributes other than Primary Key attribute(s).

Difference Between Primary key & Foreign key

Primary Key	Foreign Key
Helps you to uniquely identify a record in the table.	It is a field in the table that is the primary key of another table.

Primary Key never accept null values.	A foreign key may accept multiple null values.
Primary key is a clustered index and data in the DBMS table are physically organized in the sequence of the clustered index.	A foreign key cannot automatically create an index, clustered or non-clustered. However, you can manually create an index on the foreign key.
You can have the single Primary key in a table.	You can have multiple foreign keys in a table.

Referential Integrity

Referential integrity is a relational database concept, which states that table relationships must always be consistent. In other words, whenever a foreign key value is used it must reference a valid, existing primary key in the parent table. Thus, any primary key field changes (updation or deletion) must be applied to all foreign keys, or not at all. The same restriction also applies to foreign keys that any insertions or updates (not deletions) must be propagated to the primary parent key.

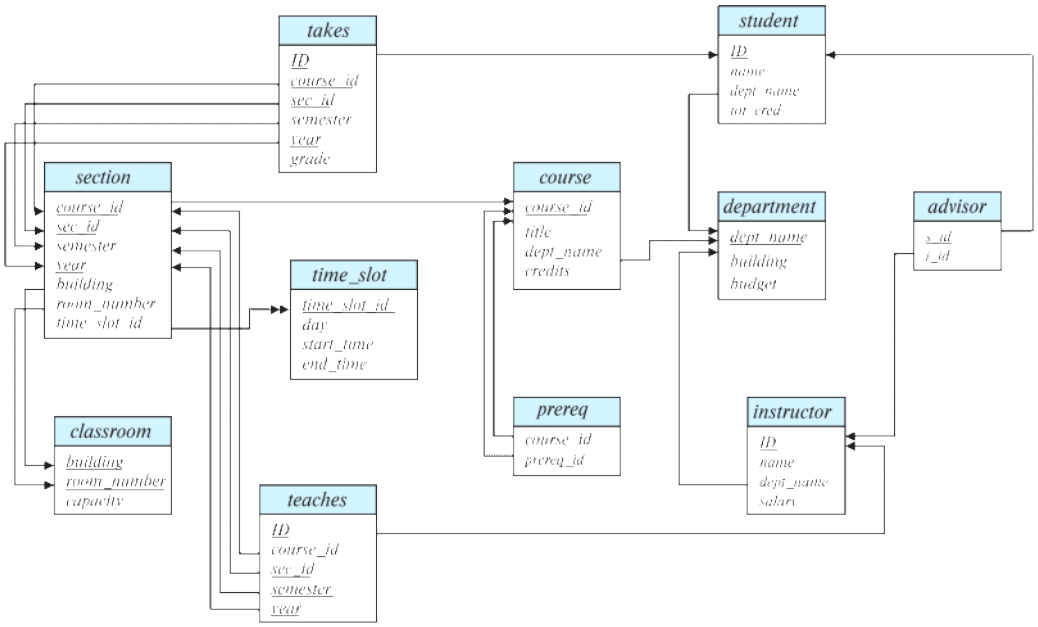
For example, suppose Table B has a foreign key that points to a field in Table A. Referential integrity would prevent you from adding a record to Table B that cannot be linked to Table A. In addition, the referential integrity rules might also specify that whenever you delete a record from Table A, any records in Table B that are linked to the deleted record will also be deleted. This is called ***cascading delete***. Finally, the referential integrity rules could specify that whenever you modify the value of a linked field in Table A, all records in Table B that are linked to it will also be modified accordingly. This is called ***cascading update.***

So referential integrity will prevent users from:

- Adding rows to a related table if there is no associated row in the primary table.
- Changing values in a primary table that result in *orphaned records* in a related table.
- Deleting rows from a primary table if there are matching related rows.

Schema Diagram (For Relational Model)

- Schema diagram is a graphical representation of a database schema along with primary key and foreign key dependencies.
- Each relation appears as a box with the relation name at top and attributes listed inside box.
- Primary key attributes are shown underlined.
- Foreign key dependencies appear as arrows from foreign key attributes of referencing relation to the primary key of the referenced relation.
- Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams (They are shown in E-R diagrams representation).



Example of Schema Diagram

Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries. It consists of a set of operations that take collection of relations as input and produce a single relation as their result.

Select, Project, Union, Set Difference, Cartesian Product and Rename are *basic/fundamental* relational operations while the joins(inner and outer), intersection and division are *derived* relational operations as they can be implemented using combination of one or more fundamental operations.

Types of Relational Operations

1. **Select (σ)**

   The SELECT operation is used for selecting a subset of the tuples according to a given selection condition.

   $\sigma_p(r)$

   σ is the predicate

   r stands for relation which is the name of the table

   p is prepositional logic

   Example 1

   ```
   σ topic = "Database" (Tutorials)
   ```

   Output - Selects tuples from Tutorials where topic = 'Database'.

   Example 2

   ```
   σ topic = "Database" and author = "guru99"( Tutorials)
   ```

   Output - Selects tuples from Tutorials where the topic is 'Database' and 'author' is guru99.

   - We allow comparisons using

     $=, \neq, >, \geq, <, \leq$

     in the selection predicate.
   - We can combine several predicates into a larger predicate by using the connectives:

     $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
   - Example: Find the instructors in Physics with a salary greater $90,000, we write:

     $\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90{,}000}(instructor)$

   - The select predicate may include comparisons between two attributes.
     - Example, find all departments whose name is the same as their building name:
     - $\sigma_{dept\_name=building}$ (*department*)

2. **Projection (π)**
   This helps to extract the values of specified attributes(columns) and discards other columns.
   Duplicate rows are removed from result, since relations are sets.

   Example of Projection:

   Consider the following table

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

   Here, the projection of CustomerName and status will give

```
Π CustomerName, Status (Customers)
```

3. **Assigment (<-) Operation**

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by ← and works like assignment in a programming language.
- Example: Find all instructor in the "Physics" and Music department.

$$Physics \leftarrow s_{dept\_name="Physics"}(instructor)$$

$$Music \leftarrow s_{dept\_name="Music"}(instructor)$$

$$Physics \cup Music$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

4. **Rename (ρ or r)**
   Rename is a unary operation used for renaming attributes of a relation.

- The results of relational-algebra expressions do not have a name that we can use to refer to them.
- The rename operator, r, is provided for that purpose
- The expression:

$$r_x(E)$$

  returns the result of expression E under the name x

- Another form of the rename operation:

$$r_{x(A1,A2,..An)}(E)$$

5. **Set Union (υ)**
   It includes all tuples that are in tables A or in B. It also eliminates duplicate tuples.
   For a union operation to be valid, the following conditions must hold -

- Number of attributes in both tables must be the same.
- Attribute domains need to be compatible.
  (For example: 2nd column of A deals with the same type of values as does the 2nd column of B)
- Duplicate tuples should be automatically removed.

Example

Consider the following tables.

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

A ∪ B gives

Table A ∪ B	
column 1	column 2
1	1
1	2
1	3

6. **Set Difference (-)**
   The result of A - B, is a relation which includes all tuples that are in A but not in B.
   The attribute name of A has to match with the attribute name in B.
   The two-operand relations A and B should be either Union compatible (Same Number of Attributes & Compatible)
   It should be defined relation consisting of the tuples that are in relation A, but not in B.

Example

```
A-B
```

Table A - B	
column 1	column 2
1	2

7. **Set Intersection (∩)**
   It defines a relation consisting of a set of all tuple that are in both A and B.
   However, A and B must be union-compatible.

Example:

```
A ∩ B
```

Table A ∩ B	
column 1	column 2
1	1

8. **Cartesian Product (X)**
   Cartesian Product is an operation used to merge columns from two relations.
   Generally, a cartesian product is never a meaningful operation when it performs alone.
   However, it becomes meaningful when it is followed by other operations.
   It is also called *Cross Product* or *Cross Join*.

   Example – Cartesian product

   $\sigma_{\text{column 2} = \text{'1'}} (A \times B)$

   Output – The above example shows all rows from relation A and B whose column 2 has value 1

σ column 2 = '1' (A X B)	
column 1	column 2
1	1
1	1

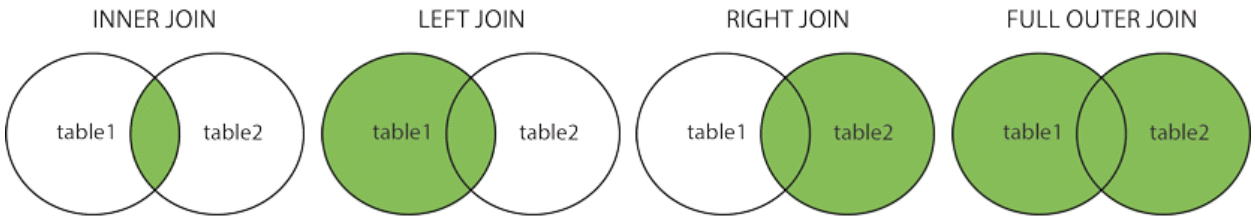
9. **Division Operator (÷)**
   Division operator A÷B can be applied if and only if:
   - Attributes of B is proper subset of Attributes of A.
   - The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)

   The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

10. **Join (⋈)**
    Join operation is essentially a cartesian product followed by a selection criterion i.e.

    JOIN = CARTESIAN PRODUCT + CONDITION

    JOIN operation also allows joining variously related tuples from different relations.



Types of Join

a. **Inner Join**
   In an inner join, only those tuples that satisfy the matching criteria are included, while the rest are excluded.

i. Theta Join (θ)

The general case of JOIN operation is called a Theta join. It is denoted by symbol θ.

Example

$$A \bowtie_{\theta} B$$

Theta join can use any conditions in the selection criteria.

For example:

$$A \bowtie_{A.column\ 2 > B.column\ 2} (B)$$

A ⋈ A.column 2 > B.column 2 (B)	
column 1	**column 2**
1	2

- Let "theta" be a predicate on attributes in the schema R "union" S. The join operation r $\bowtie_{\theta}$ s is defined as follows:

$$r \bowtie_{\theta} s = \sigma_{\theta} (r \times s)$$

- Thus

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches))$$

- Can equivalently be written as

$$instructor \bowtie_{Instructor.id = teaches.id} teaches.$$

ii. Equi Join

When a theta join uses only equivalence condition, it becomes a equi join.

Note: EQUI join is the most difficult operations to implement efficiently using SQL in an RDBMS and one reason why RDBMS have essential performance problems.

For example:

$$A \bowtie_{A.column\ 2 = B.column\ 2} (B)$$

A ⋈ A.column 2 = B.column 2 (B)	
column 1	**column 2**
1	1

iii. Natural Join

Natural join can only be performed if there is a common attribute (column) between the relations.
The name and type of the attribute must be same.
If there is no common attribute, then natural join is same as cartesian product.

Example

Consider the following two tables

C	
Num	**Square**
2	4
3	9

D	
Num	**Cube**
2	8
3	27

```
C ⋈ D
```

C ⋈ D		
Num	**Square**	**Cube**
2	4	4
3	9	27

Q) What is the difference between Natural Join, Equi Join and Theta Join ?

R) A *theta join* allows for arbitrary comparison relationships (such as ≥).
An *equijoin* is a theta join using the equality operator only.
A *natural join* is an equijoin on attributes that have the same name in each relationship (table).

b. Outer Join

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

i. Left Outer Join

In the left outer join, operation allows keeping all tuple in the left relation.
However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.

Example)

A	
Num	Square
2	4
3	9
4	16

B	
Num	Cube
2	8
3	18
5	75

A ⟖ B

A ⟖ B		
Num	Square	Cube
2	4	4
3	9	9
4	16	-

ii. Right Outer Join

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.

A ⟗ B

A ⟗ B		
Num	Cube	Square
2	8	4
3	18	9
5	75	-

iii. Full Outer Join

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.

A ⋈ B

A ⋈ B		
Num	Cube	Square
2	4	8
3	9	18
4	16	-
5	-	75

11. Aggregate Functions

These functions take a collection of values as input and return a single value as the result. Some of the Aggregate Functions are as follows :

- SUM : Takes a collection of values and return the SUM of those values.
- AVG : Takes a collection of values and return the AVERAGE of those values.
- COUNT : Takes a collection of values and return the number of those values in the collection.
- MIN, MAX : Takes a collection of values and return the MINIMUM and MAXIMUM values in a collection.

For example : Find the total sum of salaries of all Female Employees in an Organization.

Query : G (SUM(salary)) (Female_employees)

Note: Here G is the letter G in calligraphic font. It is used to show that aggregation is to be applied and its subscript specifies the aggregate operation to be applied.

To eliminate the duplicates, DISTINCT keyword is used.

For example : Count the number of branches in a university.

Gcount-distinct (BranchName) (StudentsDetails)

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000

  Query 1: $\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90{,}000}(instructor)$

  Query 2: $\sigma_{dept\_name=\text{“Physics”}}(\sigma_{salary > 90.000}(instructor))$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

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**

- 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 = 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**

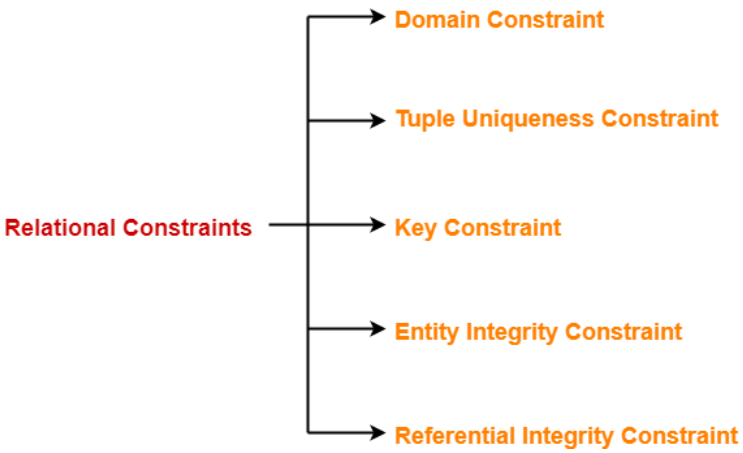
- The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes.
- Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not).
- It uses Existential (∃) 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.

Integrity Constraints



Refer Article:
https://www.gatevidyalay.com/constraints-in-dbms-types-of-constraints-in-dbms/
https://www.gatevidyalay.com/referential-integrity-constraint-violation/

1. **Domain Constraint**

Domain constraint defines the domain or set of values for an attribute.
It specifies that the value taken by the attribute must be the atomic value from its domain.

2. **Tuple Uniqueness Constraint**

Tuple Uniqueness constraint specifies that all the tuples must be necessarily unique in any relation.

3. **Key Constraint**

Key constraint specifies that in any relation:
All the values of primary key must be unique.
The value of primary key must not be null.

4. **Entity Integrity Constraint**

Entity integrity constraint specifies that no attribute of primary key must contain a null value in any relation.
This is because the presence of null value in the primary key violates the uniqueness property.

5. **Referential Integrity Constraint**

This constraint is enforced when a foreign key references the primary key of a relation.
It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.

The following two important results emerges out due to referential integrity constraint:

- We can not insert a record into a referencing relation if the corresponding record does not exist in the referenced relation.
- We can not delete or update a record of the referenced relation if the corresponding record exists in the referencing relation.

Handling Violation of Referential Integrity Constraint

To ensure the correctness of the database, it is important to handle the violation of referential integrity constraint properly.

There are following three possible causes of violation of referential integrity constraint-

Cause-01: Insertion in a referencing relation

It is allowed to insert only those values in the referencing attribute which are already present in the value of the referenced attribute.
Inserting a value in the referencing attribute which is not present in the value of the referenced attribute violates the referential integrity constraint.

Cause-02: Deletion from a referenced relation
It is not allowed to delete a row from the referenced relation if the referencing attribute uses the value of the referenced attribute of that row.
Such a deletion violates the referential integrity constraint.

Handling the Violation: The violation caused due to a deletion from the referenced relation can be handled in the following three ways:

Method-01: This method involves simultaneously deleting those tuples from the referencing relation where the referencing attribute uses the value of referenced attribute being deleted. This method of handling the violation is called as On Delete Cascade.

Method-02: This method involves aborting or deleting the request for a deletion from the referenced relation if the value is used by the referencing relation.

Method-03: This method involves setting the value being deleted from the referenced relation to NULL or some other value in the referencing relation if the referencing attribute uses that value.

Cause-03: Updation in a referenced relation
It is not allowed to update a row of the referenced relation if the referencing attribute uses the value of the referenced attribute of that row.
Such an updation violates the referential integrity constraint.

Handling the Violation: The violation caused due to an updation in the referenced relation can be handled in the following three ways:

Method-01: This method involves simultaneously updating those tuples of the referencing relation where the referencing attribute uses the referenced attribute value being updated. This method of handling the violation is called as On Update Cascade.

Method-02: This method involves aborting or deleting the request for an updation of the referenced relation if the value is used by the referencing relation.

Method-03: This method involves setting the value being updated in the referenced relation to NULL or some other value in the referencing relation if the referencing attribute uses that value.

SQL - Introduction

Structured Query Language (SQL)

- SQL query language is non-procedural language. A query takes as input several tables (possibly only one) and always returns a single table.

- SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

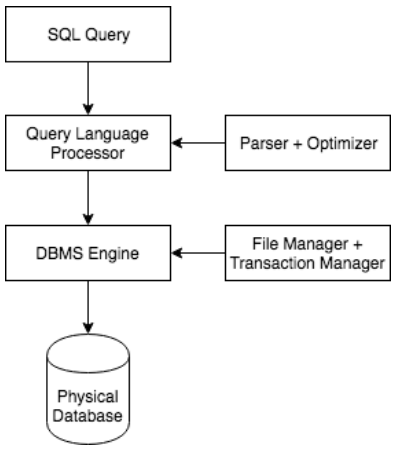
- *Difference between SQL and MySQL*:
  SQL is a standard language for retrieving and manipulating structured databases. On the contrary, MySQL is a relational database management system, like SQL Server, Oracle or IBM DB2, that is used to manage SQL databases.

- SQL does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a ***host language***, such as C/C++, Java or Python, with embedded SQL queries that access the data in the database. To be able to compute complex functions, SQL is usually embedded in some higher-level language.

- Application programs generally access databases through either language extensions to allow ***embedded SQL*** or ***Application program interface (API)*** which allow SQL queries to be sent to a database.

SQL Process

- When a SQL command is executed in any RDBMS, the system determines the best way to carry out your command and SQL engine figures out how to interpret the task.
- Components like *Query Dispatcher*, *Optimization Engines*, *Classic Query Engine*, *SQL Query Engine* etc are involved in this process.
- Note: A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files.



***SQL Syntax*:**

- SQL keywords are **NOT case sensitive**: select is the same as SELECT.
- Some database systems require a **semicolon** at the end of each SQL statement. Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.
- There are two types of **comments** in SQL, single line comments (two dashes --) and multi line comments (/* */ same as C++).

Applications or Commands in SQL

- 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

Types of SQL Commands

a. **DDL(Data Definition Language) :** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.
Examples of DDL commands:
   - CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
   - DROP – is used to delete objects from the database.
   - ALTER-is used to alter the structure of the database.
   - TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.
   - COMMENT –is used to add comments to the data dictionary.
   - RENAME –is used to rename an object existing in the database.

b. **DML(Data Manipulation Language) :** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
Examples of DML:
   - SELECT - retrieves records from one or more tables
   - INSERT - creates a record
   - UPDATE - modifies records
   - DELETE - deletes records
   - MERGE - UPSERT operation (insert or update)
   - CALL - call a PL/SQL or Java subprogram
   - EXPLAIN PLAN - interpretation of the data access path
   - LOCK TABLE - concurrency control

c. **DCL(Data Control Language) :** DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
Examples of DCL commands:
   - GRANT - gives user's access privileges to database.
   - REVOKE - withdraw user's access privileges given by using the GRANT command.

d. **TCL(Transaction Control Language) :** TCL commands deals with the transaction within the database.
Examples of TCL commands:
   - COMMIT – commits a Transaction.
   - ROLLBACK – rollbacks a transaction in case of any error occurs.
   - SAVEPOINT – sets a savepoint within a transaction.
   - SET TRANSACTION – specify characteristics for the transaction.

Select Clause

- Chose columns/attributes { Projection (π) in Relational Algebra }
- The data returned is stored in a result table, called the ***result-set***.
- Syntax:

  1.a) For selecting certain columns

```
SELECT column1, column2, ...
FROM table_name;
```

  1.b) For selecting all columns

```
SELECT * FROM table_name;
```

- **SELECT DISTINCT**: Used to display rows with *unique* set of selected column attributes.
  Syntax:

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

- **SELECT TOP:** Used to specify number of records to return in query i.e. it limit the number of records to display. It is used when there are thousands of records and it is better in terms of *performance* to display some of the records matching condition.
  Syntax:
  a. SQL Server, MS Access

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

  b. MySQL

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

  c. Oracle

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

- **SELECT INTO**
  - Copies data(row(s) and column(s)) from an existing table to a new table.
  - Syntax:
    i. Copy all Columns and all Rows:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable;
```

    ii. Copy Some Columns and all Rows:

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable;
```

    iii. Copy all Columns and some Rows:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
```

```
WHERE condition;
```

iv. Copy some Columns and some Rows:

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

- Example:

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

- Names of columns in new table will be same as in old table. For different column names, we can use aliases (AS keyword).

Q) Create a new empty table(no rows) with columns same as another table(already existing) i.e. copy only the schema of table. (IMPORTANT)

R) We can use (iii.) Syntax of SELECT INTO with condition which will be false always (like 1 = 0).
Syntax:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE 1 = 0;
```

Where Clause

WHERE Clause

- Chose rows/tuples/records { Selection (σ) in Relational Algebra } on the specified condition
- Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- WHERE clause is not mandatory to be present in the query. If there is no where clause, i.e. no condition on rows, then all rows will be displayed.
  Syntax: (To display all rows)

```
SELECT column1, column2, ...
FROM table_name
```

- *Note*: WHERE can also be used with UPDATE, DELETE, and other clauses.
- There are various **operators** supported with WHERE clause:
  - =, <, >, <=, >=, <> (!= in Some Versions)
  - AND, OR, NOT
    Syntax:

```
WHERE condition1 AND condition2;
WHERE condition1 OR condition2;
WHERE NOT condition;
```

  - **BETWEEN** & **NOT BETWEEN** operator
    - Used with WHERE clause to select column with values in given range (both inclusive).
    - It can be used with Numbers, Text (Lexicographical Range), and Dates.
    - Syntax:

```
SELECT column_name(s) FROM table_name
WHERE column_name BETWEEN | NOT BETWEEN value1 AND value2;
```

    - Example:

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;

SELECT * FROM Countries
WHERE CountryName BETWEEN 'INDIA' AND 'USA';

SELECT * FROM Orders
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

  - **LIKE** operator
    - Used with WHERE Clause for searching specific patterns in values in a column.
    - ***Wildcard*** characters are used with the LIKE operator. A wildcard character is used to substitute one or more characters in a string.

Symbol	Description	Example
* (or %)	Represents zero or more characters	bl* finds bl, black, blue, and blob
? (or _)	Represents a single character	h?t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
[!] (or [^])	Represents any character not in the brackets	h[!oa]t finds hit, but not hot and hat
[-]	Represents a range of characters	c[a-b]t finds cat and cbt
#	Represents any single numeric character	2#5 finds 205, 215, 225, 235, 245, 255, 265, 275, 285, and 295

- Syntax of LIKE operator

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE 'String';
```

- Examples of LIKE operator

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are *at least* 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

- **IN** & **NOT IN** operators
  - Used with WHERE clause as shorthand notation for *multiple OR conditions with = (or <>) operator*.
  - Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN | NOT IN (value1, value2, ...);
```

OR

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN | NOT IN (SELECT NESTED STATEMENT);
```

  - Example:

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');

SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

- **EXISTS** & **NOT EXISTS** operators
  - Used to test for the existence of any record in a subquery.
  - Returns true if the subquery returns one or more records otherwise false (for 0 record).
  - Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS | NOT EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

A) Difference between EXISTS (& NOT EXISTS) vs IN (& NOT IN)?

Exists	IN
It is used with ***correlated queries***.	It is used with ***nested queries***.
EXISTS: This cannot compare the values between the sub-query query and parent query.	IN: It can compare the values between sub-query and parent queries.
The output of EXISTS can be either FALSE or TRUE	The output of IN can be TRUE or NULL or FALSE
EXISTS is used to determine if any values are returned or not.	Whereas, IN can be used as a multiple OR operator.
If the sub-query result is large, then EXISTS is faster than IN.	If the sub-query result is less, then IN is faster than EXISTS
Once the single positive condition is met in the EXISTS	In the IN-condition SQL Engine compares all the values

condition then the SQL Engine will stop the process.

- **ANY** & **ALL** operators
  - Used with a WHERE or HAVING clause.
  - ***ANY***: returns true if any of the subquery values meet condition.
  - ***ALL***: returns true if all of the subqueries values meet condition.
  - Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY | ALL
(SELECT column_name FROM table_name WHERE condition);
```

  - Example:

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

  - *operator* used must be only standard comparison operator (=, >, <, >=, <=, <>).

Group By, Having & Order By

ORBER BY Clause

- Sort the result-set (rows) in ascending or descending order based on a column(s).
- If multiple columns are used in ORDER BY, rows are first sorted by first column, then rows with equal first column values are sorted by second column, and so on.
- Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

- Rows are sorted in ASC (ascending order) by *default* (If not mentioned).
- If we need to order rows in different orders like in ASC order for first column, but DESC by second column, then we can use ASC or DESC keyword after each column name.
  Example)

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 ASC, column2 DESC, ...;
```

GROUP BY Clause

- The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".
- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING group_by_condition;
```

- *Important Note*: Columns after SELECT can only be columns which are also after GROUP BY or other columns in aggregate functions(SUM, COUNT, AVG, MIN, and MAX) only.
  Example:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

  Is valid as Country is in columns after GROUP BY and other column (CustomerID) is in aggregate function (COUNT)

```
SELECT CustomerID, Country
FROM Customers
GROUP BY Country;
```

  Is not valid as CustomerID is not in columns after GROUP BY.

  If we need to print columns other than from GROUP BY columns and not as aggregate functions but the column values itself, we will have to use nested statements:

```
SELECT CustomerID
FROM Customers
WHERE  Country IN
(
    SELECT Country
    FROM Customers
    GROUP BY Country
    HAVING group_by_condition
);
```

HAVING Clause

- The HAVING Clause enables you to specify conditions that filter which group results appear in the results.
- If HAVING clause is used (condition on group is applied), there must be GROUP BY clause preceded by it i.e. there can be GROUP BY clause without HAVING clause (no condition on groups) but HAVING clause without GROUP BY clause is not possible.
- Also, if ORDER BY and HAVING clause both are used, ORDER BY must be preceded by HAVING clause (which is in turn preceded by GROUP BY clause).
- Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING group_by_condition
ORDER BY column_name(s);
```

- Example:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

A) What are the differences between WHERE clause and HAVING clause?

B) WHERE Clause is used to filter the records from the table or used while joining more than one table.Only those records will be extracted who are satisfying the specified condition in WHERE clause.

HAVING Clause is used to filter the records from the groups based on the given condition in the HAVING Clause. Those groups who will satisfy the given condition will appear in the final result.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause i.e.

HAVING condition = WHERE condition on Groups (exclusively for GROUP BY clause)

WHERE condition cannot be used in place of HAVING condition as WHERE condition is placed on all rows irrespective of any group.

SR.NO	WHERE Clause	HAVING Clause
1.	WHERE Clause is used to filter the records from the table based on the specified condition.	HAVING Clause is used to filter record from the groups based on the specified condition.
2.	WHERE Clause can be used without GROUP BY Clause	HAVING Clause cannot be used without GROUP BY Clause
3.	WHERE Clause implements in row operations	HAVING Clause implements in column operation
4.	WHERE Clause cannot contain aggregate function	HAVING Clause can contain aggregate function
5.	WHERE Clause can be used with SELECT, UPDATE, DELETE statement.	HAVING Clause can only be used with SELECT statement.
6.	WHERE Clause is used before GROUP BY Clause	HAVING Clause is used after GROUP BY Clause
7.	WHERE Clause is used with single row function like UPPER, LOWER etc.	HAVING Clause is used with multiple row function like SUM, COUNT etc.

Insert Into, Update, Delete

INSERT INTO Clause

- Insert new record/row in a table
- Syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

  Note: If we specify the column names, then order of columns in statement can be different from actual table. However, values should correspond to written order of columns.
  For Eg) This will work:

```
INSERT INTO table_name (column3, column1, column2, ...)
VALUES (value3, value1, value2, ...);
```

- If we want to add a new row by giving all values i.e. no values is left NULL, then we can omit writing the column names. However, in such a case, values should be written same as that in the order of columns in table.
  Syntax:

```
INSERT INTO table_name
VALUES (value1, value2, value3, …, valueN);
```

  Note: If we change the order of values (if column names are not given), then it will be logically incorrect as values will correspond to wrong columns. ("logically" incorrect because statement might still work if datatypes will be same).
  For Eg) This statement will be logically incorrect:

```
INSERT INTO table_name
VALUES (value3, value1, value2, …);
```

  Note: If we don't want to enter any particular column's value, i.e. leave it NULL, we can explicitly write NULL in statement otherwise (if we omit writing it) order will get disturbed in this case.

- For **AUTO INCREMENT** fields, we need not insert any value as it will get generated automatically.

Auto Increment Field

  - Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.
  - Often this is the primary key field that we would like to be created automatically every time a new record is inserted.
  - By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record. To increment with given value and by given factor use `AUTO INCREMENT(starting_value, increment_factor);`
  - It is used in CREATE TABLE statement.
  - Example: MySQL => *column_name datatype* `AUTO INCREMENT`
  - In SQL Server, AUTO INCREMENT is replaced by keyword `IDENTITY` but the functionality is same.

- **INSERT INTO SELECT**
  - Copies data(some or all rows) from one existing table and inserts it into another existing table.
  - Syntax:

```
INSERT INTO table2 (columni, columnj, columnk, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

  - Necessary requirement should be that selected columns (in order) should have same datatypes. Hence, `columni` and `column1` should be of same datatypes, `columnj` and `column2` should be of same datatypes, and so on.
  - Original content (rows if any) of the table in which data will be copied will remain as it i.e. new data(rows of table from which data is copied) will get ***appended.***

  - *Important Note*: Difference between SELECT INTO and INSERT INTO SELECT:
    - INSERT INTO SELECT inserts into an existing table.
    - SELECT INTO creates a new table and puts the data in it.

UPDATE Clause

- Used to modify exisiting records/rows in a table
- Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- In WHERE condition we select rows whose column(s) values need to be updated. If WHERE is not used with UPDATE, all rows will be updated.

DELETE Clause

- Delete existing records/rows from a table
- Syntax:

```
DELETE FROM table_name WHERE condition;
```

- The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted. Although all the rows will get deleted, but table structure, attributes and indexes will remain intact i.e. there wi ll be empty table(no rows) with all columns as it is.

  Note: DELETE Clause without WHERE clause is similar to TRUNCATE clause as both delete all records but not the table itself. Differences covered in DDL commands (in TRUNCATE command).

Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types
- Date and time Data types

1. Binary Datatypes

Data Type	Description
binary	Maximum length of 8000 bytes (Fixed-Length binary data)
varbinary	Maximum length of 8000 bytes(Variable Length binary data)
varbinary(max)	Maximum length of 231 bytes (SQL Server 2005 only). (Variable Length binary data)
image	Maximum length of 2,147,483,647 bytes(Variable Length binary data)

2. Numeric Datatypes

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	922,337,203,685,477.5808
smallmoney	-214,748.3648	214,748.3648

3. Approximate Numeric Datatype

Data Type	From	To
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

4. Character String Datatype

Data Type	Description
char	Maximum length of 8000 characters. (Fixed-Length non-Unicode Characters)
varchar	Maximum length of 8000 characters. (Variable-Length non-Unicode Characters)
varchar(max)	Maximum length of 231 characters (SQL Server 2005 only). (Variable Length non-Unicode data)
text	Maximum length of 2,147,483,647 characters (Variable Length non-Unicode data)

5. Data & Time Datatype

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.	

User Defined Datatypes in SQL

create type construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2) final**

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

Domain in SQL

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Large Object Datatypes

Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:

- **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

Aliases in SQL

ALIAS (**AS** Keyword or **Rename** in Relational Algebra)

- SQL aliases are used to give a table, or a column in a table, a temporary name.
- Aliases are often used to make column names more readable.
- An alias only exists for the duration of the query.
- Uses or Application of Aliases:
  - There are more than one table involved in a query
  - Functions are used in the query
  - Column names are big or not very readable
  - Two or more columns are combined together

- Alias Column Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

- Alias Table Syntax:

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

- *Note*: It requires double quotation marks or square brackets if the alias name contains spaces.
  Example:

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person] | 'Contact Person'
FROM Customers;
```

- To combine columns and *print* them as one column, we use **CONCAT** function and **AS** alias together.
  Syntax:

```
SELECT CONCAT(column1, column2, …) AS merged_column_name
FROM table_name;
```

  Example:

```
SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address
FROM Customers;
```

- Aliases are used with JOIN in two or more tables to give shorter names to the tables, and especially when both tables joined are same (self join).
  Example:
  Without Alias:

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerID=Orders.CustomerID;
```

  With Alias:

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerID=o.CustomerID;
```

SQL Functions

Syntax:

```
SELECT FUNCTION_NAME(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Will return number of rows matching the condition.
If WHERE clause is not used, it will return the count of all rows i.e. number of records present in the table.

Note: The column name (in the result set) of the sql function will be `FUNCTION_NAME(column_name)` itself. To display different column name, then we can use aliases (AS) keyword with function.
Example:

```
SELECT COUNT(column_name)
AS result_column_name
FROM table_name
WHERE condition;
```

These are some important SQL Functions:

1. String Functions

CONCAT	**Adds two or more expressions together**
INSERT	Inserts a string within a string at the specified position and for a certain number of characters
LENGTH	**Returns the length of a string (in bytes)**
LOWER	Converts a string to lower-case
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string as many times as specified
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REVERSE	Reverses a string and returns the result
STRCMP	**Compares two strings**
SUBSTR	Extracts a substring from a string (starting at any position)
TRIM	Removes leading and trailing spaces from a string
UPPER	Converts a string to upper-case

2. Numeric Functions

ABS	Returns the absolute value of a number
AVG	**Returns the average value of an expression (only int)**
CEIL	Returns the smallest integer value that is >= to a number
COUNT	**Returns the number of records returned by a select query**
FLOOR	Returns the largest integer value that is <= to a number
MAX	**Returns the maximum value in a set of values (both int & string)**
MIN	**Returns the minimum value in a set of values (both int & string)**
MOD	Returns the remainder of a number divided by another number
POW	Returns the value of a number raised to the power of another number
RAND	Returns a random number
ROUND	**Rounds a number to a specified number of decimal places**
SIGN	Returns the sign of a number
SQRT	Returns the square root of a number
SUM	**Calculates the sum of a set of values (only int)**

3. Date Functions

NOW()	**Returns the current date and time**
CURDATE()	**Returns the current date**
CURTIME()	**Returns the current time**
DATE()	Extracts the date part of a date or date/time expression
EXTRACT()	Returns a single part of a date/time
DATE_ADD()	**Adds a specified time interval to a date**
DATE_SUB()	**Subtracts a specified time interval from a date**
DATEDIFF()	Returns the number of days between two dates
DATE_FORMAT()	Displays date/time data in different formats

4. Advanced Functions

CASE	**Goes through conditions and return a value when the first condition is met**
CAST	Converts a value (of any type) into a specified datatype
COALESCE	**Returns the first non-null value in a list**
IF	**Returns a value if a condition is TRUE, or another value if a condition is FALSE**
IFNULL	**Return a specified value if the expression is NULL, otherwise return the expression**
ISNULL	**Returns 1 or 0 depending on whether an expression is NULL**
LAST_INSERT_ID	Returns the AUTO_INCREMENT id of the last row that has been inserted or updated in a table
NULLIF	Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned

NULL Value

- Field with no value (not a space or zero value)
- Can only be assigned to optional fields/attributes (not primary key or NOT NULL attributes)
- Comparison operators like <, >, <=, >=, = don't work with NULL.
- Operators: **IS NULL** (test for NULL value) and **IS NOT NULL** (tests for Non-NULL value) are used.
  Syntax:

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL | IS NOT NULL;
```

- **NULL Functions**
  a. **ISNULL()**
  The ISNULL function have different uses in SQL Server and MySQL.
  In SQL Server, ISNULL() function is used to replace NULL values i.e. return an alternative value if an expression is NULL.

```
SELECT column_names, ISNULL(column_name, value_to_replace)
FROM table_name
WHERE condition;
```

  In MySQL, ISNULL() function is used to test whether an expression is NULL or not. If the expression is NULL it returns TRUE, else FALSE.

```
SELECT column_names
FROM table_name
WHERE ISNULL(column_name);
```

  b. **IFNULL()**
  This function is available in MySQL, and not in SQL Server or Oracle. This function take two arguments. If the first argument is not NULL, the function returns the first argument. Otherwise, the second argument is returned. This function is commonly used to replace NULL value with another value.

```
SELECT column_names, IFNULL(column_name, value_to_replace)
FROM table_name
```

```
WHERE condition;
```

c. **COALESCE()**
COALESCE function in SQL returns the first non-NULL expression among its arguments. If all the expressions evaluate to null, then the COALESCE function will return null.

```
SELECT column_names, COALESCE(expression1, expression2, … expressionN)
FROM table_name
WHERE condition;
```

d. **NULLIF()**
The NULLIF function takes two argument. If the two arguments are equal, then NULL is returned. Otherwise the first argument is returned.

```
SELECT column_names, NULLIF(expression1, expression2)
FROM table_name
WHERE condition;
```

Case Function

- It is similar to switch case in programming languages.
- Returns value for the first true 'WHEN' clause else if all WHEN clauses are false, it returns value for 'ELSE' clause.
- Syntax:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN resultN
  ELSE result
END;
```

- Example 1:

```
SELECT OrderID, Quantity,
CASE
  WHEN Quantity > 30 THEN 'The quantity is greater than 30'
  WHEN Quantity = 30 THEN 'The quantity is 30'
  ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

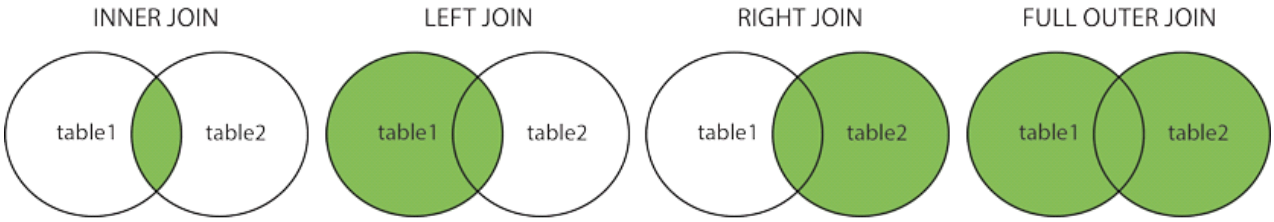
- Example 2:

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
  WHEN City IS NULL THEN Country
  ELSE City
END);
```

Join Clause

Used to combine rows from two or more tables, based on a related column between them.

Types of Joins:



1. **(INNER) JOIN**: Returns records that have matching values in both tables
   Syntax:
   ```
   SELECT column_name(s)
   FROM table1
   INNER JOIN table2
   ON table1.column_name(i) = table2.column_name(j);
   ```

   { Here i and j signify that column_names from both tables need not be same, though they need to be of same datatype and logically similar. }

   Note: To Join 3 Tables:
   ```
   SELECT column_name(s)
   FROM ((table1
   INNER JOIN table2 ON table1.column_name(i) = table2.column_name(j))
   INNER JOIN table3 ON table1.column_name(k) = table3.column_name(l));
   ```

2. **LEFT (OUTER) JOIN**:
   - Returns all records from the left table, and the matched records from the right table.
   - The result is NULL from the right side, if there is no match.
   - The result has all records from the left table even if there are no matches in the right table.

   Syntax:
   ```
   SELECT column_name(s)
   FROM table1
   LEFT JOIN table2
   ON table1.column_name(i) = table2.column_name(j);
   ```

   Note: In some SQL Versions, `LEFT JOIN` is called `LEFT OUTER JOIN`.

3. **RIGHT (OUTER) JOIN**:
   - Returns all records from the right table, and the matched records from the left table.
   - The result is NULL from the left side, when there is no match.
   - The result has all records from the right table even if there are no matches in the left table.

   Syntax:
   ```
   SELECT column_name(s)
   FROM table1
   RIGHT JOIN table2
   ON table1.column_name(i) = table2.column_name(j);
   ```

Note: In some SQL Versions, `RIGHT JOIN` is called `RIGHT OUTER JOIN`.

4. **FULL (OUTER) JOIN**:
   - Returns all records i.e. which are only in left table (with NULL values in columns from right table), which are only in right table (with NULL values in columns from left table) and records with common attributes (like in inner join).
   - FULL OUTER JOIN can potentially return very large result-sets, thus it is not present in some SQL versions like MySQL.

   Syntax:

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name(i) = table2.column_name(j)
WHERE condition;
```

- **Self Join:** It is a normal join but when the tables joined are same, i.e. two aliases of same table are used with JOIN clause. There is no keyword SELF JOIN, it is just a special type of join.

  Syntax:

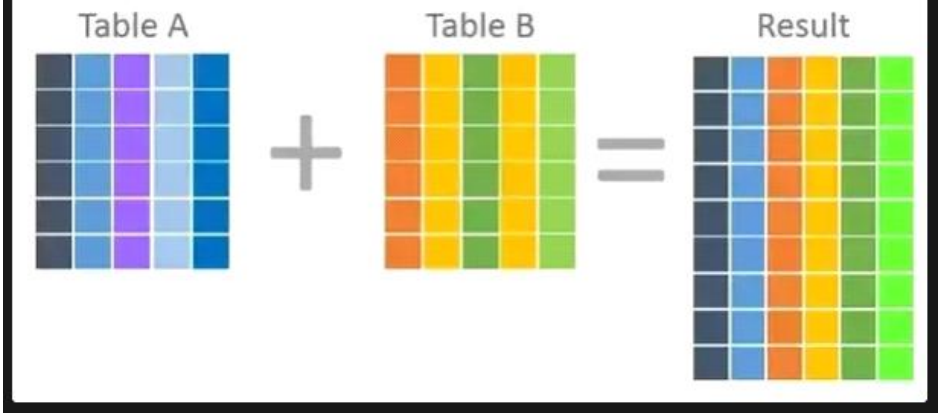
```
SELECT column_name(s)
FROM table_name AS T1, table_name AS T2
ON T1.column_name(i) = T2.column_name(j)
WHERE condition;
```

Q) Difference between Join (especially `FULL OUTER JOIN`) vs Set `UNION` ?

Joins and Unions can be used to combine data from one or more tables. The difference lies in how the data is combined. Join is used to combine columns from different tables, whereas union is used to combine rows.

Joins combine data into new columns— If two tables are joined together, then the data from the first table is shown in one set of column alongside the second table's column in the same row.
The FULL OUTER JOIN keyword returns all records when there is a match in either left (table1) or right (table2) table records.

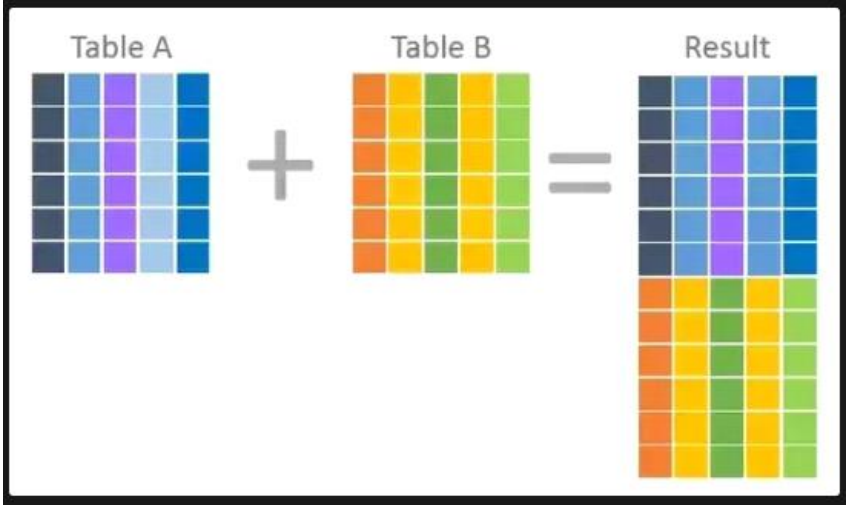
```
1 SELECT column_name(s)
2 FROM table1
3 FULL OUTER JOIN table2 ON table1.column_name = table2.column_name;
```



Unions combine data into new rows— If two tables are "unioned" together, then the data from the first table is in one set of rows, and the data from the second table in another set.
Both tables in Union statement must be *compatible* i.e. must have the same number of columns, having similar data types and in the same order. The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.

UNION Syntax—

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

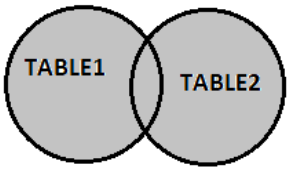


Difference between JOIN and UNION in SQL :

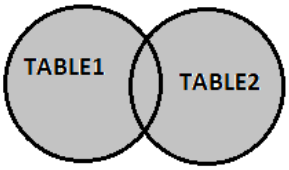
JOIN	UNION
JOIN combines data from many tables based on a matched condition between them.	SQL combines the result-set of two or more SELECT statements.
It combines data into new columns.	It combines data into new rows
Number of columns selected from each table may not be same.	Number of columns selected from each table should be same.
Datatypes of corresponding columns selected from each table can be different.	Datatypes of corresponding columns selected from each table should be same.
It may not return distinct columns.	It returns distinct rows.

Note: **Differences between Inner Join vs Set Intersection** or **Differences between Left (or Right) Outer Join vs Set Difference** can be explained on similar grounds.

Q) What is the difference between `CROSS JOIN` (CARTESIAN PRODUCT) and `FULL OUTER JOIN`?



CROSS JOIN OR CARTESIAN PRODUCT



FULL OUTER JOIN

Both ven diagram looks like the same but internally in SQL they have a different meaning.

- *Definitions*:
  A cross join produces a cartesian product between the two tables, returning all possible combinations of all rows. It has no on clause because it just joining everything to everything.
  VS
  A full outer join is a combination of a left outer and right outer join. When performing an inner join, rows from either table that are unmatched in the other table are not returned.

- *Syntax Difference*:

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2
WHERE condition;
```

VS

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name(i) = table2.column_name(j)
WHERE condition;
```

- Cartesian product (or `CROSS JOIN`) always have rows = M X N whereas the number of records in `FULL OUTER JOIN` can range in [MIN(M, N) , (M + N)].

- For `CROSS JOIN`, there is no `ON` clause specified, whereas the records will be matched using `ON` clause in `FULL OUTER JOIN`.

- `FULL OUTER JOIN` displays the **NULLs** if there are no records that exist in either table. It is the major difference between the cross join and the full outer join.

- Example: Consider two tables:

	Id	Name	Salary
1	1	Pooja	12000.00
2	2	Smita	15000.00
3	3	Vaishali	17000.00

And

	Id	Name	Salary
1	1	Supriya	15000.00
2	2	Mayuri	16000.00

The result-set of `CROSS JOIN` and `FULL OUTER JOIN` are different:

```
Select * from Table_1
cross join Table_2 ;
```

100 %

Results | Messages

	Id	Name	Salary	Id	Name	Salary
1	1	Pooja	12000.00	1	Supriya	15000.00
2	2	Smita	15000.00	1	Supriya	15000.00
3	3	Vaishali	17000.00	1	Supriya	15000.00
4	1	Pooja	12000.00	2	Mayuri	16000.00
5	2	Smita	15000.00	2	Mayuri	16000.00
6	3	Vaishali	17000.00	2	Mayuri	16000.00

The output of Cross join.

VS

```
SELECT *FROM Table_1
FULL OUTER JOIN Table_2
ON Table_1.Name=Table_2.Name;
```

100 %

Results | Messages

	Id	Name	Salary	Id	Name	Salary
1	1	Pooja	12000.00	NULL	NULL	NULL
2	2	Smita	15000.00	NULL	NULL	NULL
3	3	Vaishali	17000.00	NULL	NULL	NULL
4	NULL	NULL	NULL	1	Supriya	15000.00
5	NULL	NULL	NULL	2	Mayuri	16000.00

The Output Of Full outer join.

Q) Difference between `ON` and `USING` operators with Join ?

`ON` is the more general of the two. One can join tables on a column, a set of columns and even a condition. Columns may have different names in the two tables.

Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name(i) = table2.column_name(j);
```

`USING` is useful when both tables share column(s) of the exact same name in both tables on which they join.

Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
USING (column1, column2, ..);
```

Q) What is the difference between having *`condition`* with `ON` vs *`condition`* with `WHERE` clause ?

We can use ***`condition`*** which we usually write in `WHERE` clause. But there is a difference when the ***`condition`*** is used with `ON` keyword vs with `WHERE` keyword.

The `WHERE` clause applies to the whole result-set while the `ON` clause only applies to the join. When we are doing `INNER JOIN`, then there is no difference but there might be different results possible when `OUTER JOIN` is used.

Example:

```
select t1.f1,t2.f2 from t1 left join t2 on t1.f1 = t2.f2 and t2.f4=1

select t1.f1,t2.f2 from t1 left join t2 on t1.f1 = t2.f2 where t2.f4=1
```

The former will left join to t2 records where f4 is 1, while the latter has effectively been turned back into an inner join to t2.

Q) Difference between `JOIN` , `','` (comma) and `INNER JOIN` keywords ?

1. `JOIN` and `','` (comma) are same/ functionally equivalent. Comma (`','`) operator was used in earlier SQL versions (though it still works) and `JOIN` operator is newer method of working with Joins in SQL, hence `JOIN` should be used wherever applicable.

2. `JOIN` and `INNER JOIN` are also same/ functionally equivalent. `INNER JOIN` can be a bit clearer to read (about the type of join used) whereas `JOIN` is a *syntactical sugar*.

   Or we can say, when we don't write the type of join with `JOIN` keyword, then `INNER` is used by default. Some SQL versions require to specify the type of join used always.

Q) Difference between `INNER JOIN` and `NATURAL JOIN` keywords ?

1. `NATURAL JOIN` joins two tables based on same attribute name and datatypes. Thus it will take all columns which have common name & datatype as basis of joining the tables. `ON` (or `USING`) keyword cannot be used with `NATURAL JOIN`.
   VS
   `INNER JOIN` joins two table on the basis of the column(s) which are explicitly specified in the `ON` clause. Thus, the columns taken as common can have different names in the two tables (though they need to have same datatype still). Also if there are more than one common column, `ON` keyword can give option not only to join using all common columns but also to join using only one or some of the common columns.

2. Syntax Difference:

```
SELECT column_name(s)
FROM table1
NATURAL JOIN table2;
```

   VS

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name(i) = table2.column_name(j);
```

3. In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column.
   VS
   In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also.

Q) What is Subequery

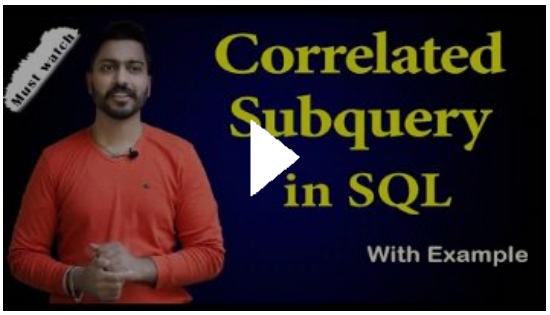
A) A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.
A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
There are a few rules that subqueries must follow —

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.

- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

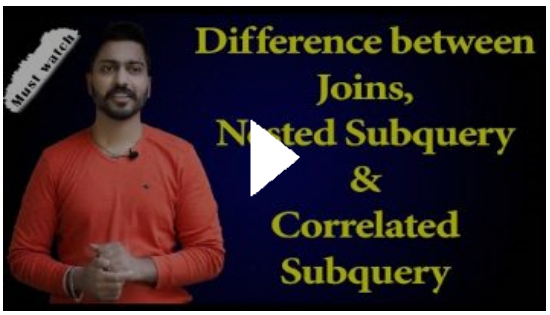
Correlated Queries

Lec-64: Correlated Subquery in SQL with Example | Imp for Placements, GATE, NET & SQL certification



Joins, Nested Subquery & Correlated Subquery Difference

Lec-65: Difference between Joins, Nested Subquery and Correlated Subquery | Most Imp Concept of SQL



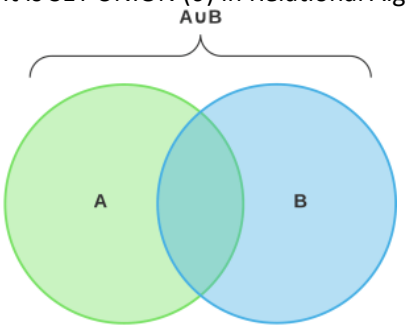
Difference between Nested Query, Correlated Query and Join Operation :

Parameters	Nested Query	Correlated Query	Join Operation
Definition	In Nested query, a query is written inside another query and the result of inner query is used in execution of outer query.	In Correlated query, a query is nested inside another query and inner query uses values from outer query.	Join operation is used to combine data or rows from two or more tables based on a common field between them.INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN are different types of Joins.
Approach	Bottom up approach i.e. Inner query runs first, and only once. Outer query is executed with result from Inner query.	Top to Down Approach i.e. Outer query executes first and for every Outer query row Inner query is executed.	It is basically cross product satisfying a condition.
Dependency	Inner query execution is not dependent on Outer query.	Inner query is dependent on Outer query.	There is no Inner Query or Outer Query. Hence, no dependency is there.
Performance	Performs better than Correlated Query but is slower than Join Operation.	Performs slower than both Nested Query and Join operations as for every outer query inner query is executed.	By using joins we maximize the calculation burden on the database but joins are better optimized by the server so the retrieval time of the query using joins will almost always be faster than that of a subquery.

Set Operations in SQL

1. **UNION Operator**

- The UNION operator could be used to find the result-set or combination of two or more tables.
- It is SET UNION (∪) in Relational Algebra.



- Two tables need to be UNION *compatible*. Necessary requirements for using UNION operation on two tables:
  a. Each table used within UNION must have the same number of columns.
  b. The columns must have same data types.
  c. The columns in each table must be in the same order.
- Syntax:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

- *IMPORTANT NOTE*: UNION operator provides only **unique** values by default.

- To allow **duplicate** values, use **UNION ALL**:
  Syntax:

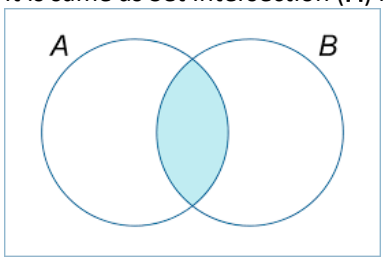
```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

- The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.
- We can use WHERE clause with both select statements. Note that condition in the first WHERE clause will correspond to first table (before keyword UNION) and similarly second WHERE clause will correspond to second table(after keyword UNION).
  Syntax:

```
SELECT column_name(s) FROM table1
WHERE conditions (On table1)
UNION
SELECT column_name(s) FROM table2
WHERE conditions (On table2);
```

2. **INTERSECT Operator**

- INTERSECT operator is used to provide the result of the intersection of two select statements.
- It is same as Set Intersection (∩) in relational algebra.



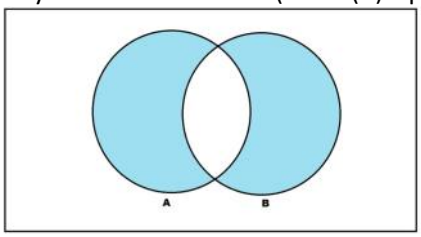
- This implies the result contains all the rows which are common to both the SELECT statements.
- Necessary Conditions for INTERSECT operator to be used with two tables is same that in UNION operator.

- Also, Similar to UNION, it show only distinct values and INTERSECT ALL can be used to display duplicate values.
- Syntax:

```
SELECT column_name(s) FROM table1
INTERSECT
SELECT column_name(s) FROM table2;
```

3. EXCEPT Operator

- This works exactly opposite to the INTERSECT clause.
- The result, in this case, contains all the rows except the common rows of the two SELECT statements.
- It is Symmetric Difference (Delta (Δ) Operation) in Relational Algebra.

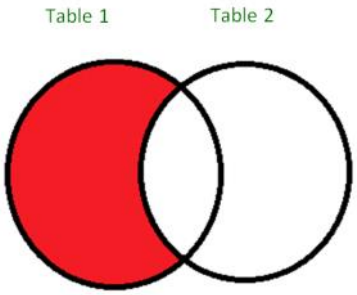


- Syntax:

```
SELECT column_name(s) FROM table1
EXCEPT
SELECT column_name(s) FROM table2;
```

4. MINUS Operator

- The MINUS operator is used to subtract the result set obtained by first SELECT query from the result set obtained by second SELECT query.
- In simple words, we can say that MINUS operator will return only those rows which are unique in only first SELECT query and not those rows which are common to both first and second SELECT queries.
- It is Set Difference(-) in Relational Algebra.



- Syntax:

```
SELECT column_name(s) FROM table1
MINUS
SELECT column_name(s) FROM table2;
```

- The MINUS operator is not supported with all databases. It is supported by Oracle database but not SQL server or PostgreSQL.

DDL Commands

Create Command

1. CREATE DATABASE: Used to create a new database

```
CREATE DATABASE databasename;
```

   Note: Admin priviledge is required for creating database.

   Note: Other Useful commands related to database are:
   - `SHOW DATABASES;`
   - `USE databasename;`

2. CREATE TABLE: Used to create a new table

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
  ....
);
```

3. CREATE TABLE AS: Used to create a new table by coping data from already exisiting table.

```
CREATE TABLE new_table_name AS
  SELECT column1, column2,...
  FROM existing_table_name
  WHERE ....;
```

Q) Differences between CREATE TABLE AS and SELECT INTO ?
CREATE TABLE gives you a better control over your table's definition prior to inserting the data, like NOT NULL, constraints, etc. things that you cannot do using SELECT INTO.
Morevoer, SELECT INTO is deprecated in some SQL versions, so it better to use CREATE TABLE AS for copying data from one existing table into a new table.

4. ***Create User***
   https://www.geeksforgeeks.org/mysql-create-user-statement/

   - MySQL allows us to specify which user account can connect to a database server. The user account details in MySQL contains two information – username and host from which the user is trying to connect in the format username@host-name.
   - If the admin user is connecting through localhost then the user account will be admin@localhost.
   - MySQL stores the user account in the user grant table of the mysql database.
   - The CREATE USER statement in MySQL allows us to create new MySQL accounts or in other words, the CREATE USER statement is used to create a database account that allows the user to log into the MySQL database.

   Syntax:
   CREATE USER user_account IDENTIFIED BY password;

   user_account: It is the name that the user wants to give to the database account.The user_account should be in the format 'username'@'hostname'
   password: It is the password used to assign to the user_account.The password is specified in the IDENTIFIED BY clause.

Drop Command

1. `DROP DATABASE databasename;`

It will delete entire database i.e. all tables will get deleted.

2. `DROP TABLE table_name;`
   It will delete entire table and removed from the database. All rows with all columns(attributes) will get deleted i.e. no schema of relation will remain after dropping the table.

Both drop statements require admin priviledge to execute.
It is a DDL (Data Definition Language) command and cannot be rolled back (deleted table/database cannot be restored) once performed.

Truncate Command

`TRUNCATE TABLE table_name;`
It will delete all rows/records from the table and not the table itself. Hence, columns/attributes will remain there and schema of the table will not get deleted.

Note: Truncate command performs same operation i.e. removing all records from the table as DELETE FROM statement without WHERE condition.

Differences between DELETE and TRUNCATE

S.NO	Delete	Truncate
1.	The DELETE command is used to delete specified rows(one or more).	While this command is used to delete all the rows from a table.
2.	It is a DML(Data Manipulation Language) command.	While it is a DDL(Data Definition Language) command.
3.	There may be WHERE clause in DELETE command in order to filter the records.	While there may not be WHERE clause in TRUNCATE command.
4.	In the DELETE command, a tuple is locked before removing it.	While in this command, data page is locked before removing the table data.
5.	The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row.	TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.
6.	DELETE command is slower than TRUNCATE command.	While TRUNCATE command is faster than DELETE command.
7.	To use Delete you need DELETE permission on the table.	To use Truncate on a table we need at least ALTER permission on the table.
8.	Identity of column retains the identity after using DELETE Statement on table.	Identity of the column is reset to its seed value if the table contains an identity column.
9.	The delete can be used with indexed views.	Truncate cannot be used with indexed views.

Backup Command

- It is used in SQL Server to create a full back up of an existing SQL database.
- Syntax:
  ```
  BACKUP DATABASE databasename
  TO DISK = 'filepath';
  ```

- A differential back up only backs up the parts of the database that have changed since the last full database backup. Hence, it reduces the backup time.
- Syntax with DIFFERENTIAL Statement:
  ```
  BACKUP DATABASE databasename
  TO DISK = 'filepath'
  WITH DIFFERENTIAL;
  ```

- Example:
  ```
  BACKUP DATABASE testDB
  ```

```
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

Alter Command

- It is used to add, delete, or modify columns in an existing table.
- It is also used to add and drop various **constraints** on an existing table {in next slide}.

1. Add Column

   ```
   ALTER TABLE table_name
   ADD column_name datatype;
   ```

2. Drop Column

   ```
   ALTER TABLE table_name
   DROP COLUMN column_name;
   ```

   Note: Some SQL versions don't allow drop column.

3. Alter/Modify Column { Change Datatype }

   ```
   ALTER TABLE table_name
   ```
   SQL Server => `ALTER COLUMN column_name datatype;`
   MySQL => `MODIFY COLUMN column_name datatype;`
   Oracle => `MODIFY column_name datatype;`

4. Add Constraints

Constraints

- SQL constraints are used to specify rules for the data in a table. Constraints are used to limit the type of data that can go into a table. This ensures the *accuracy* and *reliability* of the data in the table.
- If there is any violation between the constraint and the data action, the action(insertion/updation of data) is aborted and error is displayed.
- Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.
- It can be added while creating table (using CREATE TABLE) or after creating by ALTER TABLE.
- There can be multiple constraints NOT NULL, UNIQUE, FOREIGN KEY, however, there can be only one PRIMARY KEY constraint in the entire table (though primary key can be composed of multiple columns {composite key}).

Syntax: **Add Constraint**

1. Single Column using CREATE TABLE

   1.a) SQL Server or Oracle

```
CREATE TABLE table_name (
    column1 datatype CONSTRAINT_TYPE,
    column2 datatype,
    column3 datatype,
  ....
);
```

   1.b) MySQL

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....,
    CONSTRAINT_TYPE(column_name)
);
```

2. Multiple columns using CREATE TABLE

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....,
    CONSTRAINT constraint_name CONSTRAINT_TYPE(column_name(s))
);
```

3. Single Column using ALTER TABLE

```
ALTER TABLE table_name ADD CONSTRAINT_TYPE(column_name);
```

4. Multiple Columns using ALTER TABLE

```
ALTER TABLE  table_name ADD CONSTRAINT constraint_name CONSTRAINT_TYPE(column_name(s));
```

Syntax: **Drop Constraint**:

1. MySQL

```
ALTER TABLE table_name DROP CONSTRAINT_TYPE constraint_name;
```

2. SQL Server / Oracle

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

Note: Syntax of Drop Constraint can be different for different constraints. Refer:
https://www.w3schools.com/sql/sql_ref_drop_constraint.asp

Note: Here `constraint_name` means a user-defined name given to column(s) based on the `CONSTRAINT_TYPE` imposed.

For Eg) For `UNIQUE` constraint, *`constraint_name`* can be *UC_Persons*.

The following constraints are commonly used in SQL:

i) **NOT NULL** -
- By default, a column can hold NULL values. The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

ii) **UNIQUE** -
- Ensures that all values in a column are different.

iii) **PRIMARY KEY** -
- A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
- A table can have only ONE primary key in the table, though this primary key can consist of single or multiple columns (fields).
- If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

Difference between Unique Constraint & Primary Key

Paramenter	**PRIMARY KEY**	**UNIQUE KEY**
Basic	Used to serve as a unique identifier for each row in a table.	Uniquely determines a row which isn't primary key.
NULL value acceptance	Cannot accept NULL values.	Can accept one NULL value.
Number of keys that can be defined in the table	Only one primary key	More than one unique key
Index	Creates clustered index	Creates non-clustered index

iv) **FOREIGN KEY** -
- Uniquely identifies a row/record in another table.
- Used to link two tables together.
- A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.
- The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.
- Necessary Conditions for Foreign Key are:
  1. Must reference PRIMARY KEY in primary table.
  2. Child may have duplicates and nulls.
  3. Foreign key column and constraint column should have matching data types.

- Syntax: It should be defined in Referencing Table.

```
FOREIGN KEY (referencing_table_foreignkey_columns)
REFERENCES referenced_table_name(referenced_table_primarykey_columns)
ON DELETE delete_type;
```

- We can use any of the following delete(or modify) type with foreign key (in referencing table) which works when *`referenced_table_primarykey_columns`* (in referenced table) get deleted:
  - **ON DELETE NO ACTION**: It is used by default. It will not allow to delete/modify records which cause violation.
  - **ON DELETE CASCADE**: When this option is specified in foreign key definition, if a record is deleted in master table, all corresponding record in detail table will be deleted.
  - **ON DELETE SET NULL**: A Foreign key with SET NULL ON DELETE means if record in parent table is deleted, corresponding records in child table will have foreign key fields set to null. Records in child table will not be deleted.

- For allowed Insertions, Deletions & Modification operations on both tables, refer Gate Smasher Videos.
  https://www.youtube.com/watch?v=UyqpQ3D2yCw
  https://www.youtube.com/watch?v=DM2lAomoDrg

v) **CHECK** -
- It limits the value range that can be placed in a column.
- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- Example:
  Single Column => `CHECK (Age>=18)`
  Multiple Columns => `CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Delhi')`

vi) **DEFAULT** -
- Sets a default value for a column when no value is specified.
- The default value will be added to all new records IF no other value is specified.
- Example:
```
DEFAULT 'General'
DEFAULT 0
DEFAULT GETDATE()
```
- Adding DEFAULT constraint using ALTER TABLE is different in various SQL versions:
  MySQL => `ALTER City SET DEFAULT 'Sandnes';`
  SQL Server=> `ADD CONSTRAINT df_City DEFAULT 'Sandnes' FOR City;`
  MS Access => `ALTER COLUMN City SET DEFAULT 'Sandnes';`
  Oracle => `MODIFY City DEFAULT 'Sandnes';`

vii) **INDEX** -
- Indexes are used to retrieve data from the database very quickly. The users cannot see the indexes, they are just used to speed up searches/queries.
- It enhances the speed of operations accessing data from a database table at the cost of additional writes and memory to maintain the index data structure.
- *Note*: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.
- Syntax:
  CREATE INDEX (Creates an index on a table. Duplicate values are allowed)
```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```
  CREATE UNIQUE INDEX (Creates a unique index on a table. Duplicate values are not allowed)
```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```
- In all databases, while creating the table, if primary key is specified, then a clustered indexing is implemented automically on the primary key i.e. the main file gets sorted on the primary key.

  If we need to create a clustered index on non-key attribute, we first have to drop the index on primary key (since there can only be one clustered index for a table), then create index using above commands.

  However, if a non-clustered index has to be created, then we need not drop previous index (whether clustered or non-clustered) as there can be multiple non-clustered indexes for a single table.

- DROP INDEX:
  MS Access => `DROP INDEX index_name ON table_name;`
  SQL Server => `DROP INDEX table_name.index_name;`
  Oracle => `DROP INDEX index_name;`
  MySQL => `ALTER TABLE table_name DROP INDEX index_name;`

Q) Describe the different index configurations possible for a table?

i) A clustered index: When only a single clustered index is present.
ii) A non-clustered index: When only a single non-clustered index is present.
iii) Many non-clustered indexes: When more than one non-clustered indexes is present.
iv) A clustered index and a non-clustered index: When a single clustered index and a single non-clustered index is present.
v) A clustered index and many non-clustered indexes: When a single clustered index and more than one non-clustered indexes are present.
vi) No index: When there are no indexes present.

Q) Difference between UNIQUE INDEX and NON-UNIQUE INDEX?

R) Unique indexes are indexes that help maintain data integrity by ensuring that no two rows of data in a table have identical key values. Once a unique index has been defined for a table, uniqueness is enforced whenever keys are added or changed within the index.

Non-unique indexes, on the other hand, are not used to enforce constraints on the tables with which they are associated. Instead, non-unique indexes are used solely to improve query performance by maintaining a sorted order of data values that are used frequently.

Q) Difference between CLUSTERED INDEX and NON-CLUSTERED INDEX?

Clustered Index	**Non-Clustered Index**
It physically sorts the rows of a table based on the primary key or on a column that is unique and not null (generally we use primary key). Hence, it modifies the original records in the main file.	This is an index structure that is separate from the actual table which sorts one or more selected columns. Hence, it does not modify the original records in the main file.
If search key is primary key, then it is also known as **primary indexing** else it is known as clustered indexing.	It is also known as **secondary indexing**.
Querying data is fast as the main file is sorted on the basis of search key only.	Querying data is slower (when compared to clustered-index) as main file is unsorted on the search key in non-clustered index. (However query operations are still fast when compared to no indexing as atleast index file is sorted.)
There can only be one clustered index per table as the main file can be sorted only on one attribute.	There can be many non-clustered indexes per table but separate index files need to be maintained for each indexing on the table.
It doesn't need extra disk space as the main file itself gets sorted on the basis of search key and there is no need to maintain index file.	It requires extra space to store those indexes in an index file.
A typical use case can be where there are range-based queries on the primary key.	They are used when tuples need to be queried on a basis of search key on basis of which main file is unordered.
Updation and Insertion are slow because the sorted order in the main file has to be maintained.	Updation and Insertion are slow because the sorted order in index file has to be maintained.

Advantages of Clustered Index

The pros/benefits of the clustered index are:

- Clustered indexes are an ideal option for range or group by with max, min, count type queries
- In this type of index, a search can go straight to a specific point in data so that you can keep reading sequentially from there.
- Clustered index method uses location mechanism to locate index entry at the start of a range.
- It is an effective method for range searches when a range of search key values is requested.
- Helps you to minimize page transfers and maximize the cache hits.

Advantages of Non-clustered index

Pros of using non-clustered index are:

- A non-clustering index helps you to retrieves data quickly from the database table.
- Helps you to avoid the overhead cost associated with the clustered index
- A table may have multiple non-clustered indexes in RDBMS. So, it can be used to create more than one index.

Disadvantages of Clustered Index

Here, are cons/drawbacks of using clustered index:

- Lots of inserts in non-sequential order
- A clustered index creates lots of constant page splits, which includes data page as well as index pages.
- Extra work for SQL for inserts, updates, and deletes.
- A clustered index takes longer time to update records when the fields in the clustered index are changed.
- The leaf nodes mostly contain data pages in the clustered index.

Disadvantages of Non-clustered index

Here, are cons/drawbacks of using non-clustered index:

- A non-clustered index helps you to stores data in a logical order but does not allow to sort data rows physically.
- Lookup process on non-clustered index becomes costly.
- Every time the clustering key is updated, a corresponding update is required on the non-clustered index as it stores the clustering key.

SQL Problems

1. **Second Maximum Salary**
   Problem Link: https://leetcode.com/problems/second-highest-salary/
   Solution Video: https://www.youtube.com/watch?v=9lLpe_detTY

2. **Nth Maximum Salary**
   Problem Link: https://leetcode.com/problems/nth-highest-salary/
   Solution Video: https://www.youtube.com/watch?v=fh4yBn0oTaM

3. **Fetch Alternate Records from a table:**
   To fetch even Numbered row:
   SELECT * FROM table_name WHERE column_name % 2 = 0;
   To fetch odd Numbered row:
   SELECT * FROM table_name WHERE column_name % 2 = 1;

4. **Fetch First N Characters from the string:**
   Use Substring Function:
   SELECT SUBSTR(column_name, length_required) FROM table_name;

5. **Operator used for Pattern Matching:**
   LIKE operator and REGEXP() functions are used

6. Reformat Table: https://leetcode.com/problems/reformat-department-table/

7. Swap contents for alternate records: https://leetcode.com/problems/swap-salary/

8. **Select only Duplicate records**: https://leetcode.com/problems/duplicate-emails/

9. **Duplicate Duplicate records**: https://leetcode.com/problems/delete-duplicate-emails/

10. **Using Having vs Using Where**: https://leetcode.com/problems/classes-more-than-5-students/

11. **Using Case Function**: https://leetcode.com/problems/exchange-seats/

12. Highest Salaries in Each Group (**Using Nested Queries vs Join**): https://leetcode.com/problems/department-highest-salary/

13. **Problems on Joins**:
    https://www.interviewbit.com/problems/neutral-reviewers/
    https://www.interviewbit.com/problems/movie-character/
    https://www.interviewbit.com/problems/short-films/
    https://www.interviewbit.com/problems/actors-and-their-movies/

DCL Commands

Data Control Language(DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges. Privileges are of two types,

System: This includes permissions for creating session, table, etc and all types of other system privileges.
Object: This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

- GRANT: Used to provide any user access privileges or other priviliges for the database.
- REVOKE: Used to take back permissions from any user.

Granting Privileges
Using the Create User Statement only creates a new user but does not grant any privileges to the user account. Therefore to grant privileges to a user account, the GRANT statement is used.

Privileges that can be granted to the users are:

1. Select: Select statements on tables
2. Insert: Insert statements on tables
3. Delete: Delete statements on tables
4. Index: Create indexes on existing tables
5. Create: Create table statements
6. Alter: Perform changes to table definition
7. Drop: Drop table statements
8. *ALL*: Grant all permissions except GRANT OPTION
9. Update: Update statements on tables
10. *GRANT*: Grant access to user to grant privileges to other users.

Syntax:
GRANT privileges_names ON object TO user;

- *privileges_name:* These are the access rights or privileges granted to the user.
- *object:* It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.
- *user:* It is the name of the user to whom the privileges would be granted.

Revoking Privileges
The Revoke statement is used to revoke some or all of the privileges which have been granted to a user in the past.

Syntax:
REVOKE privileges ON object FROM user;

object: It is the name of the database object from which permissions are being revoked. In the case of revoking privileges from a table, this would be the table name.
user: It is the name of the user from whom the privileges are being revoked.

Detailed Articles:

1. https://www.geeksforgeeks.org/mysql-grant-revoke-privileges/
2. https://www.studytonight.com/dbms/dcl-command.php

Differences between GRANT and REVOKE

S.NO	Grant	Revoke
1	This DCL command grants permissions to the user on the database objects.	This DCL command removes permissions if any granted to the users on database objects.
2	It assigns access rights to users.	It revokes access rights of users.

3	For each user you need to specify the permissions.	If access for one user is removed; all the particular permissions provided by that users to others will be removed.
4	When the access is decentralized granting permissions will be easy.	If decentralized access removing the granted permissions is difficult.

TCL Commands

- Transaction Control Language(TCL) commands are used to manage transactions in the database.
- These are used to manage the changes made to the data in a table by DML statements.
- It also allows statements to be grouped together into logical transactions.
- It is important to note that these statements cannot be used while creating tables and are only used with the DML Commands such as – INSERT, UPDATE and DELETE.

1) **COMMIT**
   - COMMIT command is used to permanently save any transaction into the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.
   - When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back. To avoid that, we use the COMMIT command to mark the changes as permanent.
   - Syntax: `COMMIT;`

2) **ROLLBACK**
   - This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.
   - Example) If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.
   - Syntax: `ROLLBACK TO savepoint_name;`

Differences Between COMMIT and ROLLBACK statements:

COMMIT	ROLLBACK
COMMIT permanently saves the changes made by current transaction.	ROLLBACK undo the changes made by current transaction.
Transaction can not undo changes after COMMIT execution.	Transaction reaches its previous state after ROLLBACK.
When transaction is successful, COMMIT is applied.	When transaction is aborted, ROLLBACK occurs.

3) **SAVEPOINT**
   - SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.
   - In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.
   - Syntax: `SAVEPOINT savepoint_name;`

- **RELEASE SAVEPOINT**
  - To remove a savepoint, we use `RELEASE` command. Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

- Syntax: `RELEASE savepoint_name;`

- **SET TRANSACTION**
  - The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.
  - For example, you can specify a transaction to be read only or read write.
  - Syntax: `SET TRANSACTION [ READ WRITE | READ ONLY ];`

Views in SQL

- 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.

Uses of a View :
A good database should contain views due to the given reasons:

1. Restricting data access: Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.

2. Hiding data complexity: A view can hide the complexity that exists in a multiple table join.

3. Simplify commands for the user: Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.

4. Store complex queries: Views can be used to store complex queries.

5. Rename Columns: Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.

6. Multiple view facility: Different views can be created on the same table for different users.

Create View
We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables. (Multiple tables means we can create a view from join of two or more 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
```

WITH CHECK OPTION
- The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

  Syntax:
  ```
  CREATE VIEW view_name AS
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition
  WITH CHECK OPTION;
  ```

Update View
A view can be updated with the CREATE OR REPLACE VIEW statement.

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.

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- 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.

Syntax:

- ***Add or remove fields in a view:***

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- ***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.

```
INSERT INTO view_name
(column1, column2, ...)
VALUES (value1, value2, ...);
```

- ***Deleting a row in 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.

```
DELETE FROM view_name
WHERE condition;
```

Drop View

A view is deleted with the DROP VIEW statement.

Syntax:

```
DROP VIEW view_name;
```

Functions & Procedures

Functions in PL-SQL

A function can be used as a part of SQL expression i.e. we can use them with select/update/merge commands. One most important characteristic of a function is that unlike procedures, it must return a value.

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name  type [, ...])]
// this statement  is must for functions
RETURN return_datatype
{IS | AS}

BEGIN
    // program code

[EXCEPTION
   exception_section;

END [function_name];
```

Advantages:

1. We can make a single call to the database to run a block of statements thus it improves the performance against running SQL multiple times. This will reduce the number of calls between the database and the application.
2. We can divide the overall work into small modules which becomes quite manageable also enhancing the readability of the code.
3. It promotes reusability.
4. It is secure since the code stays inside the database thus hiding internal database details from the application(user). The user only makes a call to the PL/SQL functions. Hence security and data hiding is ensured.

Detailed Article: https://www.geeksforgeeks.org/functions-in-plsql/

Procedures in PL-SQL

PL/SQL is a block-structured language that enables developers to combine the power of SQL with procedural statements.
A stored procedure in PL/SQL is nothing but a series of declarative SQL statements which can be stored in the database catalogue.
A procedure can be thought of as a function or a method. They can be invoked through triggers, other procedures, or applications on Java, PHP etc.
All the statements of a block are passed to Oracle engine all at once which increases processing speed and decreases the traffic.
A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.
You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Syntax:

Create Procedure

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Execute Procedure

```
EXEC procedure_name;
```

Advantages:

- They result in performance improvement of the application. If a procedure is being called frequently in an application in a single connection, then the compiled version of the procedure is delivered.
- They reduce the traffic between the database and the application, since the lengthy statements are already fed into the database and need not be sent again and again via the application.
- They add to code reusability, similar to how functions and methods work in other languages such as C/C++ and Java.

Disadvantages:

- Stored procedures can cause a lot of memory usage. The database administrator should decide an upper bound as to how many stored procedures are feasible for a particular application.
- MySQL does not provide the functionality of debugging the stored procedures.

Detailed Article: https://www.w3schools.com/sql/sql_stored_procedures.asp
https://www.tutorialspoint.com/plsql/plsql_procedures.htm

Difference between Function and Procedure:

S.NO	Function	Procedure
1.	A function deals with as an expression.	Whereas a procedure does not deal with as an expression.
2.	Function is used to calculate something from a given input. Hence it got its name from Mathematics.	While procedure is the set of commands, which are executed in a order.
3.	The function can be called by a procedure.	But a procedure can not be called by a function.
4.	In sql, inside the function we can not use the DML(Data manipulation language) commands such as Insert, Delete, Update.	Here, in sql, inside the procedure we can use DML commands.
5.	Functions can be called through sql queries.	However, the procedure can't be called through a sql query.
6.	Each time functions are compiled when they are called.	Whereas, procedures are compiled only once and can be called again and again as needed without being compiled each time.
7.	The return statement of a function returns the control and function's result value to the calling program.	While the return statement of the procedure returns control to the calling program, it can not return the result value.
8.	Function doesn't support try-catch blocks.	While it supports try-catch blocks.
9.	Function can be operated in the SELECT statement.	While it can't be operated in the SELECT statement.
10.	Function does not support explicit transaction handles.	While procedure supports explicit transaction handles.

Trigger in SQL

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:
Create Trigger Trigger_Name
(Before | After) [Insert | Update | Delete]
on [Table_Name]
[for each row | for each column]
[trigger_body]

Create Trigger
These two keywords are used to specify that a trigger block is going to be declared.

Trigger_Name
It specifies the name of the trigger. Trigger name has to be unique and shouldn't repeat.

(*Before | After*)
This specifies when the trigger will be executed. It tells us the time at which the trigger is initiated, i.e, either before the ongoing event or after.

- Before Triggers are used to update or validate record values before they're saved to the database.
- *After Triggers* are used to access field values that are set by the system and to effect changes in other records. The records that activate the after trigger are read-only. We cannot use After trigger if we want to update a record because it will lead to read-only error.

[*Insert | Update | Delete*]
These are the DML operations and we can use either of them in a given trigger.

***on* [*Table_Name*]**
We need to mention the table name on which the trigger is being applied. Don't forget to use **on** keyword and also make sure the selected table is present in the database.

[for each row | for each column]

- Row-level trigger gets executed before or after *any column value of a row* changes
- Column Level Trigger gets executed before or after the *specified column* changes

[*trigger_body]*
It consists of queries that need to be executed when the trigger is called.

Note: We can also create a nested trigger that can do multi-process. Also handling it and terminating it at the right time is very important. If we don't end the trigger properly it may lead to an infinite loop.

Example of Trigger)

```
CREATE TRIGGER sample_trigger
before INSERT
ON student
FOR EACH ROW
SET new.total = new.marks/6;
```

- To drop a trigger use following command:
  DROP TRIGGER trigger name;

- To display the list of all triggers in all databases, use following command:
  SHOW TRIGGERS;

- To display the list of all triggers in a particular database, use following command:
  SHOW TRIGGERS IN database_name;

Advantages of Triggers

- Forcing security approvals on the table that are present in the database
- Triggers provide another way to check the integrity of data
- Counteracting invalid exchanges
- Triggers handle errors from the database layer
- Normally triggers can be useful for inspecting the data changes in tables
- Triggers give an alternative way to run scheduled tasks. Using triggers, we don't have to wait for the scheduled events to run because the triggers are invoked automatically before or after a change is made to the data in a table

Disadvantages of Triggers

- Triggers can only provide extended validations, i.e, not all kind validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints
- Triggers may increase the overhead of the database
- Triggers can be difficult to troubleshoot because they execute automatically in the database, which may not invisible to the client applications

Difference between Triggers and Procedures :

Triggers	Procedures
A Trigger is implicitly invoked whenever any event such as INSERT, DELETE, UPDATE occurs in a TABLE.	A Procedure is explicitly called by user/application using statements or commands such as exec, EXECUTE, or simply procedure_name
Only nesting of triggers can be achieved in a table. We cannot define/call a trigger inside another trigger.	We can define/call procedures inside another procedure.
In a database, syntax to define a trigger: CREATE TRIGGER TRIGGER_NAME	In a database, syntax to define a procedure: CREATE PROCEDURE PROCEDURE_NAME
Transaction statements such as COMMIT, ROLLBACK, SAVEPOINT are not allowed in triggers.	All transaction statements such as COMMIT, ROLLBACK are allowed in procedures.
Triggers are used to maintain referencial integrity by keeping a record of activities performed on the table.	Procedures are used to perform tasks defined or specified by the users.
We cannot return values in a trigger. Also, as an input, we cannot pass values as a parameter.	We can return 0 to n values. However, we can pass values as parameter

SQL from Programming Language

To access SQL from other programming languages, we can use:

1. ***Dynamic SQL*:**
   ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) serve as APIs for a program to interact with a database server.
   In general, the application must make calls to:
   - connect with the database server
   - send SQL commands to the database server
   - fetch tuples of result one-by-one into program variables

   ODBC works with C, C++, C# and Visual Basic (other APIs such as ADO.NET sit on top of ODBC).
   ODBC is the standard for application programs communicating with a database server.

   JDBC works with Java.
   Along with supporting various features for querying and updating data, and for retrieving query results, JDBC also supports metadata retrieval i.e. retrieving information about the database such as relations present in the database and the names and types of relation attributes.

2. ***Embedded SQL:***
   Embedded SQL refers to embedding SQL queries in another language. SQL can be embedded in various languages including C, Java and Cobol.
   A language into which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
   The EXEC SQL statement is used to identify embedded SQL request to the preprocessor:
   EXEC SQL <embedded SQL statement> END_EXEC

Static (Embedded) SQL	**Dynamic (Interactive) SQL**
In Static SQL, how database will be accessed is predetermined in the embedded SQL statement.	In Dynamic SQL, how database will be accessed is determined at run time.
It is more swift and efficient.	It is less swift and efficient.
SQL statements are compiled at compile time.	SQL statements are compiled at run time.
Parsing, Validation, Optimization and Generation of application plan are done at compile time.	Parsing, Validation, Optimization and Generation of application plan are done at run time.
It is generally used for situations where data is distributed uniformly.	It is generally used for situations where data is distributed non uniformly.
EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are not used.	EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are used.
It is less flexible.	It is more flexible.

Limitation of Dynamic SQL:

We cannot use some of the SQL statements Dynamically.

Performance of these statements is poor as compared to Static SQL.

Limitations of Static SQL:

They do not change at runtime thus are hard-coded into applications.

Injection in SQL

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is the placement of malicious code in SQL statements, via web page input.
- SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.
- Example) A hacker might get access to all the user names and passwords in a database, by simply inserting '1=1-- into the input field.
- To protect a web site from SQL injection, you can use SQL parameters. SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.
- The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

Detailed Article: https://www.w3schools.com/sql/sql_injection.asp
Detailed Video: What is SQL Injection? | SQL Injection Tutorial | Cybersecurity Training | Edureka

Cursor in SQL

SQL Server creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

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:**
  - Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
  - 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

- **Explicit Cursors :**
  - Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.
  - 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.
  - 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

How to create Explicit Cursor:

1. **Declare Cursor Object.**
   Syntax : DECLARE cursor_name CURSOR FOR SELECT * FROM table_name

DECLARE s1 CURSOR FOR SELECT * FROM studDetails

2. **Open Cursor Connection.**
   Syntax : OPEN cursor_connection
   OPEN s1

3. **Fetch Data from cursor.**
   There are total 6 methods to access data from cursor. They are as follows :
   FIRST is used to fetch only the first row from cursor table.
   LAST is used to fetch only last row from cursor table.
   NEXT is used to fetch data in forward direction from cursor table.
   PRIOR is used to fetch data in backward direction from cursor table.
   ABSOLUTE n is used to fetch the exact nth row from cursor table.
   RELATIVE n is used to fetch the data in incremental way as well as decremental way.

   Syntax : FETCH NEXT/FIRST/LAST/PRIOR/ABSOLUTE n/RELATIVE n FROM cursor_name
   FETCH FIRST FROM s1
   FETCH LAST FROM s1
   FETCH NEXT FROM s1
   FETCH PRIOR FROM s1
   FETCH ABSOLUTE 7 FROM s1
   FETCH RELATIVE -2 FROM s1

4. **Close cursor connection.**
   Syntax : CLOSE cursor_name
   CLOSE s1

5. **Deallocate cursor memory.**
   Syntax : DEALLOCATE cursor_name
   DEALLOCATE s1

Database Design Phases

Database Design Phases

1. **Requirements Collection Phase:**

   Goal of this phase is collecting correct requirements from stakeholders and users. This is only possible when user has a clear view of his needs. Following two kinds of requirements are collected in this phase:

   a. ***Data Model Requirements*:**

      These deal with different pieces of data that need to be stored along with their relationships with one another. The data model requirements are represented using conceptual level data models, like entity/relationship model (ER model) and Unified Modelling Language (UML).

   b. ***Functional Requirements:***

      This involves day-to-day tasks and operations that are undertaken by enterprise for which database is being developed. E.g. The functional requirements for a Hospital System would be: acquiring new medicines, maintaining doctor records, maintaining patient records, adding new patient records etc.

2. **Conceptual Design:**

When every data requirement is stored and analyzed, we apply the concepts of the chosen data model, translate these requirements into a conceptual schema of the database. A fully developed conceptual schema indicates the functional requirements of the enterprise.

3. **Logical Design**

The *logical phase* of database design is also called the *data modeling mapping phase*. This phase gives us a result of relation schemas from the ER Model or the Class Diagram. There are rules for converting the ER model or class diagram to relation schemas.

Finally in logical design, we perform normalization of relation schema. The main purpose of ***normalization*** is to remove superfluity and every other potential anomaly during the update. With every normalization phase, a new table is added to the database.

4. **Physical Design**

In this last phase, we implement the database design by chosing a DBMS (Database Management System) to be used like RDBMS (MySQL, Oracle, SQL Server, etc) or NoSQL Database like MongoDB.

SQL clauses are written to help in creating the database. Also, the indexes and the integrity constraints (rules) are defined in this phase. And finally the data is added and the database can finally be tested.

Note: In designing a database schema, we must ensure to avoid various problems like:

- **Redundancy:** A bad design may result in repeat information. Redundant representation of information may lead to

data inconsistency among the various copies of information

- **Incompleteness**: A bad design may make certain aspects of the enterprise difficult or impossible to model.

ER Model

ER Model is used to model the logical view of the system from data perspective as a collection of basic objects called ***entities*** and ***relationships*** among these entities and ***attributes*** which define their properties.

It is represented diagrammatically by an ***entity-relationship (ER) diagram***.

Advantages of ER Diagram

- It constructs used in the ER diagram can easily be transformed into relational tables.
- It is simple and easy to understand with minimum training.

Disadvantages of ER Diagram

- Loss of information content
- Limited constraints representation
- It is overly complex for small projects

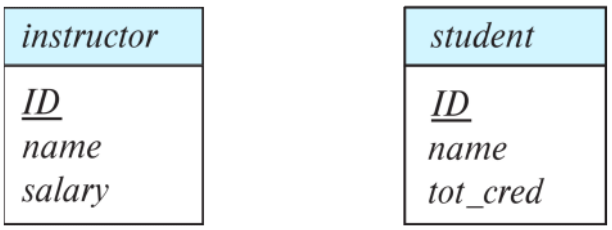
ER Model consists of various components:

Entity:

- It is an object in the model that is distinguishable from other objects based on values of attributes it possess.
- Types of Entities:
  It may have physical existence (*tangible entity*) like a particular employee, student, etc. or may have conceptual existence (*intangible entity*) like particular company, job, course, etc.
- *Entity in ER Diagram:* Entity cannot be represented in an ER diagram as it is an instance/data.
- *Entity in Relational Model*: Entity is represented as a record/row/tuple in a relation(table).

Entity Set:

- It is a set or collection of entities of the same type i.e. which share the same properties or attributes.
- Example: set of all persons, companies, trees, holidays
- *Entity Set in ER Diagram*: Entity sets can be represented graphically as follows:
  - **Rectangles** represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



- *Entity Set in Relational Model*: Entity Set is represented by a table/relation in Relational Model.

Attributes:

- They are the units or properties which describe the characteristics of entities.
- For each attribute, there is a set of permitted values called **domain**.
- In ER Diagram, attribute is represented by a **oval** or ellipse, whereas in Relational Model, attribute is represented as a column/field.
- Types of attributes:
  a. **Simple Attributes vs Composite Attributes:**
  An attribute which cannot be further subdivided into components is a simple attribute. It is represented by simple oval. Example: The roll number of a student, the id number of an employee.

  An attribute which can be splitted into components is a composite attribute. It is represented with oval connected with further ovals. Example: The address can be further splitted into house number, street number, city, state,

country and pincode.

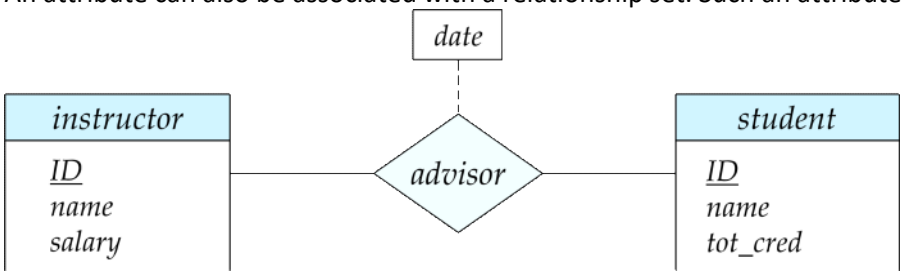
b. Single Valued Attributes vs Multi Valued Attributes:
The attribute which takes up only a single value for each entity instance is single-valued attribute.
Example: The age of a student.

The attribute which takes up more than a single value for each entity instance is multi-valued attribute. It is represented by double oval (2 concentric ovals).
Example: Phone number of a student:Landline and mobile.

c. Stored Attributes vs Derived Attributes:
Those attributes which are actually stored in the database are known as stored attributes. Example) Date of Birth

Those attributes which can be computed at run time and thus need not be stored are known as derived attributes. It is represented by dotted oval in ER diagram. Example) Age

d. Discriptive Attribute:
An attribute can also be associated with a relationship set. Such an attribute is known as discriptive attribute.



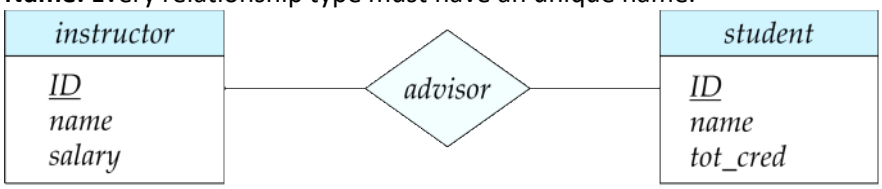
Relationship:

- Relationship is an association between two or more entities of same or different entity set.
- Since relationship is an instance or data, it cannot be represented in ER diagram.

Relationship Type/Set:

- It is the set of similar type of relationships. It is a mathematical relation among two or more entities from entity sets.
- In ER diagram, it is represented by a **diamond**.
- In relational model, it is represented by a separate table or by separate column (foreign key).

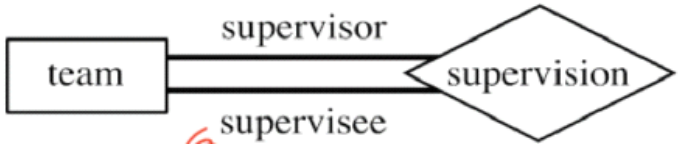
- Every relationship set consists of three components:
  a. Name: Every relationship type must have an unique name.



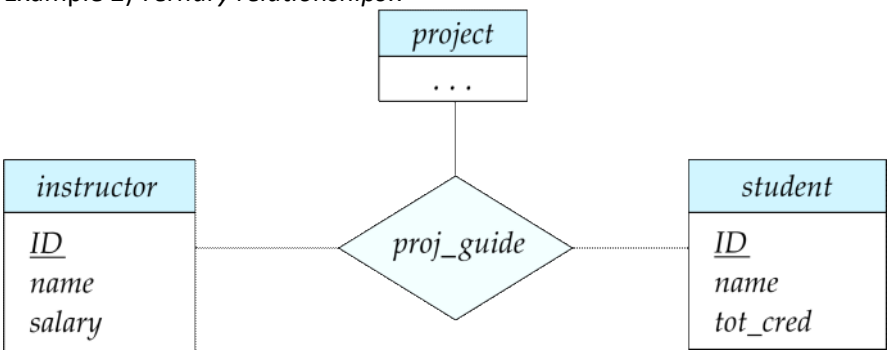
b. Degree:
- It represents the number of entity sets associated with a relationship set.
- Most relationsip sets in ER diagram are binary (degree = 2).
- There are occasions when it is more convenient to represent relationships as non-binary i.e. *n-ary relationship*.

Example 1) *Unary relationships*: One single entity set participate in a Relationship, means two entities of the same entity set are related to each other. These are also called as self -referential relationship set. E.g.- A

member in a team maybe supervisor of another member in team.
Note: Each occurrence of entity set playes a **role** in relationship. In this example) roles of *team* are *supervisor* and *supervisee*.



Example 2) *Ternary relationships:*.



c. Cardinality Mapping Ratio:

- It express the number of entities to which entity can be related via a relationship.
- It can be used in describing relationship set of any degree but it is most useful in binary relationship.
- We express cardinality constraints by drawing either a directed line (->), signifying *one*, or an undirected line (—), signifying *many,* between the relationship set and the entity set.

- For a binary relationship set the mapping cardinality must be one of the following types:

1) One to one

A student is associated with at most one instructor via the relationship advisor
A student is associated with at most one department via stud_dept

2) One to Many *OR* Many to One

An instructor is associated with several (including 0) students via advisor
A student is associated with at most one instructor via advisor

An instructor is associated with at most one student via advisor,
A student is associated with several (including 0) instructors via advisor

3) Many to many

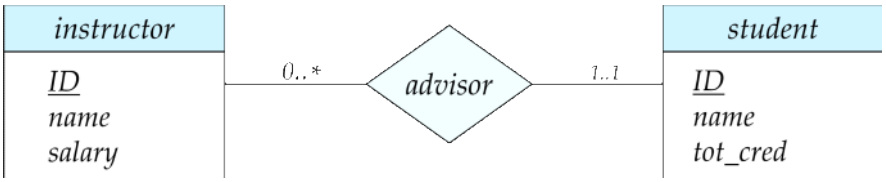
An instructor is associated with several (possibly 0) students via advisor
A student is associated with several (possibly 0) instructors via advisor

d. Participation Constraint:

- It specifies whether the existence of entity depends on its being related to another entity via a relationship type.
- These constraints specify the minimum and maximum number of relationship instances that each entity can/must participate in.

- ***Maximum Cardinality***: It defines the number of maximum times an entity can participate in a relationship.
- ***Minimum Cardinality***: It defines the number of minimum times an entity can participate in a relationship.
- A line may have an associated minimum and maximum cardinality, shown in the form **l..h**, where l is the minimum and h the maximum cardinality.

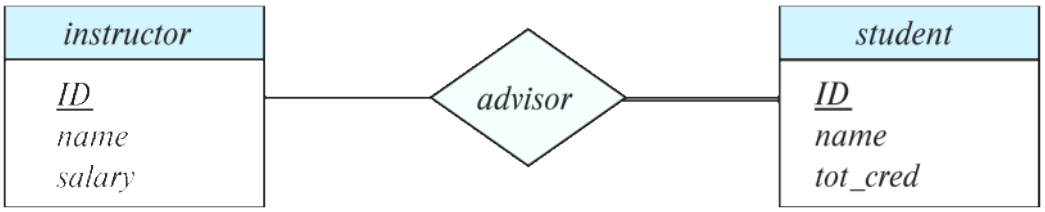
Example) Instructor can advise 0 or more students. A student must have 1 advisor; cannot have 0 or multiple advisors



- **Total Participation:** Every entity in the entity set participates in at least one relationship in the relationship set. It is represented by **double line**.
- **Partial Participation:** Some entities may not participate in any relationship in the relationship set, i.e. there exists atleast one entity which do not participate in relationship. It is represented by **single line**.

Note: If minimum cardinality is greater than or equal to 1, it represents total participation else it is partial participation is minimum cardinality is equal to 0.

Example) Participation of student in advisor relation is total i.e.every student must have an associated instructor whereas participation of instructor in advisor relation is partial i.e. some instructos may not be associated with a student.



Key:

- A key is a subset of attributes of the entity set which can uniquely distinguish/identify each member of the entity set from each other.
- Keys can also help to identify relationships uniquely and thus distinguish relationships from each other.
- The concepts of super key, candidate key and primary key are applicable to entity sets just as they are applicable to relation schemas.

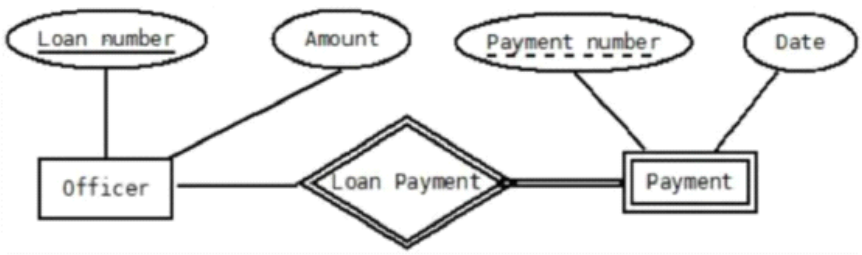
Strong Entity Set:

- It is an entity set which has a primary key, so that all the tuples in the set are distinguishable by that key.
- It is represented by **single rectangle**.
- It may have partial or total participation.
- It contain *primary key* attributes which is represented by oval with attribute name underlined with straight line.

Weak Entity Set:

- It is an entity set which does not possess sufficient attributes to form a primary key, so that each tuple in the set can be uniquely identified.
- Weak entity set is represented by **double rectangle**.
- Weak entity set will always have *total participation (double line)*.
- Weak entity set contains **discriminator attributes** (partial key) which contain partial information about the weak entity set but is not sufficient enough toe identify each tuple uniquely. It is represented as oval with attribute name underlined with **dotted line**.
- For a weak entity set to be meaningful and converted into strong entity set, it must be associated with another entity set called the **identifying or owner entity set** such that weak entity set is said to be existence dependent on the identity set. The primary key of weak entity set will be the union of primary key and discriminator attributes.
- *Note*: Identifying or owner entity set may be strong entity set or weak entity set which has already been converted into strong entity set using another owner entity set.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship** and is represented by **double diamond** in ER diagram.
- *Note*: The identifying relationship set should not have any descriptive attributes, since any such attributes can instead be associated with the weak entity set.

Example: ER Diagram)



Reasons of having Weak Entity Set

- Weak entities reflect the logical structure of an entity being dependent on another.
- Weak entity can be deleted automatically when their strong entity is deleted.
- Without weak entity set it will lead to duplication and consequent possible inconsistencies.

Convert ER Model to Relational Model

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database. A database can be represented by a collection of schemas.

- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set. Each schema has a number of columns/attributes, which have unique names.

- A strong entity set reduces to a schema with the same attributes.
  Eg) *student(ID, name, tot_cred)*

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set.
  Eg) section (course-id, section-id, sem, year)

- *Note*: The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.

- Composite attributes are flattened out by creating a separate attribute for each component attribute.

- A multivalued attribute M of an entity E is represented by a separate schema EM. Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M.
  Example: Multivalued attribute phone_number of instructor is represented by a schema: *inst_phone*= (*ID, phone_number*)

- *Conversion of 1-1 Binary Relationship from ER Diagram to Relational Model:* No separate table is required, take primary key of one side as foreign key on other side. Priority must be given to the side having total participation.

- *Conversion of 1-n or n-1 Binary Relationship from ER Diagram to Relational Model:* No separate table is required, modify n side by taking primary key of 1 side as foreign key on n side.

- *Conversion of m-n Binary Relationship from ER Diagram to Relational Model:* Separate table is required, take primary key of both table and declare their combination as primary key of new table.

- *Note*: If participation is partial on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values.

Article: https://www.gatevidyalay.com/er-diagrams-to-tables/

Functional Dependencies

Problems with one table containing all data/information

1. When same data is stored multiple times in the database, it is known as **data redundancy**, which often leads to **data inconsistency**.
2. There will be increase in size of the database table and thus access time of any record/data will become slow.
3. Anomalies that cause redundant work to be done during insertion, modification or deletion of records/attributes in a database are:
   a. **Insertion Anomalies:** An independent piece of information cannot be recorded into a relation unless an irrelevant information must be inserted together at the same time.
   b. **Modification Anomalies:** The update of a pice of information must occur at multiple locations.
   c. **Deletion Anomalies:** The deletion of a piece of information unintentionally removes other information.

Purpose of Normalization

- Normalization is a refinement process which includes creating tables and establishing relationships between those tables according to rules designed both to protect data and make the database more flexible by eliminating various factors like:
  - Data Redundancy
  - Data Inconsistency
  - Updation(Insertion, Modification & Deletion) Anomalies.
- Without normalization, database system may be inaccurate, slow and inefficient and it might not be able to produce the data we expect.
- Every table must have a single idea and if a table contains more than one idea, then the table must be ***decomposed*** (using tool known as functional dependencies) until each table contains only one idea.

Functional Dependencies:

- It is a formal tool for analysis of relational schemas.
- In a relation R, if $\alpha \subseteq R$ and $\beta \subseteq R$, then attribute or a set of attribute α functionally derives an attribute or set of attributes β, if and only if each α value is associated with precisely one β value.

For all pairs of tuples t_1 and t_2 in R such that
- If $T_1[\alpha] = T_2[\alpha]$
- Then, $T_1[\beta] = T_2[\beta]$

- Here α is the **determinant** (as it determines the value of β), whereas β is the **dependent** (as it is depends on value of α).
- If K -> R, then K is a super key of R.
- *Note*: Functional dependency is a property of relational schema not a particular relational instance.

- Steps to find whether a functional dependency (FD) α -> β can be concluded on a given instance or not are:
  a. Find whether all the values of α are different or not, if not then it is invalid.
  b. Find whether all the values of β for each α are same or not, if not then it is invalid, i.e. find if there are two same values of α having different values of β for an invalid FD.

Types of Functional Dependencies

- **Trivial Funcional Dependency**: If β is a subset of α, then the functional dependency α -> β will always hold. For Eg) AB -> A.
- **Fully Functional Dependency**: An attribute is fully functional dependent on another attribute, if it is Functionally Dependent on that attribute and not on any of its proper subset.
- **Partial Dependency:** When a non-prime attribute is dependent only on a part(proper subset) of candidate key, then it is called partial dependency.
- **Total Dependency:** When a non-prime attribute is dependent on the entire candidate key, then it is called total dependency.
- **Transitive Dependency**: A functional dependency from non-Prime attribute to non-Prime attribute is called transitive dependency.

Attributes Closure/ Closure set of attributes/ Closuer of attributes set

- Attribute closure on an attributes set can be defined as a set of attributes which can be functionally determined from F (either directly through FD or derived logically). It is represented by $\mathbf{F^+}$.

- ***Armstrong Axioms:*** They are the set of statements, which are considered to be true for all relational databases and can be used to generate closure set.
  - Primary rules:
    - **Reflexivity:** If Y is a subset of X, then X -> Y.

        - **Augmentation:** If X -> Y, then XZ -> YZ
        - **Transitivity:** If X -> Y and Y -> Z, then X -> Z.
    - Derived rules:
        - **Union**: If X-> Y and X -> Z, then X -> YZ
        - **Decomposition**: If X -> YZ, then X -> Y and X -> Z.
        - **Pseudo Transitivity**: If X -> Y and WY -> Z, then WX -> Z.
        - **Composition**: If X -> Y and Z -> W, then XZ -> YW.

- ***Equivalence of two FD Sets***: Two FD sets F_1 and F_2 are equivalent if:
  $F_1^+ = F_2^+$ or $F_1 \subseteq F_2^+$ and $F_2 \subseteq F_2^+$

Minimal Cover/Canonical Cover/Irreducible Set

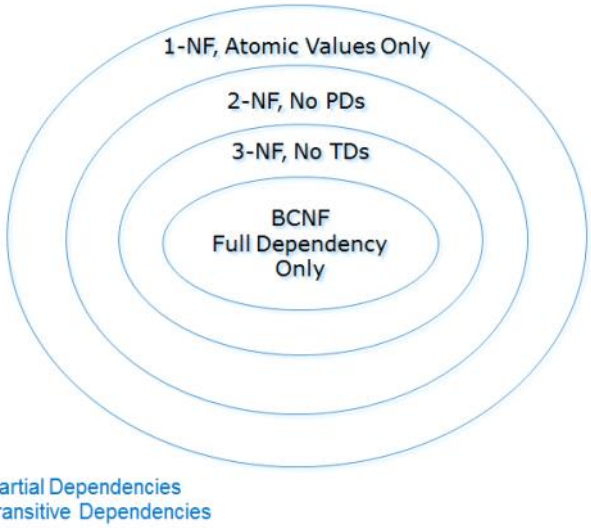
- Minimal cover- It means to eliminate any kind of redundancy from FD set.
- A canonical cover of a set of functional dependencies, F is a simplified set of functional dependencies that has the same closure as the original set F, there may be any following type of redundancy in the set of functional dependencies: -
    - Complete production may be redundant.
    - One more than one attributes may be redundant on right hand side of a production.
    - One or than one attributes may be redundant on Left hand side of a production.

- Procedure to find minimal cover:
    - Use decomposition rule wherever applicable so that RHS of a production/FD contains only single attribute.
    - Remove extraneous attribute on LHS of a production by finding the closure for every possible subset, if in any case the closu re is same it means remaining attributes are redundant.
    - For every production find the closure value of LHS of production keeping the production in a set, and next time ignoring the production to be in a set. If both closure set matches, it means the production is redundant.

Normalization

- Normalization of data can be considered a process of analyzing the given relation schema to achieve the desirable properties of minimizing redundancy using decomposition.
- The tool we use for normalization is functional dependencies and candidate keys. Note: Functional dependency can be used only to normalize upto BCNF.
- A series of normal form tests can be carried out on individual relation schemas so that the relational database can be normalized to any desired degree.

Types of Normal Forms



1. **First Normal Form (1NF)**

- Relational table is said to be in first normal form if only if each attribute in each cell have single value (atomic), i.e. a relation should not contain any multi-valued or composite attributes.

- Every row should be unique, that is no two rows should have the same values of all the attributes. Hence there must be a primary key to uniquely identify each row.

- Every column should have a unique name and the order of rows & columns are irrelevant. Also, attribute's domain should not change.

- *Note*: 1NF cannot solve the update anomalies (insertion, modification and deletion), 2NF and 3NF are used to divide table into multiple tables/relations and hence they only can solve update anomalies.

2. Second Normal Form (2NF)

- Relation R is in 2NF if,
  - R should be in 1 NF.
  - R should not contain any Partial dependency, i.e. every non-key column should be fully dependent upon candidate key

- *Formal Definition*: A relation is said to be in 2NF if it is in 1NF and every non-primary-key attribute is fully functionally dependent on the primary key i.e. relation must not contain any partial dependency.

- Reasons for normalization to 2NF:
  - 2NF applies to relations with composite keys, i.e, relations with a primary key composed of two or more attributes. A relation with a single-attribute primary key is automatically in at least 2NF.
  - A relation that is not in 2NF may suffer from the update anomalies.

- The normalization of 1NF relations to 2NF involves the removal of partial dependencies. If a partial dependency exists, we remove the partially dependent attribute(s) from the relation by placing them in a new relation along with a copy of their

determinant.

3. **Third Normal Form (3NF)**

- Relation R is said to be in 3NF if,
  - R should be in 2NF
  - R must not contain any transitive dependency for non-prime attributes.

- *Formal Definition*: A relational schema R is said to be 3 NF if for every functional dependency in R α --> β, either α is super key or β is the prime attribute.

- Reasons for normalization to 3NF:
  Although Second Normal Form (2NF) relations have less redundancy than those in 1NF, they may still suffer from update anomalies. If we update only one tuple and not the other, the database would be in an inconsistent state. This update anomaly is caused by a transitive dependency (which is to be removed in 3NF).

- The normalization of 2NF to 3NF involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attribute(s) from the relation by placing the attribute(s) in a new relation along with a copy of the determinant.

- <u>*Note*</u>: 3NF is considered *adequate* for normal relational database design because most of the 3NF tables are free of insertion, update, and deletion anomalies. Moreover, 3NF always ensures functional dependency preserving and lossless.

4. **Boycee Codd Normal Form (BCNF)**

- *Formal Definition*: A relational schema R is said to be in BCNF if it is in 3NF and for every functional dependency in R from α --> β, α must be a super key.

- Reasons for Normalization upto BCNF:
  Although 3NF is considered *adequate* normal form as it can identify additional redundancy caused by dependencies that violate one or more candidate keys, dependencies can still exist that will cause redundancy to be present in 3NF relations. 3NF cannot remove all of the redundancy when there is a functional dependency X -> Y and X is not a candidate key of the given relation.

- To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys.

- BCNF decomposition may always not possible with dependency preserving, however, it always satisfies lossless join condition.

- Redundancies are sometimes still present in a BCNF relation as it is not always possible to eliminate them completely.

<u>*Important Notes*</u>:

- If a relation R does not contain any non- trivial dependency, then R Is in BCNF.
- A Relation with two attributes is always in BCNF.
- A relation schema R consist of only simple candidate key then, R is always in 2NF but may or may not be in 3NF or BCNF.
- A Relation schema R consist of only prime attributes then R is always in 3NF, but may or may not be in BCNF.
- A relation schema R in 3NF and with only simple candidate keys, then R surely in BCNF.

<u>Multivalued Dependency</u>

Multivalued dependencies are a consequence of first normal form (1NF) which disallows an attribute in a tuple to have multiple values.
It is denoted by A -> -> B, which means for every value of A, there *may* exist more than one value of B.

A ***trivial multivalued dependency*** X -> -> Y is one where either Y is a subset of X, or X and Y toegether form the whole set of attributes of the relation i.e. X U Y = R. In simple terms, multivalued dependency in a relation is trivial if there is atmax 1 (either 0 or 1) multivalued independent attributes.

If there is a functional dependency X -> Y, then there will also be multivalued dependency X -> -> Y, but converse is not true

always.

If we have two or more multivalued independent attributes in the same relation schema i.e. non-trivial multivalued dependency, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. Hence data redundancy may occur even in BCNF due to multivalued dependency.

Example) Following two tables have trivial multivalued dependency and hence no data redundancy. Also, they are in BCNF.

1. Student Name -> -> Club Name

S_Name	Club_name
Kamesh	Dance
Kamesh	Guitar

2. Student Name -> -> Phone Number

S_Name	P_no
Kamesh	123
Kamesh	789

But the third table having non-trivial multivalued dependency i.e. having 2 multivalued independent attributes are present in the same table and hence cause data redundancy even though it is in BCNF.

3. Student Name -> -> Club Name AND Student Name -> -> Phone Number

S_Name	Club_name	P_no
Kamesh	Dance	123
Kamesh	Guitar	123
Kamesh	Dance	789
Kamesh	Guitar	789

5. Fourth Normal Form (4NF)

- A relation is in 4NF if:
  - It is in BCNF
  - There must not exist any non-trivial multivalued dependency.

- Each Multivalued dependency is decomposed into separate table so that it becomes trivial MVD, and hence data redundancy is avoided.

Lossless vs Lossy Decomposition and Dependency Preserving Decomposition

Relation R should be decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

i) Lossless vs Lossy Decomposition:

- During normalization of a table, it is decomposed into two or more tables, now when we want to join two (or more) tables , then we should get the original table (without normalization) without any extra tuples or less tuples. Hence no information should be lost while retreival of original relation.

- If we decompose a relation R into relations R1 and R2,
  - Decomposition is lossy if R1 ⋈ R2 ⊃ R *or* R ⊃ R1 ⋈ R2: Either there are additional or less tuples in join.
  - Decomposition is lossless if R1 ⋈ R2 = R: Join yields the same relation as original one.

- Lossless decomposition is a *necessary* condition (extremely critical) to be achieved at any cost so that no information is lost in normalization.

- To check for lossless join decomposition using FD set, following conditions must hold:

  a. Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.
  $Att(R1) \cup Att(R2) = Att(R)$

  b. Intersection of Attributes of R1 and R2 must not be NULL. There must be atleast one common attribute for join.
  $Att(R1) \cap Att(R2) \neq \Phi$

  c. Common attribute must be a key (unique for all tuples) for at least one relation (R1 or R2)
  $Att(R1) \cap Att(R2) \rightarrow Att(R1)$ **or** $Att(R1) \cap Att(R2) \rightarrow Att(R2)$

Differences between Lossless and Lossy Decomposition

Lossless	**Lossy**
The decompositions R1, R2, R2…Rn for a relation schema R are said to be Lossless if there natural join results the original relation R.	The decompositions R1, R2, R2…Rn for a relation schema R are said to be Lossy if there natural join results into additon of extraneous tuples with the the original relation R.
Formally, Let R be a relation and R1, R2, R3 … Rn be it's decomposition, the decomposition is lossless if – $R1 \bowtie R2 \bowtie R3 .... \bowtie Rn = R$	Formally, Let R be a relation and R1, R2, R3 … Rn be it's decomposition, the decomposition is lossy if – $R \subset R1 \bowtie R2 \bowtie R3 .... \bowtie Rn$
There is no loss of information as the relation obtained after natural join of decompositions is equivalent to original relation.Thus, it is also referred to as non-additive join decomposition	There is loss of information as extraneous tuples are added into the relation after natural join of decompositions. Thus, it is also referred to as careless decomposition.
The common attribute of the sub relations is a superkey of any one of the relation.	The common attribute of the sub relation is not a superkey of any of the sub relation.

ii) Dependency Preserving Decomposition

- R is decomposed or divided into R1 with FD { f1 } and R2 with { f2 }, the there can be three cases:
  - **f1 U f2 = F** -----> Dependency Preserving Decomposition
  - **f1 U f2** is a subset of F -----> Not Dependency Preserving Decomposition
  - **f1 U f2** is a super set of F -----> This case is not possible

- *Defintiion*: If we decompose a relation R into relations R1 and R2, All dependencies of R either must be a part of R1 or R2 or must be derivable from combination of FD's of R1 and R2.
  $\{ F1 \cup F2 \}^+ = F^+$

- Dependency preserving decomposition, although desirable, is not *necessary* to be present and can be sacrificed sometimes.

6. Fifth Normal Form (5NF)

A relation R is in 5NF if:

- It is in 4NF
- Any further decomposition of the relation will be lossy decomposition, i.e. it cannot be further lossless decomposed.

File Organization

Relative data and information is stored collectively in file formats. A **file** is a sequence of records stored in binary format. A **disk drive** is formatted into several **blocks** that can store records. File Organization defines how file records are mapped onto disk blocks.

Types of File Organization

1. **Heap File Organization**
   When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

2. **Sequential File Organization**
   Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

3. **Hash File Organization**
   Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of disk block where the records are to be placed.

4. **Clustered File Organization**
   Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

For In Depth Knowledge, refer articles:

1. https://www.geeksforgeeks.org/file-organization-in-dbms-set-1/
2. https://www.geeksforgeeks.org/file-organization-in-dbms-set-4/
3. https://www.geeksforgeeks.org/file-organization-in-dbms-set-3/

Types of File Operations

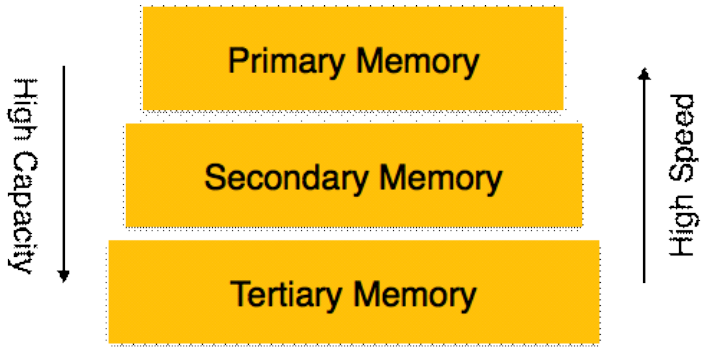
Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files:

- **Open** – A file can be opened in one of the two modes, read mode or write mode. In read mode, the operating system does not allow anyone to alter data. In other words, data is read only. Files opened in read mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.
- **Locate** – Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.
- **Read** – By default, when files are opened in read mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Write** – User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so.
- **Close** – This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
  - removes all the locks (if in shared mode),
  - saves the data (if altered) to the secondary storage media, and
  - releases all the buffers and file handlers associated with the file.

The organization of data inside a file plays a major role here. The process to locate the file pointer to a desired record inside a file various based on whether the records are arranged sequentially or clustered.

Storage System

Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types:



- **Primary Storage** – The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.
- **Secondary Storage** – Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** – Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

Memory Hierarchy

A computer system has a well-defined hierarchy of memory. A CPU has direct access to it main memory as well as its inbuilt registers. The access time of the main memory is obviously less than the CPU speed. To minimize this speed mismatch, cache memory is introduced. Cache memory provides the fastest access time and it contains data that is most frequently accessed by the CPU.

The memory with the fastest access is the costliest one. Larger storage devices offer slow speed and they are less expensive, however they can store huge volumes of data as compared to CPU registers or cache memory.

Magnetic Disks

Hard disk drives are the most common secondary storage devices in present computer systems. These are called magnetic disks because they use the concept of magnetization to store information. Hard disks consist of metal disks coated with magnetizable material. These disks are placed vertically on a spindle. A read/write head moves in between the disks and is used to magnetize or de-magnetize the spot under it. A magnetized spot can be recognized as 0 (zero) or 1 (one).

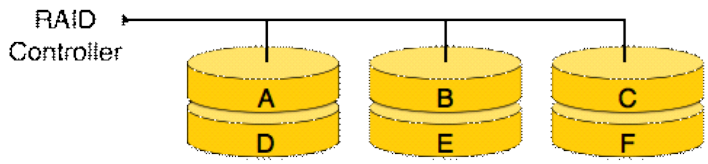
Hard disks are formatted in a well-defined order to store data efficiently. A hard disk plate has many concentric circles on it, called tracks. Every track is further divided into sectors. A sector on a hard disk typically stores 512 bytes of data.

Redundant Array of Independent Disks (RAID)

RAID or Redundant Array of Independent Disks, is a technology to connect multiple secondary storage devices and use them as a single storage media. RAID consists of an array of disks in which multiple disks are connected together to achieve different goals. RAID levels define the use of disk arrays.

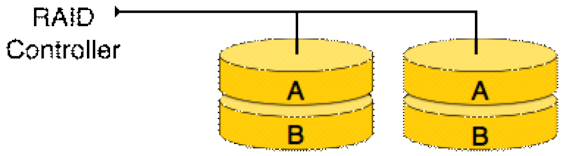
RAID 0

In this level, a striped array of disks is implemented. The data is broken down into blocks and the blocks are distributed among disks. Each disk receives a block of data to write/read in parallel. It enhances the speed and performance of the storage device. There is no parity and backup in Level 0.



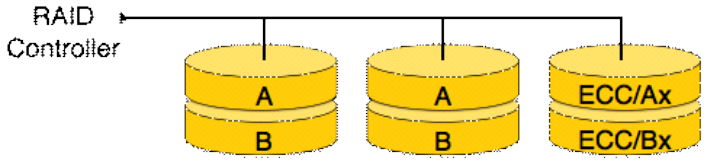
RAID 1

RAID 1 uses mirroring techniques. When data is sent to a RAID controller, it sends a copy of data to all the disks in the array. RAID level 1 is also called mirroring and provides 100% redundancy in case of a failure.



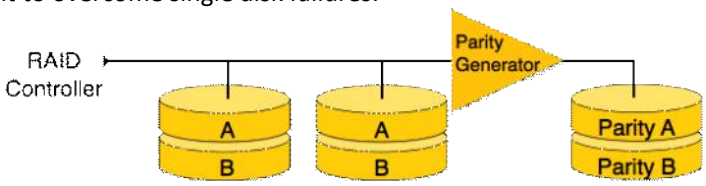
RAID 2

RAID 2 records Error Correction Code using Hamming distance for its data, striped on different disks. Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on a different set disks. Due to its complex structure and high cost, RAID 2 is not commercially available.



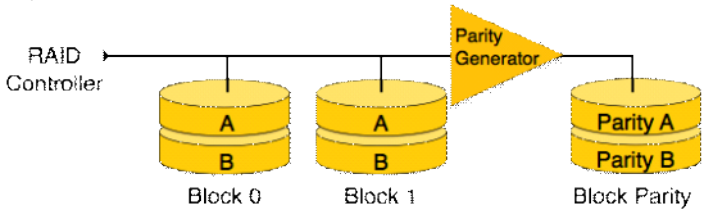
RAID 3

RAID 3 stripes the data onto multiple disks. The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failures.



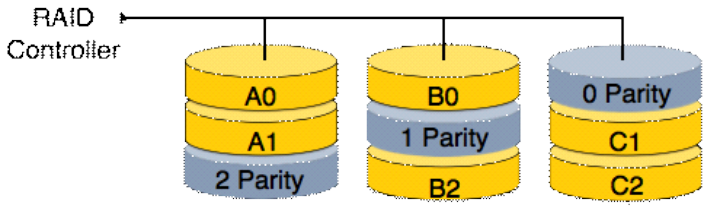
RAID 4

In this level, an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. Note that level 3 uses byte-level striping, whereas level 4 uses block-level striping. Both level 3 and level 4 require at least three disks to implement RAID.



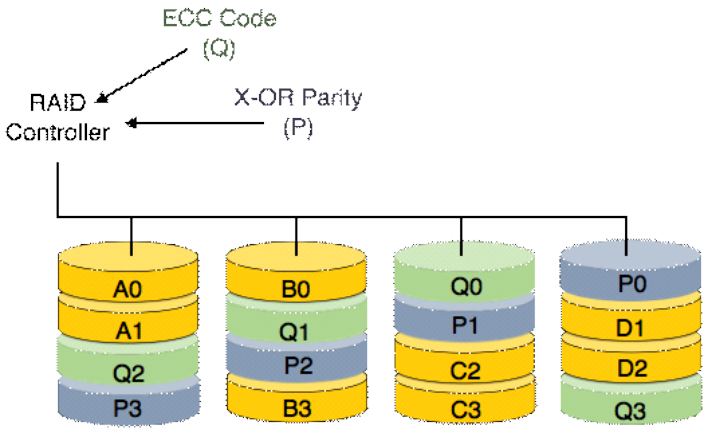
RAID 5

RAID 5 writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.



RAID 6

RAID 6 is an extension of level 5. In this level, two independent parities are generated and stored in distributed fashion among multiple disks. Two parities provide additional fault tolerance. This level requires at least four disk drives to implement RAID.


ECC Code
(Q)
RAID
Controller
X-OR Parity
(P)
A0
A1
Q2
P3
B0
Q1
P2
B3
Q0
P1
C2
C3
P0
D1
D2
Q3

Indexing

- Theoretically relational database is derived from set theory, and in a set the order of elements in a set is irrelevant, so does in relations(tables). But in practice implementation we have to specify the order.
- A number of properties, like search, insertion and deletion will depend on the order in which elements are stored in the tables.
- There are only two ways in which elements can be stored in a table ordered (Sorted) or unordered (Unsorted).

Types of File Organization

1. **Ordered File Organization**

- All the records in the file are ordered on some search key field.
- *Advantage*: Binary Search is possible here. Hence, data search and retreival is faster.
- *Disadvantage*: Maintenance (insertion of new data/ deletion of old data) is costly, as it requires the reorganization of entire file.
- *Note*: Binary search will only work here when we are using the same key for searching on which indexing is done, otherwise file will behave as unordered file.

2. **Unordered File Organization**

- All the records are inserted usually in the end of the file, i.e. no ordering is implemented according to any field.
- *Disadvantage*: Since file is unordered, binary search is not possible or only linear search is possible. Hence data retreival is slow.
- *Advantage*: Maintenance (Insertion & Deletion) is easy, as it does not require re-organization of entire file.

Note: Without Indexing:

- If file is unordered then no of block assesses required to reach correct block which contain the desired record is O(n), where n is the number of blocks.

- If file is ordered then no of block assesses required to reach correct block which contain the desired record is $O(\log_2 n)$, where n is the number of blocks.

Indexing

Indexes*:*

- Indexes are additional auxiliary access structure which provides a data technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.

- Index typically provides secondary access path or an alternay way to access the records without affecting the physical placement of records in the main file.

- Indexes can be created on any field/attribute whether it is prime attribute or non-prime/non-key attribute. This attribute is known as search key.

Index File:

An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Two basic kinds of indices:
  a. Ordered indices: search keys are stored in sorted order
  b. Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”

- The size of index file should be way smaller than that of main file, as index file record contain only two columns, key(attribute on which searching is done) and block pointer (base address of the block of main file which contains the record holding the key), while main file contains all the columns.

- One index file of a relation/table can correspond to only one attribute, hence there can be multiple index files designated for a single main file, differing in the index attribute.

- Index file is always ordered, irrespective of whether main file is ordered or unordered. Hence binary search is always possible on the index file.

  Hence, number of access to search the correct block of the main file using indexing is: $\log_2$*(number of entries in index file) + 1.*

(+1) is required as there need to be one block access for the index file itself.

Constraints associated with Indexing

- **Access Types**: This refers to the type of access such as value based search, range access, etc.
- **Access Time**: It refers to the time needed to find particular data element or set of elements.
- **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: It refers to the additional space required by the index.

Advantages or Reason for Indexing:

- For a large table, with many tuples/records, and hence large number of blocks, data access and retreival without indexing becomes slow. Indexing gives the advantage of *faster data retreival* time.

  Hence, indexing in SQL speeds up SELECT and UPDATE (with WHERE clause) queries.

- Indexing helps to make a row unique or without duplicates (primary or unique values).

Disadvantages of Indexing:

- Space taken by index file will be an overhead, as index file is nothing but a *metadata* (data about data).

- For faster retreival using indexing, records in the main file's blocks should be stored in *unspanned* fashion, i.e. if last record in a block can fit only partially, then it has to be placed in the next block. It will lead to *internal fragmentation*.

- Since index file is sorted, any record when inserted/modified/deleted in the main file will require modification of index file as block in which the records are placed may change.

  Hence, indexing in SQL slows down INSERT, DELETE and UPDATE (without WHERE clause) queries.

<u>Types of Indexing (on the basis of number of entries in index file):</u>

1. **Sparse Index**
   - If an index entry is created only for some records of the main file, then it is called sparse index.

   - Number of index entries in the index file are less than number of records in the main file.

2. **Dense Index**
   - In dense index, there is an entry in the index file for every search key value in the main file (only unique values of the search attribute). This makes searching faster but requires more space to store index records itself.

   - Note that it is not for every record, it is for every search key value. Sometime number of records in the main file > number of search keys in the main file, for example if search key is repeated.

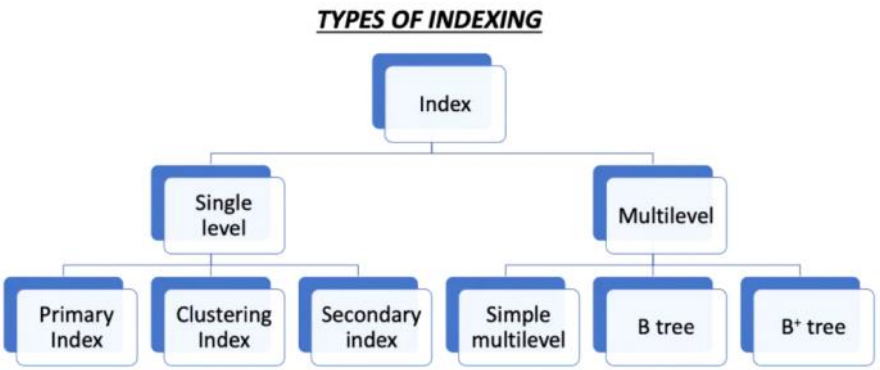
<u>*Note*</u>: Dense & Sparse indexing are not complementary to each other, sometimes it is possible that an indexing is both dense and sparse.

Basic Terms used in Indexing

Blocking Factor = Number of Records per Block = *floor*(block size in bytes / record size in bytes)

Number of blocks required by a file (main or index) = *ceil*(number of records / blocking factor)

<u>Types of Indexing (on the basis of Level of Indexing):</u>

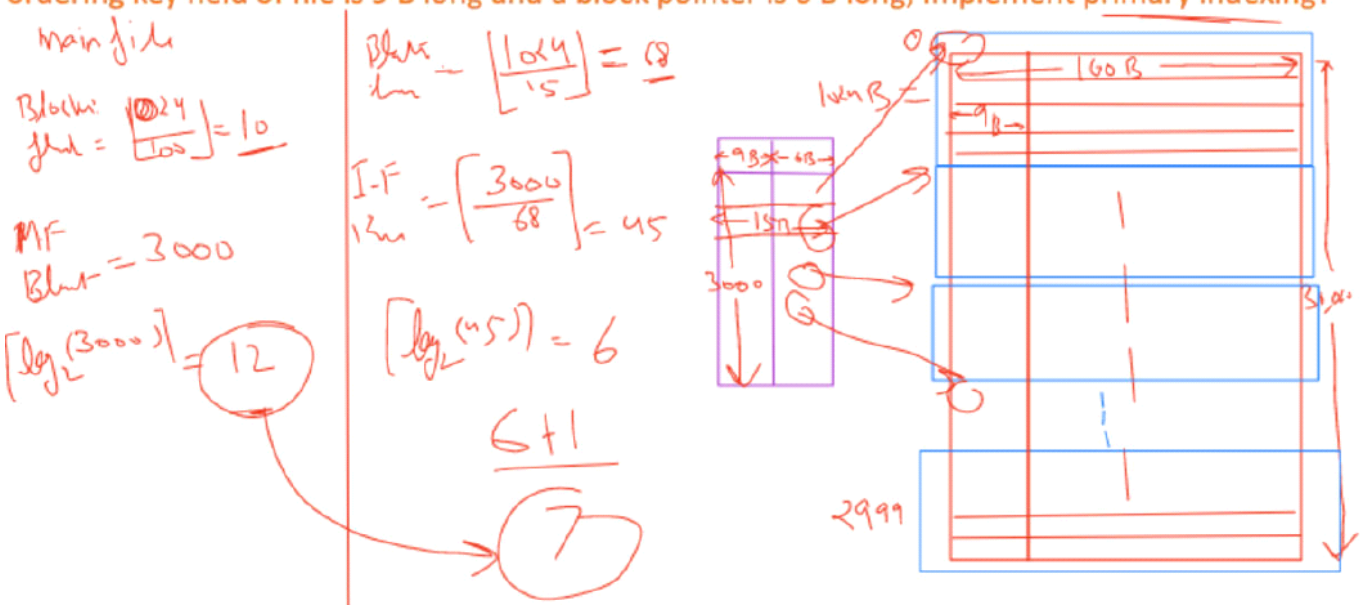


1. **Single Level Indexing:** It refers to the creation of only one index file for the main file which will take space less than or equal to 1 block space.

   a. ***Primary Indexing:***
   - Main file is always sorted according to the primary key.
   - Indexing is done on primary key, therefore called as primary indexing.
   - Index file have two columns, first primary key and second anchor pointer (base address of block).
   - Here first record (anchor record) of every block gets an entry in the index file, hence No. of entries in the index file = No of blocks acquired by the main file.
   - Since only first record of every block is stored in the index file, i.e. all records are not stored in the index file, it is an example of *sparse indexing*.

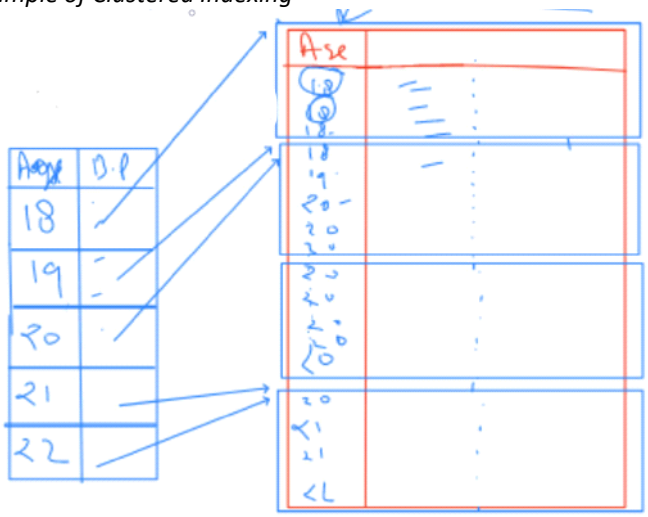
Example Numerical on Primary Indexing

Q Suppose we have ordered file with records stored r = 30,000 on a disk with Block Size B = 1024 B. File records are of fixed size and are unspanned with record length R = 100 B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement primary indexing?



   b. ***Clustered Indexing:***
   - Main file will be ordered on some non-key attributes.
   - Since non-key attributes need not be unique, they can contain duplicate records for the same attribute.
   - We need to store only unique values of attributes in the index file, hence No of Entries in index file = No of unique values of attribute on which indexing is done.
   - Since, all records are not stored in the index, file it is sparse indexing. Since all unique values of the non-key attributes are stored in the index file, it is also dense indexing.
     Hence, clustered indexing is an example of both sparse indexing and dense indexing.

Example of Clustered Indexing

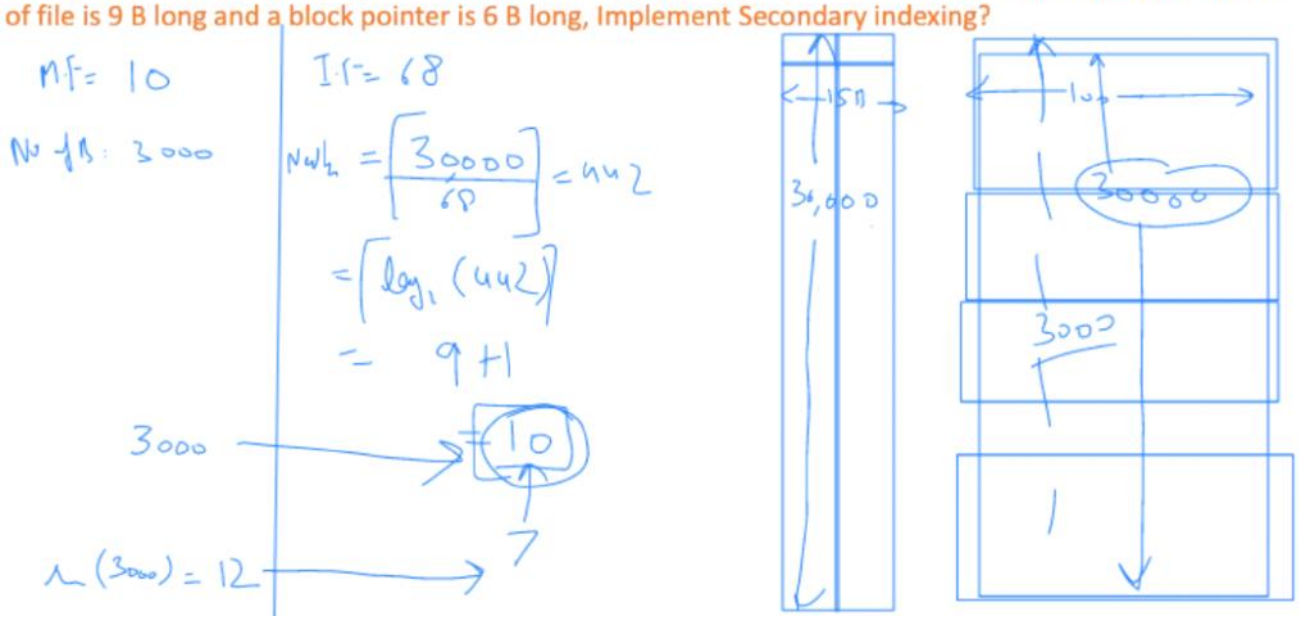


c. Secondary Indexing:

- Most common scenarios, suppose that we already have a primary indexing on primary key, but there is frequent query on some other attributes, so we may decide to have one more index file with some other attribute.
- Main file is ordered only according to the attribute on which indexing is done but *unordered* on the attribute on which search queries are frequent.
- Secondary indexing can be done on key or non-key attribute.
- No of entries in the index file = Number of entries in the index file, as main file is unordered on the search key attribute, hence all records need to be inserted into the index file of secondary indexing.
- It is an example of dense indexing, as records with all values of attribute are stored in the index file.

Example Numerical on Secondary Indexing

Q Suppose we have ordered file with records stored r = 30,000 on a disk with Block Size B = 1024 B. File records are of fixed size and are unspanned with record length R = 100 B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement Secondary indexing?



Note: In SQL terms, both primary indexing and clustered indexing are collectively termed as clustered indexing while secondary indexing is termed as non-clustered indexing.

2. Multi Level Indexing:

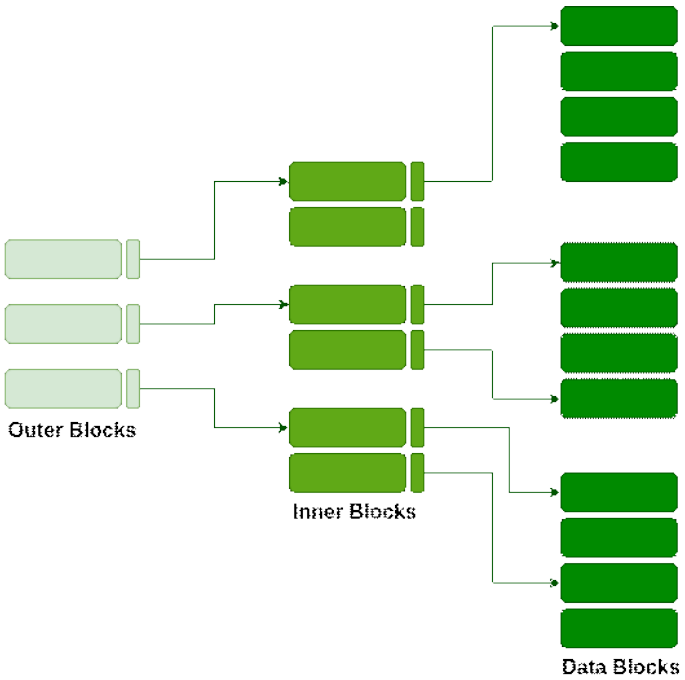
- With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses.

It refers to the creation of multiple index files for the main file, i.e. further indexing the index file until the last index file fits in 1 block space.

a. Simple MultiLevel Indexing:

- Multi-level indexing helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accomodated anywhere in the main memory.

- The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Need of B and B+ Trees

- There are number of options in data structure like array, stack, link list, graph, table etc. but we want a data structure which support frequent insertion deletion, and modify it self accordingly but at the same time also provide speed search and give us the advantage of having a sorted data.

- If we look at the data structures option then tree seem to be the most appropriate but every kind of tree in the available option have some problems either simple tree or binary search tree or AVL tree, so we end up on designing new data structure called B tree and B+ tree which are kind of specially designed for sorted stored index files in databases.

- In general, with multilevel indexing, we require dynamic structure, b and b+ tree is generalized implementation of multilevel indexing, which are dynamic in nature, that is increasing and decreasing number of records. In the first level index file can be easily supported by other level index.

- B tree and B+ tree also provides efficient search time, as the height of the structure is very less and they are also perfectly balanced.

b. B Tree:

- A B-tree is a self-balancing tree data structure that maintains sorted data that allows searches, sequential access, insertions and deletions in logarithmic time.
- A search tree of order p is a tree such that each node contains at most p-1 search values and p pointers (tree pointers/ block pointers).
- Each Pi is a pointer to a child node (or a NULL pointer), and each Ki is a search value from some ordered set of values. All search values are assumed to be unique.

Properties of B Tree

B-tree of order m (non-empty) is a self-balancing m-way search tree with following properties:

- The root has at least zero child nodes and at most m child nodes.
- The internal nodes except the root have at least celling(m/2) child nodes and at most m child nodes.
- The number of keys in each internal node is one less than the number of child nodes and these keys partition the subtrees of the nodes in a manner similar to that of m-way search tree.
- All leaf nodes are on the same level(perfectly balanced).

Root

Rules	MAX	MIN
CHILD	m	0
DATA	m-1	1

Internal except Root

Rules	MAX	MIN
CHILD	m	⌈m/2⌉
DATA	m-1	⌈m/2⌉ - 1

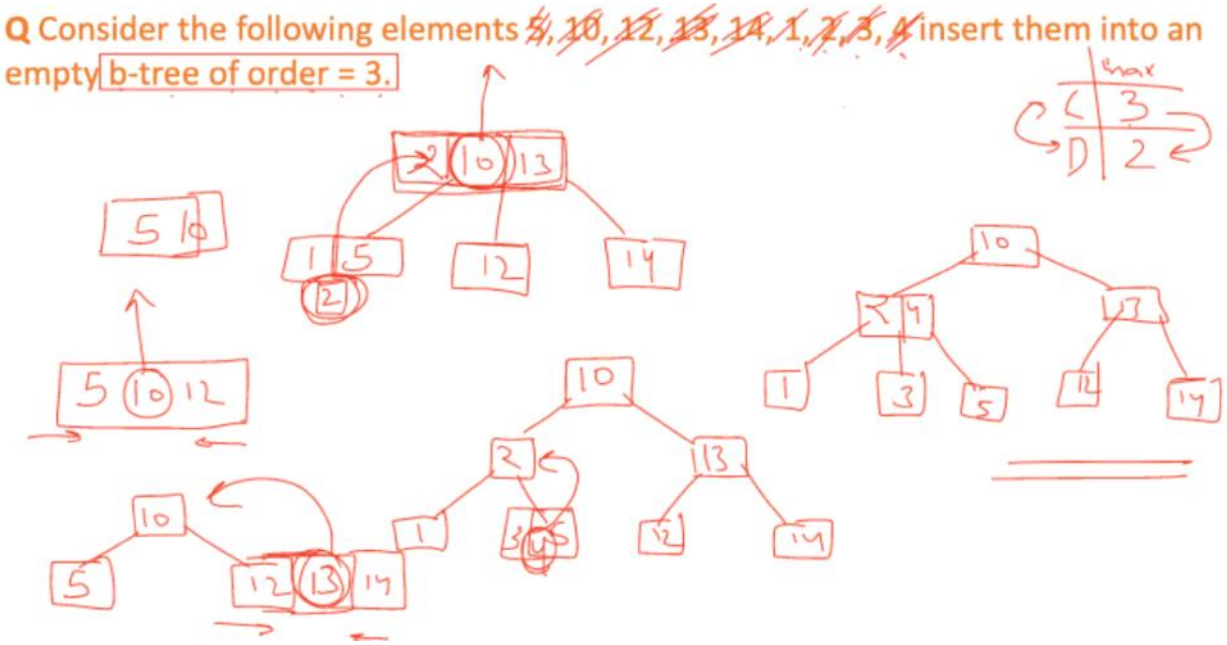
Leaf

Rules	MAX	MIN
CHILD	0	0
DATA	m-1	⌈m/2⌉ - 1

Insertion in B Tree

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.



Deletion in B Tree

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Time Complexity of B Tree Operations

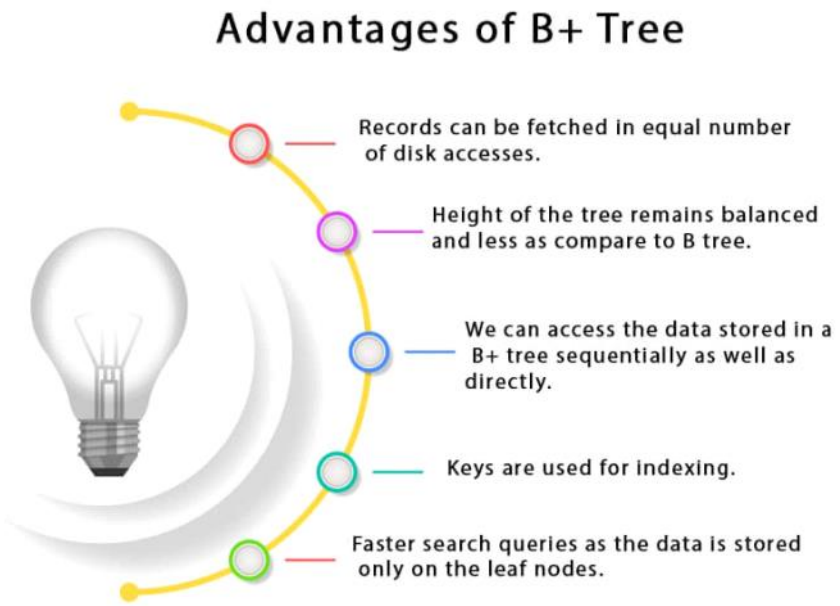
Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Drawback of B Tree

The drawback of B-tree used for indexing is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.

c. B+ Tree:

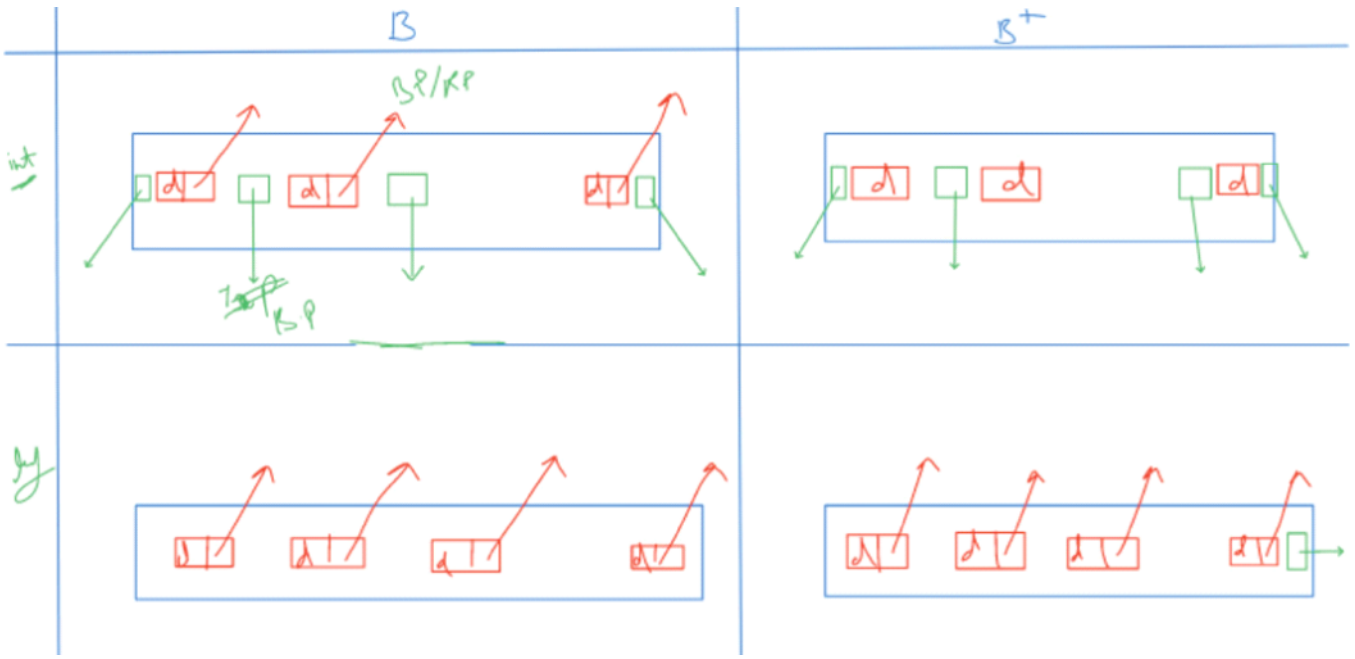
- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.
- The internal nodes of B+ tree are often called index nodes.



B Tree vs B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.

2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

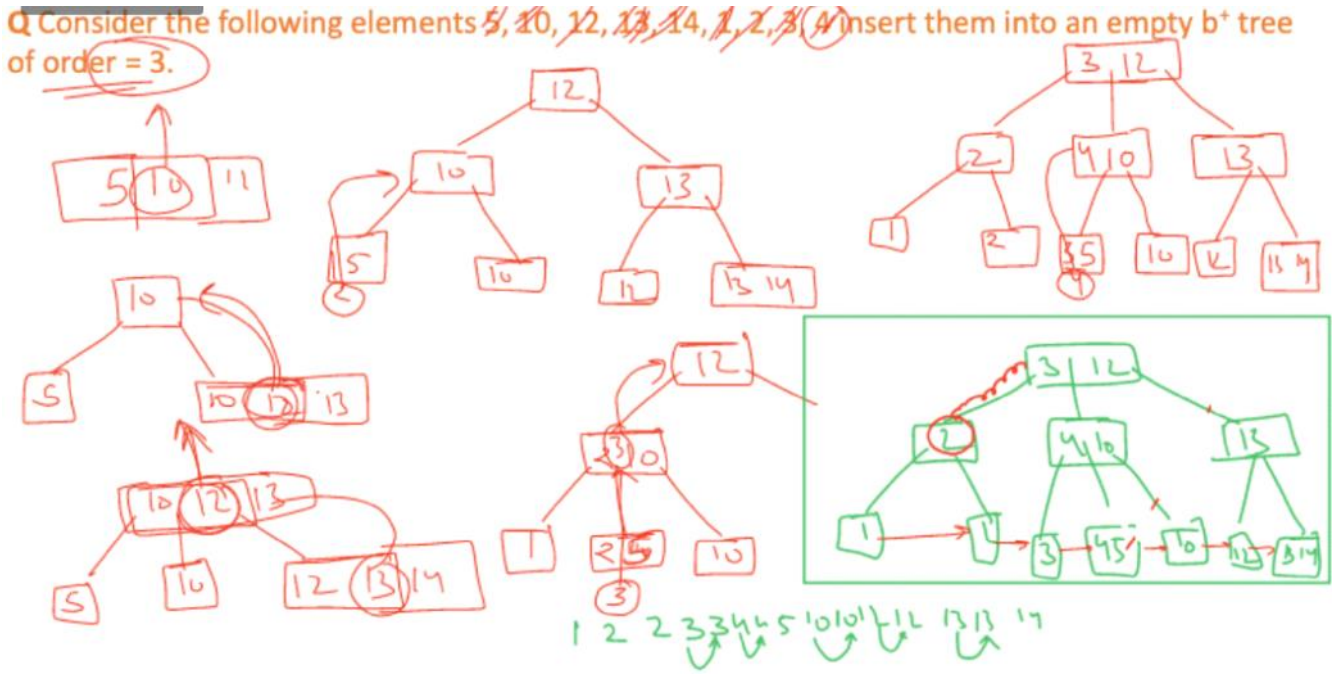


Insertion in B+ Tree

Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.



Deletion in B+ Tree

Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Hashing

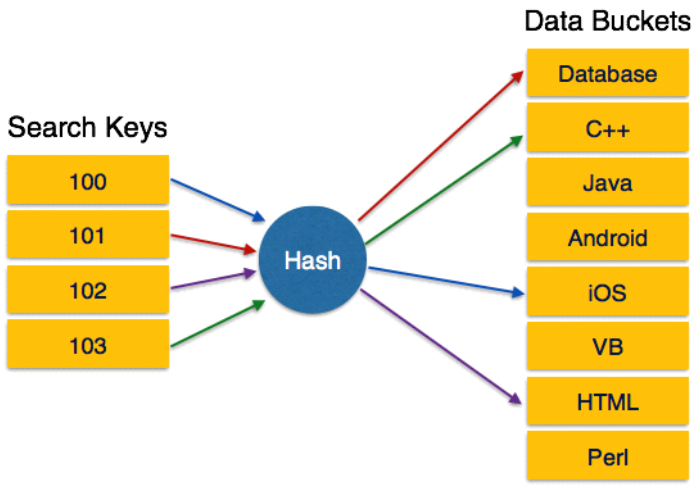
For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.
Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** — A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** — A hash function, h, is a mapping function that maps all the set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



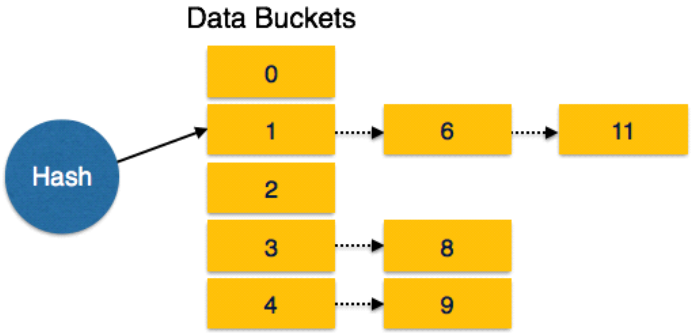
Operation

- Insertion — When a record is required to be entered using static hash, the hash function h computes the bucket address for search key K, where the record will be stored.
  Bucket address = h(K)
- Search — When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- Delete — This is simply a search followed by a deletion operation.

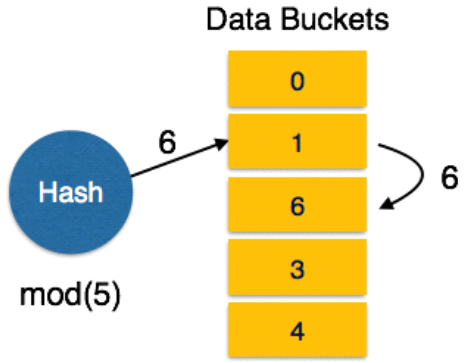
Bucket Overflow

The condition of bucket-overflow is known as collision. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** — When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.



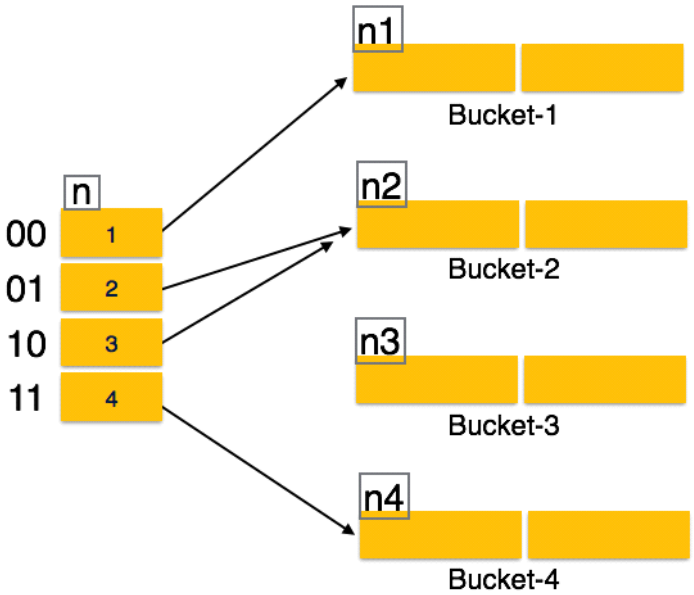
- **Linear Probing** — When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed — that is, when all the buckets are full — then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying** — Look at the depth value of the hash index and use those bits to compute the bucket address.
- **Update** — Perform a query as above and update the data.
- **Deletion** — Perform a query to locate the desired data and delete the same.
- **Insertion** — Compute the address of the bucket
  - If the bucket is already full.
    - Add more buckets.
    - Add additional bits to the hash value.
    - Re-compute the hash function.
  - Else
    - Add data to the bucket,

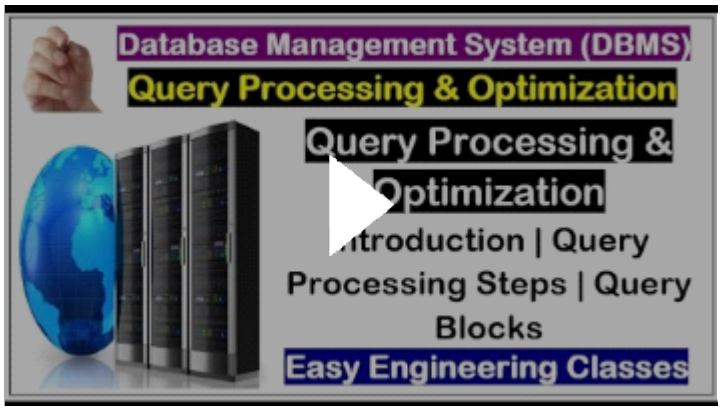
- If all the buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.
Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

Query Processing

Refer Article
https://www.javatpoint.com/query-processing-in-dbms

Refer Video
Query Processing & Optimization Introduction | Query Processing Steps | Query Blocks



Transactions

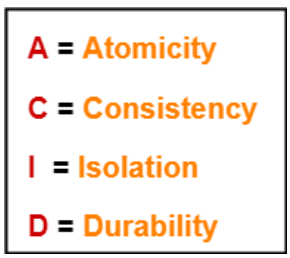
- **Transaction**: Transaction is a set of logically related instructions to perform a logical unit of work.
- A DBMS operation is atomic in nature, i.e. either entire set of instructions should be executed or none of the instruction (s ome of the instructions if ran before failure must be rollbacked).

Operations in Transactions
The main operations in a transaction are-

1. Read Operation:
   Read operation reads the data from the database and then stores it in the buffer in main memory.
   For example- Read(A) instruction will re ad the value of A from the database and will store it in the buffer in main memory.

2. Write Operation:
   Write operation writes the updated data value back to the database from the buffer.
   For example- Write(A) will write the updated value of A from the buffer to the database.

ACID Properties
Transactions should possess several properties, often called the ACID properties to provide accuracy, completeness and data integrity in the database. The ACID properties are **A**tomicity, **C**onsistency, **I**solation and **D**urability.



1. **Atomicity:**
   - A transaction is an atomic unit of processing, i.e. it should be performed in its entirety or not performed at all.
   - It is also referred as "All or Nothing" Rule.
   - It is the responsibility of ***transaction control manager*** (or recovery control manager) of DBMS to ensure atomicity.

2. **Consistency**:
   - A transaction should be consistency preserving, i.e. if it is completely executed from beginning to end without interference from other transactions, it should take database from one consistent state to another.
   - This property ensures that integrity constraints are maintained.
   - It is the responsibility of ***application programmer*** or DBMS modules that enforces integrity constraints.

3. **Isolation**:
   - This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.
   - During execution, each transaction feels as if it is getting executed alone in the system. A transaction does not realize that there are other transactions as well getting executed parallely.
   - Changes made by a transaction becomes visible to other transactions only after they are written in the memory.
   - The resultant state of the system after executing all the transactions is same as the state that would be achieved if the transactions were executed serially one after the other.
   - The isolation property of database is the responsibility of ***concurrency control manager*** of database.

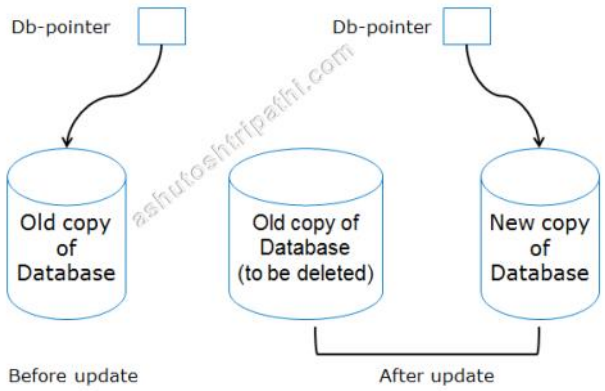
4. **Durability**:
   - This property ensures that all the changes made by a transaction after its successful execution are written successfully to the disk.
   - It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.

- It is the responsibility of ***recovery control manager*** of DBMS to ensure durability.

Q) Give a scheme used by recovery control manager to ensure atomaticity and durability in DBMS transactions.

R) Shadow Copy Technique

- This scheme is based on making copies of database called as shadow copies.
- A pointer called DB pointer is maintained and it points to current copy.
- In this technique, a transaction that wants to update the DB first creates a copy of complete DB.
- All updates are done on the new copy of DB leaving original copy untouched.
- If transaction completes, it is commited as follows:
  - OS is asked to make sure all the updates of new copy of DB are written out to the disk.
  - After OS has written successfully to disk, the db-pointer is updates and points to the new copy of DB and this copy now becomes current copy.



Shadow-copy technique for atomicity and durability

Transaction States

- **Active** – The initial state, where the transaction stays while it is executing its' operations is known as active state.

  All the changes made by the transaction now are stored in the buffer in main memory.

- **Partially committed** – After the final statement has been executed, the state of a transaction is partially committed as it may be possible that it may have to be aborted due to any failure.

  Hence the actual output may still be residing in buffer in main memory (local) and not to the disk (database).

- **Failed -** After the discovery that transaction can no longer proceed it's normal execution, because of hardware/logical errors, such a transaction is said to be in failed state which must be rolled back.

- **Aborted** – After the transaction has been rolled back and the database is restored to its state prior to the start of the transaction, the transaction is said to be in aborted state.

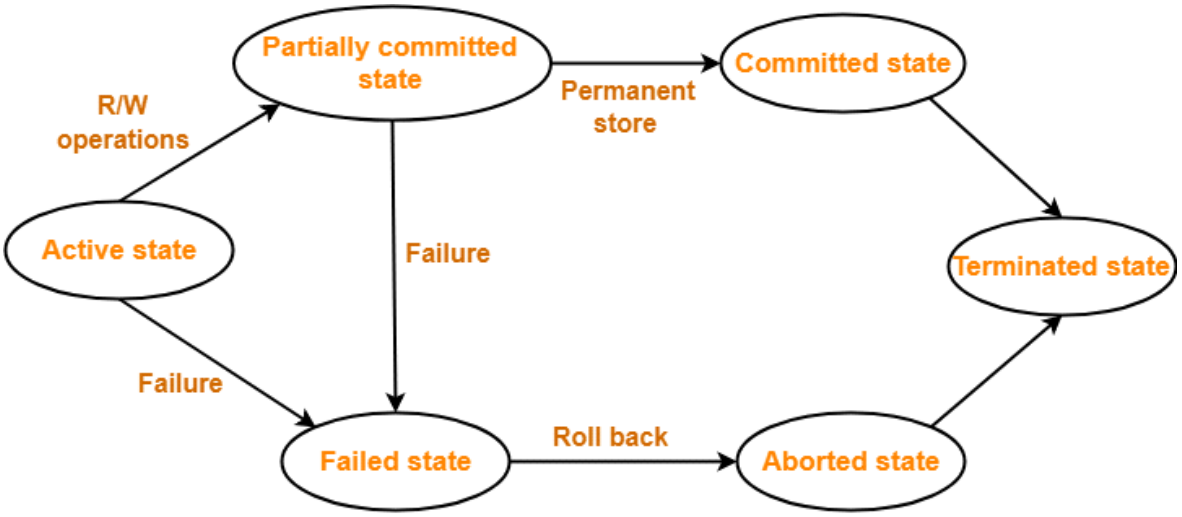
  There are two options after a transaction has been aborted:

  - Restart the transaction (It can be done only if there is no internal logical error)
  - Kill the transaction.

- **Committed** – Transaction is said to be in committed state after it's successful completion and final updation in the database.

- **Terminated** - This is the last state in the life cycle of a transaction. After entering the committed state or aborted state, the transaction finally enters into a terminated state where its life cycle finally comes to an end.

Note: After a transaction has entered the committed state, it is not possible to roll back the transaction, i.e. it is not possible to undo the changes that has been made by the transaction. This is because the system is updated into a new consistent state. The only way to undo the changes is by carrying out another transaction called as ***compensating transaction*** that performs the reverse operations.



Transaction States in DBMS

Advantages of Concurrency

Advantages of running multiple transactions concurrently in the system are:

- **Less Waiting Time** which leads to good database performance.
- **Increased processor and disk utilization**, leading to better transaction *throughput.* E.g., one transaction can be using the CPU while another is reading from or writing to the disk.
- **Reduced average response time** for transactions: short transactions need not wait behind long ones.

Problems of Concurrency

Interleaving of instructions between transactions may lead to certain problems that can cause data inconsistency in the database even after the individual transactions satisfing ACID properties.

1. **Dirty Read Problem (or Read-Write Problem):**
   - Reading the data written by an uncommitted transaction is called as dirty read.
   - This read is called as dirty read because there is always a chance that the uncommitted transaction might roll back later.
   - Thus, uncommitted transaction might make other transactions read a value that does not even exist.
   - Hence, if there is any rollback, it ***may*** lead to inconsistency of the database.

   Example)

Transaction T1	Transaction T2
R (A)	
W (A)	
⋮	R (A) // Dirty Read
	W (A)
	Commit
Failure	

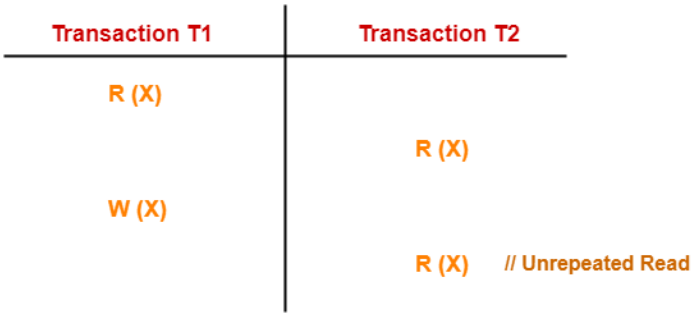
   T1 reads the value of A. T1 updates the value of A in the buffer. T2 reads the value of A from the buffer. T2 writes the updated the value of A. T2 commits. T1 fails in later stages and rolls back.

   T2 reads the dirty value of A written by the uncommitted transaction T1. T1 fails in later stages and roll backs. Thus, the

value that T2 read now stands to be incorrect. Therefore, database becomes inconsistent.

2. **Unrepeatable Read Problem:**
   This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different re ad operations even when it has not updated its value.

   Example)

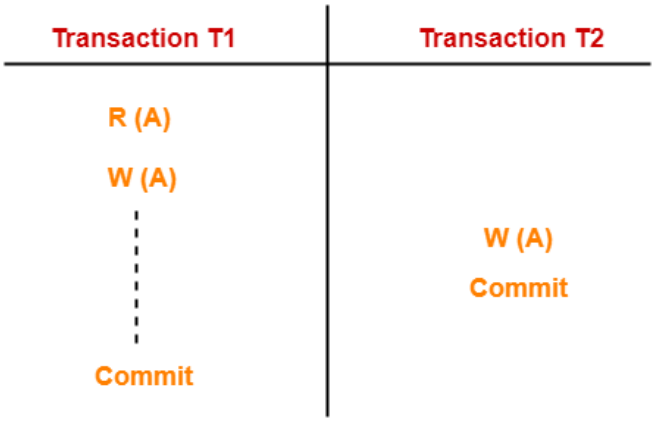


   T1 reads the value of X (= 10 say). T2 reads the value of X (= 10). T1 updates the value of X (from 10 to 15 say) in the buff er. T2 again reads the value of X (but = 15).

   T2 gets to read a different value of X in its second reading. T2 wonders how the value of X got changed because according to it, it is running in isolation.

3. **Lost Update Problem (or Write-Write Problem):**
   This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.
   In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

   Example)



   T1 reads the value of A (= 10 say). T2 updates the value to A (= 15 say) in the buffer. T2 does blind write A = 25 (write wit hout read) in the buffer. T2 commits. When T1 commits, it writes A = 25 in the database.

   T1 writes the over written value of X in the database. Thus, update from T1 gets lost.

4. **Phantom Read Problem:**
   This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it find s that the variable does not exist.

   Example)

Transaction T1	Transaction T2
R (X)	
	R (X)
Delete (X)	
	Read (X)

T1 reads X. T2 reads X. T1 deletes X. T2 tries reading X but does not find it.

T2 finds that there does not exist any variable X when it tries reading X again. T2 wonders who deleted the variable X becaus e according to it, it is running in isolation.

5. **Incorrect Summary Issue**
   This problem 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.

   Example) Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

T1	T2
	sum = 0 read_item(A) sum = sum + A
read_item(X) X = X - N write_item(X)	
	read_item(X) sum = sum + X read_item(Y) sum = sum + Y
read_item(Y) Y = Y + N write_item(Y)	

Note: ***Concurrency Control Protocols*** help to prevent the occurrence of above problems and maintain the consistency of the database. It controls the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules

- When two or more transaction executed together or one after another then they can be bundled up into a higher unit of execution called schedule. The order in which the operations of multiple transactions appear for execution is called as a schedule.
- Operations from different transactions can be interleaved in the schedule. However, schedule for a set of transaction must contain all the instruction of those transaction, and for each transaction Ti, the operations of Ti in schedule must appear in the same order in which they occur in Ti.

Serial Schedule

- In serial schedules all the transactions execute serially one after the other.
- When one transaction executes, no other transaction is allowed to execute, hence there is no *concurrency*.
- Serial schedules are always *consistent, recoverable, cascadeless* and *strict*.
- For a set of n transactions, there exist **n!** different valid serial schedules.
- Since, there is no concurrency, throughput of system is less.

Example)

Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

There are two transactions T1 and T2 executing serially one after the other. Transaction T1 executes first. After T1 completes its execution, transaction T2 executes. So, this schedule is an example of a Serial Schedule.

Non Serial Schedules

- In non-serial schedules, multiple transactions execute concurrently. Operations of all the transactions are inter leaved or mixed with each other.
- Non-serial schedules may or may not be consistent, may or may not be recoverable, may or may not be cascadeless and may or may not be strict.

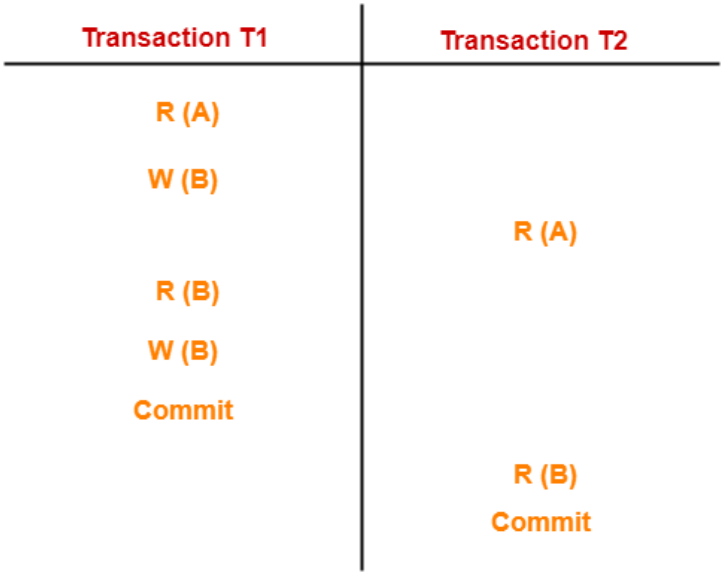
- *Number of Non Serial Schedules:*
  Consider there are n number of transactions T1, T2, T3 …. , Tn with N1, N2, N3 …. , Nn number of operations respectively.

  Total number of possible schedules (serial + non-serial) is given by:

$$\frac{(N1 + N2 + N3 + ... + Nn)!}{N1! \times N2! \times N3! \times .... \times Nn!}$$

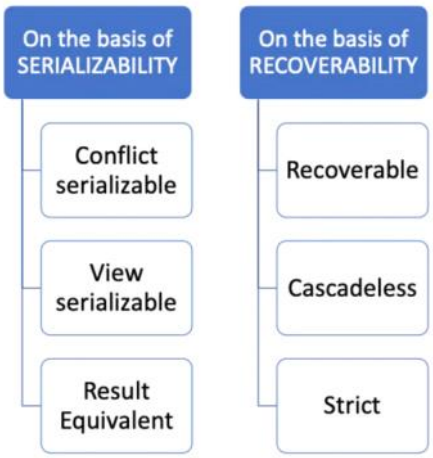
  Since there are n! serial schedules, hence the number of non serial schedules are Total number of schedules – number of serial schedules, i.e. total schedules - n! .

  Example)



There are two transactions T1 and T2 executing concurrently. The operations of T1 and T2 are interleaved. So, this schedule is an example of a Non-Serial Schedule.

Types of Schedules



Serializibility and Recoverability are two different factors to check consistency of a schedule. A serializable schedule may or may not be recoverable. Similarly, a recoverable schedule may or may not be serializable.

Serializability

- If a schedule is serial, then it will always be consistent, but if the schedule is non-serial, it does not mean that it will always be inconsistent. Hence non-serial schedule may (or may not) lead to inconsistency.
- ***Serializability*** is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.
- A non-serial schedule can be proved to be consistent if it is shown to be logically equivalent to a serial schedule. If a non-serial schedule is logically equivalent to a serial schedule, then it is known as ***serializable schedule***.

Important Note: Since properties of recoverability, cascadelessness, strict cannot be defined by serializability, hence serializable schedules may or may not be recoverable, cascadeless, or strict.

Serial vs Serializable Schedule

Serial Schedules	Serializable Schedules
No concurrency is allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
Serial schedules lead to less resource utilization and less CPU	Serializable schedules improve both resource utilization and CPU

throughput. Hence they are less efficient.	throughput, due to concurrency.
Serial Schedules are always consistent. Hence we try to make non-serial schedules logically equivalent to serial schedule (serializable).	Serializable Schedules are always better than serial schedules, since they are consistent, and more efficient.

Conflict Serializable Schedule

- ***Conflicting instructions:***
  Let I and J be two consecutive instructions belonging to two different transactions Ti and Tj in a schedule S, the possible I and J instruction can be as:

  - I = READ(Q), J = READ(Q) : Non-conflicting
  - I = READ(Q), J = WRITE(Q) : Conflicting
  - I = WRITE(Q), J = READ(Q) : Conflicting
  - I = WRITE(Q), J = WRITE(Q): Conflicting

  So, the instructions I and J are said to be conflicting, if they are operations by ***different*** transactions on the ***same data item***, and ***at least one*** of these instructions is a ***write*** operation.

  Example) W1 (A) and R2 (A) are called as conflicting operations:

Transaction T1	Transaction T2
R1 (A)	
W1 (A)	
	R2 (A)
R1 (B)	

  Note: Two instructions of different transactions are non-conflicting if they operate on different data items.
  For eg) I = READ(x) or WRITE(x) and J = READ(y) or WRITE(y) are non-conflicting since I and J are working on different data items x and y.

- ***Conflict Equivalent:***
  If one schedule can be converted to another schedule by swapping of non-conflicting instructions, then they are called conflict equivalent schedule.

  Example)

T_1	T_2
R(A)	
A=A-50	
	R(B)
	B=B+50
R(B)	
B=B+50	
	R(A)
	A=A+10

T_1	T_2
	R(B)
	B=B+50
R(A)	
A=A-50	
R(B)	
B=B+50	
	R(A)
	A=A+10

The schedules that are conflict equivalent to a serial schedule are called conflict serializable schedule. Hence, if a non - serial schedule S can be transformed into a serial schedule S' by a series of non-conflicting instructions, the schedule S is said to be ***conflict serializable schedule***.

Procedure for Determining Conflict Serializability of a Schedule

- It can be determined using PRECEDENCE GRAPH method. A precedence graph consists of a pair G (V, E)
  V = set of vertices consisting of all the transactions participating in the schedule.

E = set of edges consists of all edges Tj for which one of the following conditions holds:
  - Ti executes write(Q) before Tj executes read(Q)
  - Ti executes read(Q) before Tj executes write(Q)
  - Ti executes write(Q) before Tj executes write(Q)

- If an edge Ti -> Tj exists in the precedence graph, then in any serial schedule S' equivalent to S, Ti must appear before Tj.
- If the precedence graph for S has *no cycle*, then schedule S *is conflict serializable*, else it is not.
- This cycle detection can be done by cycle detection algorithms, one of them based on depth first search takes O(n2) time.
- The *serialiazibility order* of transactions of equivalent serial schedule can be determined using topological order in a precedence graph.
- There can be more than 1 serialiazibility order possible for a given schedule since there can be multiple topological orderings for a graph possible.

A non-serial schedule which is conflict serializable is definitely *consistent* but it does not mean that non-serial schedule which is not conflict serializable is always inconsistent.

View Serializable Schedule

- If a schedule is conflict serializable then it will also be view serializable, hence we should check view serializability only if a schedule is not conflict serializable. "*All conflict serializable schedules are view serializable also*".

- If a schedule is not conflict serializable then it must have at least one blind write to be eligible for view serializable. i.e.
  - If schedule is not conflict serializable and it does not contain any blind write then it can never be view serializable.
  - If schedule is not conflict serializable and have atleast one blind write then it may or may not be view serializable.

- *Note*: A view serializable schedule may or may not be conflict serializable. Hence, conflict serializibility is a restrictive form of view serializibility.

- Before checking for view equivalency, we must check for conflict serializibility and blind write condition because checking view seriazibility is a *NP-Complete Problem*, hence it should be avoided due to heavy time complexity.

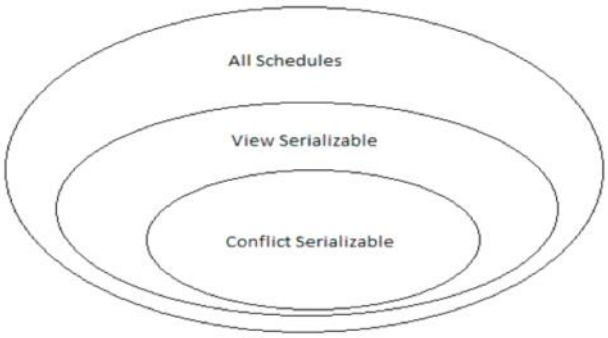
- To check view serializibility, tabulate all serial schedules possible (n!). Now, check one by one whether given schedule is view equivalent to any of the serial schedule. If the schedule is view equivalent to any of the serial schedule, then it is view serializable, otherwise if it is not view equivalent to any serial schedule, then it is not view serializable.

Two schedules S and S' are ***view equivalent***, if they satisfy following conditions:

- ***Initial reads*** *must be same for all the data items*:
  For each data item X, if transaction Ti reads X from the database initially in schedule S1, then in schedule S2 also, Ti must perform the initial read of X from the database.

- ***Intermediate write-read sequence*** *must be same:*
  If transaction Ti reads a data item that has been updated by the transaction Tj in schedule S1, then in schedule S2 also, transaction Ti must read the same data item that has been updated by the transaction Tj.

- ***Final writes*** *must be same for all the data items:*
  For each data item X, if X has been updated at last by transaction Ti in schedule S1, then in schedule S2 also, X must be updated at last by transaction Ti.



Note: If a non-serial schedule is neither conflict serializable, nor view serializable, even then also, it does not mean that it will always lead to inconsistency. It may or may not be consistent. But we can check for consistency only upto view serializibility.

Note:
Result Equivalent Schedules:

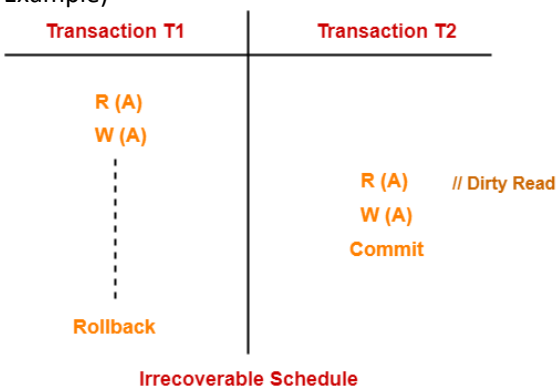
- If any two schedules generate the same result after their execution, then they are called as result equivalent schedules.
- This equivalence relation is considered of least significance.
- This is because some schedules might produce same results for some set of values and different results for some other set of values.

Recoverability

- ***Non-Recoverable Schedule:***
  - If in a schedule, a transaction performs a dirty read operation from an uncommitted transaction and commits before the transaction from which it has read the value then such a schedule is known as an Irrecoverable Schedule.

  - A schedule in which for each pair of transaction Ti and Tj such that if Tj reads a data item previously written by Ti, then the commit or abort operation of Ti appears before Tj. Such a schedule is called Non-Recoverable Schedule.

    Example)



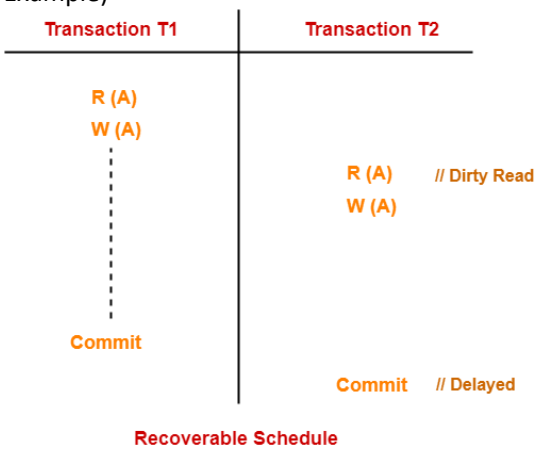
    T2 performs a dirty read operation. T2 commits before T1. T1 fails later and roll backs. The value that T2 read now stands to be incorrect. T2 can not recover since it has already committed.

- ***Recoverable Schedule:***
  - If in a schedule, a transaction performs a dirty read operation from an uncommitted transaction and its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is known as a Recoverable Schedule.

  - The commit operation of the transaction that performs the dirty read is delayed. This ensures that it still has a chance to recover if the uncommitted transaction fails later.

  - A schedule in which for each pair of transaction Ti and Tj such that if Tj reads a data item previously written by Ti, then the commit or abort of Ti must appear before Tj. Such a schedule is called Recoverable schedule.

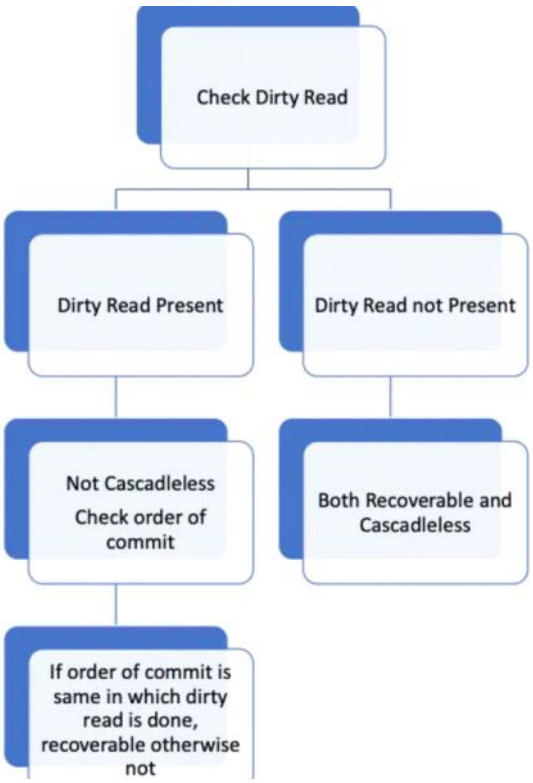
Example)



Recoverable Schedule

T2 performs a dirty read operation. The commit operation of T2 is delayed till T1 commits or roll backs. T1 commits later. T2 is now allowed to commit. In case, T1 would have failed, T2 has a chance to recover by rolling back.

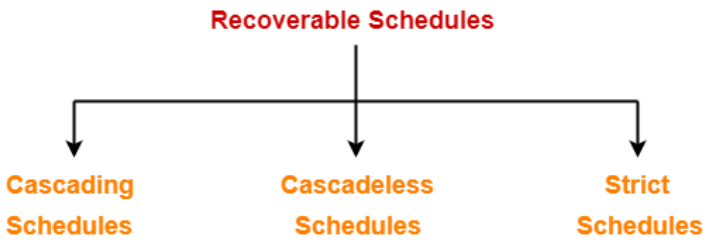
Procedure for Determining Recoverability of a Schedule



Check if there exists any dirty read operation. (Reading from an uncommitted transaction is called as a dirty read)

- If there does not exist any dirty read operation, then the schedule is surely recoverable (it is cascadeless also).
- If there exists any dirty read operation, then the schedule may or may not be recoverable. (Since it consists of dirty read, it is not cascadeless but may be cascading recoverable)
    - To check if it is cascading recoverable, check the order of commit
        - If it same in which dirty read is done, then it is recoverable (cascading but not cascadeless).
        - Else, schedule is irrecoverable.

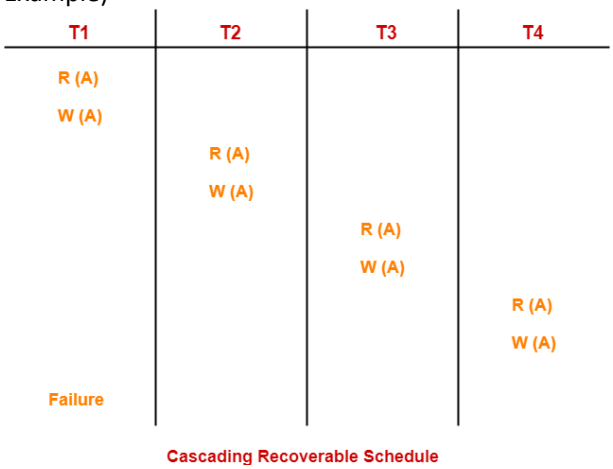
Types of Recoverable Schedules



1. **Cascading Schedule or Cascading Rollback or Cascading Abort**:

- It is a schedule, in which a single transaction failure leads to a series of transaction rollbacks.
- Even if the schedule is recoverable the, the commit of transaction may lead lot of transaction to rollback.
- Cascading rollback is undesirable, since it leads to undoing of a significant amount of work, leading too much wastage of CPU time.

Example)



Cascading Recoverable Schedule

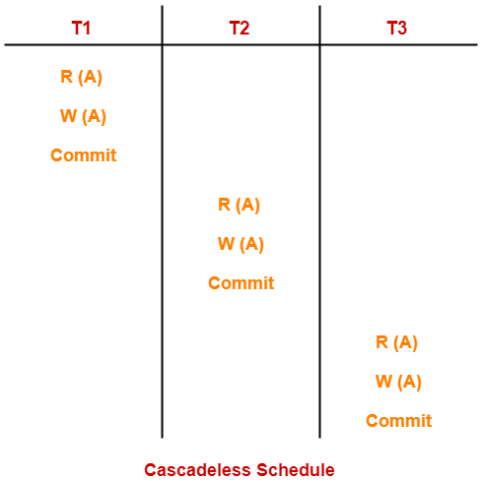
Transaction T2 depends on transaction T1. Transaction T3 depends on transaction T2. Transaction T4 depends on transaction T3. The failure of transaction T1 causes the transaction T2 to rollback. The rollback of transaction T2 causes the transaction T3 to rollback. The rollback of transaction T3 causes the transaction T4 to rollback. Such a rollback is called as a Cascading Rollback.

Note: If the transactions T2, T3 and T4 would have committed before the failure of transaction T1, then the schedule would have been irrecoverable.

2. ***Cascadeless Schedule***:

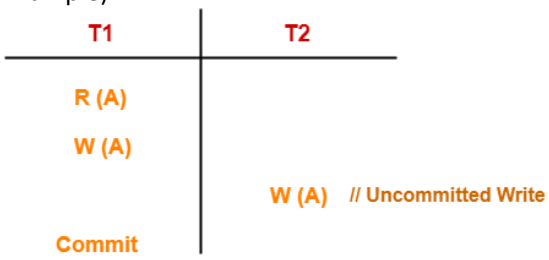
- To avoid cascading rollback, cascade less schedule are used.
- If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a Cascadeless Schedule.
- Cascadeless schedule allows only committed read operations. Therefore, it avoids cascading roll back and thus saves CPU time.
- A schedule in which for each pair of transactions Ti and Tj such that if Tj reads a data item previously written by Ti then the commit or abort of Ti must appear before read operation of Tj. Such a schedule is called cascade less schedule.

- Example)



Cascadeless Schedule

- Note: Cascadeless schedule may have uncommitted writes. (Only uncommitted read are not allowed).
  Example)

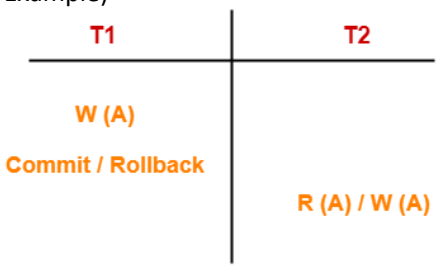


Cascadeless Schedule

3. ***Strict Schedule:***

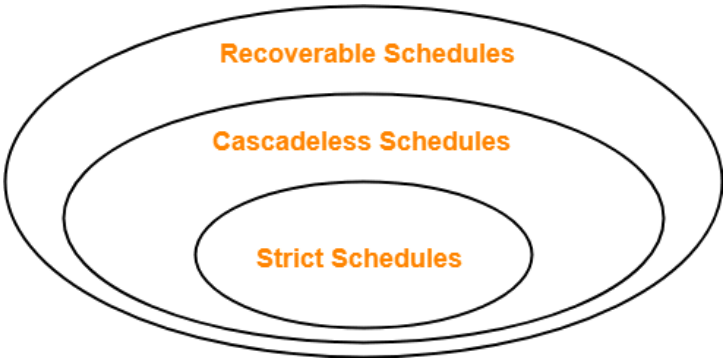
- If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a Strict Schedule.
- Strict schedule allows only committed read and write operations. Clearly, strict schedule implements more restrictions than cascadeless schedule.
- A schedule in which for each pair of transactions Ti and Tj, such that if Tj reads a data item previously written by Ti then the commit or abort of Ti must appear before read and write operation of Tj.

- Example)



Strict Schedule

Relation among Recoverable (Cascading), Cascadeless and Strict Schedule:



Strict schedules are more strict than cascadeless schedules. All strict schedules are cascadeless schedules. All cascadeless schedules are not strict schedules.

Concurrency Control

- Schedules must be conflict or view serializable, and recoverable, and preferably(optional) cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- To ensure conflict serializibility in a concurrency control protocol, we need transactions in schedule to not have conflictin g instructions, i.e. transactions accessing the same data item at the same time should be managed.

Goals of a Protocol

We desire the following properties from schedule generating protocols:

- Concurrency should be as high as possible, as this is our ultimate goal because of which we are making all the effort.
- The time taken by a transaction should also be less.
- Desirable Properties satisfied by the protocol
- Easy to understand and implement

Types of Protocols

1. **Time Stamping Based Method**

- Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp), in case of any conflict during the execution order can be decided using the time stamp.
  We have two timestamping, one for the transaction, and other for the data item.

Time stamp for transaction,

- With each transaction t_i, in the system, we associate a unique fixed timestamp, denoted by $TS(t_i)$.
- This timestamp is assigned by database system to a transaction at time transaction enters into the system.
- If a transaction has been assigned a timestamp $TS(t_i)$ and a new transaction t_j , enters into the system with a timestamp $TS(t_j)$, then always $TS(t_i) < TS(t_j)$.

- Time stamp of a transaction remain fixed throughout the execution. It is unique means no two transaction can have the same timestamp.

- The reason why we called time stamp not stamp, because for stamping we use the value of the system clock as stamp, advantage is, it will always be unique as time never repeats There is no requirement of refreshing and starting with fresh value.

- The time stamp of the transaction also determines the serializability order. Thus if TS(ti) <TS(tj), then the system must ens ure that the produced schedule is equivalent to a serial schedule in which transaction ti appears before transaction tj.

- Time stamp with data item, in order to assure such scheme, the protocol maintains for each data item Q two timestamp values:

  1. **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.
  2. **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully

- Time stamp with data item, in order to assure such scheme, the protocol maintains for each data item Q two timestamp values:

  1. **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.

  2. **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully.

- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

- Suppose a transaction Ti request a ***read(Q)***

  1. 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.

  2. If **TS(Ti)≥ W-timestamp(Q),** then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Suppose that transaction Ti issues ***write(Q).***

1. 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 write operation is rejected, and Ti is rolled back.

2. If **TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.

3. If **TS(Ti) ≥ R-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).

4. If **TS(Ti) ≥ W-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).

- If a transaction T_i is rolled back by the concurrency control scheme as a result of either a read or write operation, the system assigns it's a new timestamp and restarts it.

Properties

- Time stamp ordering protocol ensures conflict serializability. Because conflicting operations are processed in timestamp order, since all the arcs in the precedence graph are of the form thus, there will be no cycles in the precedence graph.
- As we know that view is liberal form conflict so view serializability also holds good
- As there is a possibility of dirty read, and no restriction on when to commit, so can be irrecoverable and may suffer from cascading rollback.
- At the time of request, here either we allow or we reject, so there is no idea of deadlock.
- If a schedule is not conflict serializable then it is not allowed by time stamp ordering scheme.
- But it is not necessary that all conflict serializable schedule generated by time stamping.

Conclusion

- It may cause starvation to occur, as if a sequence of conflicting transactions causes repeated restarting of the long transaction, and then there is a possibility of starvation of long transaction.
- It is relatively slow as before executing every instruction we have to check conditions before. Time stamping protocol ensure that the schedule designed through this protocol will always be conflict serializable.
- This protocol can also be used for determining the serializability order (order in which transaction must execute) among the transaction in advance.
- Further modifications are possible if we want to ensure recoverability and cascade lessness, using different approaches
    - By performing all writes together at the end of the transaction, i.e. while writers are in progress, no transaction is permitted to access any of data items that have been written.
    - By using a limited form of locking, where by read of uncommitted items are postponed until the transaction that uploaded the item commit.
    - Recoverability alone can be ensured by tracking uncommitted writes and allowing a transaction t_i to commit only after the commit of any transaction that wrote a value that t_i read.

THOMAS WRITE RULE

- Thomas write is an improvement in time stamping protocol, which makes some modification and may generate those protocols that are even view serializable, because it allows greater potential concurrency.

- It is a Modified version of the timestamp-ordering protocol in which Blind write operations may be ignored under certain circumstances.

- The protocol rules for read operations remain unchanged. while for write operation, there is slightly change in Thomas write rule than timestamp ordering protocol.

When T_i attempts to write data item Q,

- **if TS(T_i) < W-timestamp(Q),** then T_i is attempting to write an obsolete value of {Q}. Rather than rolling back T_i as the timestamp ordering protocol would have done, this {write} operation can be ignored.

- This modification is valid as the any transaction with **TS(T_i) < W-timestamp(Q),** the value written by this transaction will never be read by any other transaction performing Read(Q) ignoring such obsolete write operation is considerable.

- Thomas' Write Rule allows greater potential concurrency. Allows some view-serializable schedules that are not conflict serializable.

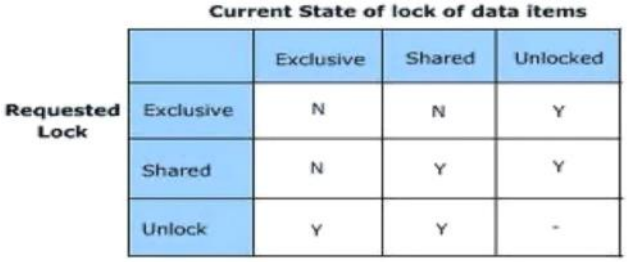
2. **Lock Based Method:**
   - We ask a transaction to first lock a data item before using it so that no different transactions can use a data item at the same time, removing any possibility of conflict.
   - To ensure isolation is to require that data items be accessed in a mutually exclusive manner i.e. while one transaction is accessing a data item, no other transaction can modify that data item. Locking is the most fundamental approach to ensure this.
   - Lock based protocols ensure this requirement. Idea is first obtain a lock on the desired data item then if lock is granted then perform the operation and then unlock it.

   - In general, we support two modes of locks to support better concurrency:
     - **Shared Mode**
       If transaction Ti has obtained a shared-mode lock (denoted by S) on any data item Q, then Ti can read, but cannot write Q, any other transaction can also acquire a shared mode lock on the same data item(this is the reason we called this shared mode).

     - **Exclusive Mode**
       If transaction Ti has obtained an exclusive-mode lock (denoted by X) on any data item Q, then Ti can both read and write Q, any other transaction cannot acquire either a shared or exclusive mode lock on the same data item. (this is the reason we called this exclusive mode)

   - *Lock — Compatibility Matrix*:
     Shared is compatible only with shared while exclusive is not compatible either with shared or exclusive. To access a data item, transaction Ti must first lock that item, if the data item is already locked by another transaction in an incompatible mode, or some other transaction is already waiting in non-compatible mode, then concurrency control manager will not

grant the lock until all incompatible locks held by other transactions have been released. The lock is then granted.



Current State of lock of data items

Requested Lock	Exclusive	Shared	Unlocked
Exclusive	N	N	Y
Shared	N	Y	Y
Unlock	Y	Y	-

- Lock based protocol do not ensure serializability as granting and releasing of lock do not follow any order and any transaction any time may go for lock and unlock.

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states, as there exists a possibility of dirty read. On the other hand, if we do not unlock a data item before requesting a lock on another data item, concurrency will be poor.

- We shall require that each transaction in the system follow a set of rules, called a locking protocol, indicating when a transaction may lock and unlock each of the data items for e.g. 2pl or graph based locking.

- Locking protocols restrict the number of possible schedules.

Types of Lock Based Method:

a. 2 Phase Locking

i. Basic 2 Phase Locking

- The protocol ensures that each transaction issue lock and unlock requests in two phases, note that each transaction will be 2 phased not schedule.
  - **Growing phase** - A transaction may obtain locks, but not release any locks.
  - **Shrinking phase** - A transaction may release locks, but may not obtain any new locks.

- Initially a transaction is in growing phase and acquires lock as needed and in between can perform operation reach to lock point and once a transaction releases a lock, it can issue no more lock requests i.e. it enters the shrinking phase.
- 2PL ensures conflict serializability, and the ordering of transaction over lock points is itself a serializability order of a schedule in 2PL. View serializability is also guaranteed.
- If a schedule is allowed in 2PL protocol then definitely it is always conflict serializable. But it is not necessary that if a schedule is conflict serializable then it will be generated by 2pl.
- Equivalent serial schedule is based on the order of lock points.
- Does not ensure freedom from deadlock. May cause non-recoverability. Cascading rollback may occur.

ii. Conservative 2 Phase Locking

- The idea is there is no growing phase transaction start directly from lock point, i.e. transaction must first acquire all the required locks then only it can start execution. If all the locks are not available then transaction must release the acquired locks and must wait.
- Shrinking phase will work as usual, and transaction can unlock any data item anytime.
- We must have a knowledge in future to understand what is data required so that we can use it .

iii. Rigorous 2 Phase Locking

- Requires that all locks be held until the transaction commits.
- This protocol requires that locking be two phase and also all the locks taken be held by transaction until that transaction commit.
- Hence there is no shrinking phase in the system.

iv. Strict 2 Phase Locking

- All exclusive-mode locks taken by a transaction be held until that transaction commits.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- This protocol requires that locking be two phase and also that exclusive mode locks taken by transaction be held until that transaction commits.
- So it is simplified form of rigorous 2pl

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL	YES	YES	YES	YES	NO
Strict 2PL	"	"	"	"	"

b. Graph Based Protocol

- If we wish to develop lock based protocol that are not based on 2PL, we need additional info that how each transaction will access the data.
- These are various model that can give additional information each differing in the amount of info provided.
- One idea is to have prior knowledge about the order in which the database items will be accessed.
- We impose ***partial ordering*** -> on set all data items D = {d1, d2, …. dn}, if di -> dj, then any transaction accessing both di and dj must access di before dj.
- Partial Ordering may be because logical or physical organisation (due to data structure like linked list used) or only because of concurrency control.
- After Partial ordering, set of all data items D will now be viewed as directed acyclic graph called database graph.
- For sake of simplicity, we follow two restrictions:
  - Will study graph that are rooted trees.
  - Will only use exclusive mode locks.

Tree Based Protocol

- Each transaction Ti can lock a data item Q with following rules:
  - First lock by Ti may be on any data item
  - Subsequently a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti.
  - Data item may be unlocked at any time.
  - Data item Q that has been locked and unlocked by Ti cannot be subsequently relocked by Ti.

Properties of Tree Protocol

- All schedules that are legal under the tree protocol are Conflict serializable and View serializable.
- Tree Protocol ensure freedom from deadlock.
- Tree protocol do not ensure recoverablity and cascadelessness.

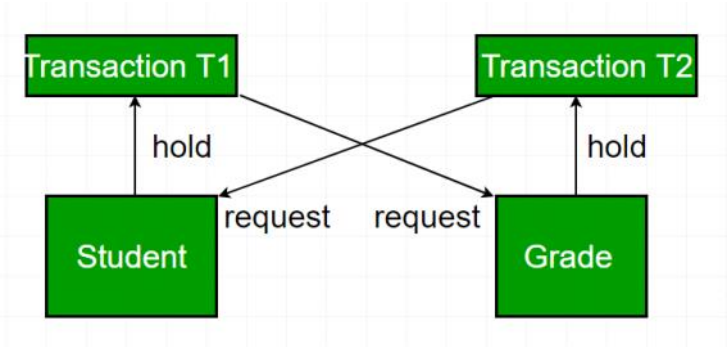
- Early unlocking is possible which leads shorter waiting time, thus increasing concurrency.
- A transaction may have to lock data items that it does not access leading to overhead, increasing waiting time and decreasing concurrency.
- A transaction must know exactly what data item are to be accessed.
- Every schedule by Tree Protocol is conflict serializable but every conflict serializable schedule cannot be derived from tree protocol.
- Recoverability and cascadelessness can be provided by not unlocking before commit.

3. **Multiple Granularity Protocol:**
   Refer Article: https://www.geeksforgeeks.org/multiple-granularity-locking-in-dbms/

4. **Validation Based Protocol:**
   Refer Article: https://www.geeksforgeeks.org/validation-based-protocol-in-dbms/

Deadlock

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. None of the transactions can make progess in such a situation.



- The system has to *rollback some of the transactions* involved in the deadlock. Rollback of a transaction may be *partial*, i.e. it may be rolled back to the point where it obtained a lock whose release resolves a deadlock.

- Principal methods for dealing with deadlock are:

a. Deadlock Prevention:

It ensures that the sytem will never enter a deadlock state. Prevention is commonly used if the probability of the system would enter a deadlock state is relatively high, otherwise detection & recovery are more efficient.

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

Some schemes that use transaction timestamps for deadlock prevention are:

- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

- a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

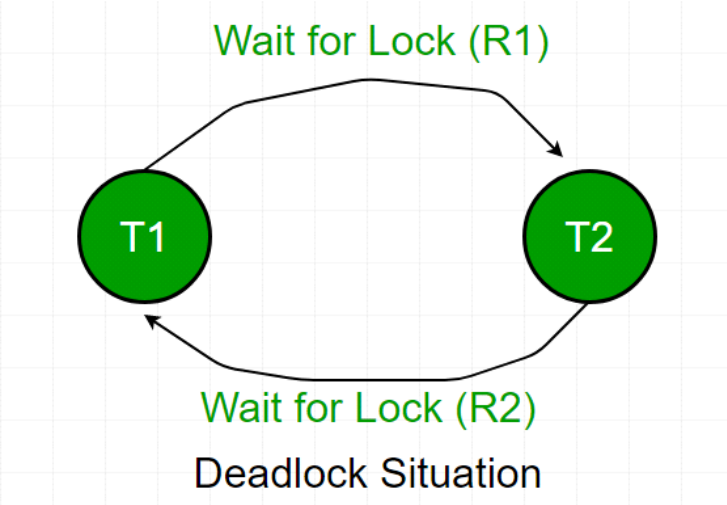
b. Deadlock Detection & Deadlock Recovery:

- We can allow the system to enter a deadlocked state, and then try to recover by first detecting it and then applying suitable recovery scheme.
- Detection & Recovery Scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

- For detection & recovery scheme, the system must
  - □ Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
  - □ Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
  - □ Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a ***wait-for graph***. 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 in the system.



- Each element in the set E of edges is an ordered pair Ti → Tj. If Ti → Tj is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.
- When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.
- If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken.
- In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

Deadlock Recovery

- When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock.
- Three actions need to be taken:

1) **Selection of a victim**:
   - □ Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back

to break the deadlock. We should roll back those transactions that will incur the minimum cost.

- Many factors may determine the cost of a rollback, including:
  - How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
  - How many data items the transaction has used.
  - How many more data items the transaction needs for it to complete.
  - How many transactions will be involved in the rollback.

2) Rollback:

- Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- The simplest solution is a ***total rollback***: Abort the transaction and then restart it.
- However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such ***partial rollback*** requires the system to maintain additional information about the state of all the running transactions.
- Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.
- The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point.
- The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback.

3) Starvation:

- In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim.
- As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that a transaction can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.

Solutions to avoid Starvation:

- Increasing Priority –

Starvation occurs when a transaction has to wait for an indefinite time, In this situation, we can increase the priority of that particular transaction/s. But the drawback with this solution is that it may happen that the other transaction may have to wait longer until the highest priority transaction comes and proceeds.

- Modification in Victim Selection algorithm –

If a transaction has been a victim of repeated selections, then the algorithm can be modified by lowering its priority over other transactions.

- First Come First Serve approach –

A fair scheduling approach i.e FCFS can be adopted, In which the transaction can acquire a lock on an item in the order, in which the requested the lock.

- Wait die and wound wait scheme –

These are the schemes that use the timestamp ordering mechanism of transaction.

c. Deadlock Avoidance

- When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database.
- Deadlock avoidance method is suitable for smaller databases whereas deadlock prevention method is suitable for larger databases.

Recovery System

Refer Articles:

1. https://www.geeksforgeeks.org/why-recovery-is-needed-in-dbms/
2. https://www.geeksforgeeks.org/database-recovery-techniques-in-dbms/
3. https://www.geeksforgeeks.org/log-based-recovery-in-dbms/
4. https://www.geeksforgeeks.org/difference-between-deferred-update-and-immediate-update/

Types of Failures

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- For the system to recover from failures, we need recovery algorithms to ensure database consistency and transaction atomaticity despite failures.
- Recovery algorithms have two parts:
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Storage Structure Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    i. Write the information onto the first physical block.
    ii. When the first write successfully completes, write the same information onto the second physical block.
    iii. The output is completed only after the second write successfully completes.
- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item *X* is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(*X*) assigns the value of data item *X* to the local variable x_i.

- **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
- **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Recovery & Atomacity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log Based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.
Log-based recovery works as follows −

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

`<Tn, Start>`

- When the transaction modifies an item X, it write logs as follows −

`<Tn, X, V1, V2>`

It reads Tn has changed the value of X, from V1 to V2.

- When the transaction finishes, it logs −

`<Tn, commit>`

The database can be modified using two approaches −

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <Tn, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.