Database Management System (DBMS)

Introduction to database Management System

1. All about DBMS

A Database Management System (DBMS) is a software system that is designed to manage and organize data in a structured manner. It allows users to create, modify, and query a database, as well as manage the security and access controls for that database.
DBMS provides an environment to store and retrieve the data in coinvent and efficient manner.

Key Features of DBMS

- **Data modeling:** A DBMS provides tools for creating and modifying data models, which define the structure and relationships of the data in a database.
- **Data storage and retrieval:** A DBMS is responsible for storing and retrieving data from the database and can provide various methods for searching and querying the data.
- **Concurrency control:** A DBMS provides mechanisms for controlling concurrent access to the database, to ensure that multiple users can access the data without conflicting with each other.
- **Data integrity and security:** A DBMS provides tools for enforcing data integrity and security constraints, such as constraints on the values of data and access controls that restrict who can access the data.
- **Backup and recovery:** A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.
- **DBMS can be classified into two types:** Relational Database Management System (RDBMS) and Non-Relational Database Management System (NoSQL or Non-SQL)
- **RDBMS:** Data is organized in the form of tables and each table has a set of rows and columns. The data are related to each other through primary and foreign keys.
- **NoSQL:** Data is organized in the form of key-value pairs, documents, graphs, or column-based. These are designed to handle large-scale, high-performance scenarios.

A database is a collection of interrelated data which helps in the efficient retrieval, insertion, and deletion of data from the database and organizes the data in the form of tables, views, schemas, reports, etc. For Example, a university database organizes the data about students, faculty, admin staff, etc. which helps in the efficient retrieval, insertion, and deletion of data from it.

Database Languages

1) Data Definition Language
2) Data Manipulation Language
3) Data Control Language
4) Transactional Control Language

Data Definition Language

DDL is the short name for Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.
0 seconds of 21 seconds Volume 0%

- **CREATE:** to create a database and its objects like (table, index, views, store procedure, function, and triggers)
- **ALTER:** alters the structure of the existing database
- **DROP:** delete objects from the database
- **TRUNCATE:** remove all records from a table, including all spaces allocated for the records are removed
- **COMMENT:** add comments to the data dictionary
- **RENAME:** rename an object

Data Manipulation Language

DML is the short name for Data Manipulation Language which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete, and update data in a database.

- **SELECT:** retrieve data from a database
- **INSERT:** insert data into a table
- **UPDATE:** updates existing data within a table
- **DELETE:** Delete all records from a database table
- **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

Data Control Language

DCL is short for Data Control Language which acts as an access specifier to the database. (Basically, to grant and revoke permissions to users in the database

- **GRANT:** grant permissions to the user for running DML (SELECT, INSERT, DELETE…) commands on the table
- **REVOKE:** revoke permissions to the user for running DML (SELECT, INSERT, DELETE…) command on the specified table

Transactional Control Language

TCL is short for Transactional Control Language which acts as a manager for all types of transactional data and all transactions. Some of the commands of TCL are.

- **Roll Back:** Used to cancel or Undo changes made in the database.
- **Commit:** It is used to apply or save changes in the database
- **Save Point:** It is used to save the data on the temporary basis in the database.

Data retrieval language:

DRL is short for Data Retrieval Language which is used for retrieval of data. It can also be said as DML.

- **SELECT:** Used for extracting the required data.

Applications of DBMS:

- **Enterprise Information:** Sales, accounting, human resources, Manufacturing, online retailers.
- **Banking and Finance Sector:** Banks maintaining the customer details, accounts, loans, banking transactions, credit card transactions. Finance: Storing the information about sales and holdings, purchasing of financial stocks and bonds.
- **University:** Maintaining the information about student course enrolled information, student grades, staff roles.
- **Airlines:** Reservations and schedules.
- **Telecommunications:** Prepaid, postpaid bills maintenance.

Paradigm Shift from File System to DBMS

File System manages data using files on a hard disk. Users are allowed to create, delete, and update the files according to their requirements. Let us consider the example of file-based University Management System. Data of students is available to their respective Departments, Academics Section, Result Section, Accounts Section, Hostel Office, etc. Some of the data is common for all sections like Roll No, Name, Father Name, Address, and Phone number of students but some data is available to a particular section only like Hostel allotment number which is a part of the hostel office. Let us discuss the issues with this system:

- **Redundancy of data:** Data is said to be redundant if the same data is copied at many places. If a student wants to change their Phone number, he or she has to get it updated in various sections. Similarly, old records must be deleted from all sections representing that student.
- **Inconsistency of Data:** Data is said to be inconsistent if multiple copies of the same data do not match each other. If the Phone number is different in Accounts Section and Academics Section, it will be

inconsistent. Inconsistency may be because of typing errors or not updating all copies of the same data.

- **Difficult Data Access:** A user should know the exact location of the file to access data, so the process is very cumbersome and tedious. If the user wants to search the student hostel allotment number of a student from 10000 unsorted students' records, how difficult it can be.
- **Unauthorized Access:** File Systems may lead to unauthorized access to data. If a student gets access to a file having his marks, he can change it in an unauthorized way.
- **No Concurrent Access:** The access of the same data by multiple users at the same time is known as concurrency. The file system does not allow concurrency as data can be accessed by only one user at a time.
- **No Backup and Recovery:** The file system does not incorporate any backup and recovery of data if a file is lost or corrupted.

Advantages of DBMS

- **Data organization:** A DBMS allows for the organization and storage of data in a structured manner, making it easy to retrieve and query the data as needed.
- **Data integrity:** A DBMS provides mechanisms for enforcing data integrity constraints, such as constraints on the values of data and access controls that restrict who can access the data.
- **Concurrent access:** A DBMS provides mechanisms for controlling concurrent access to the database, to ensure that multiple users can access the data without conflicting with each other.
- **Data security:** A DBMS provides tools for managing the security of the data, such as controlling access to the data and encrypting sensitive data.
- **Backup and recovery:** A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.
- **Data sharing:** A DBMS allows multiple users to access and share the same data, which can be useful in a collaborative work environment.

Disadvantages of DBMS

- **Complexity:** DBMS can be complex to set up and maintain, requiring specialized knowledge and skills.
- **Performance overhead:** The use of a DBMS can add overhead to the performance of an application, especially in cases where high levels of concurrency are required.
- **Scalability:** The use of a DBMS can limit the scalability of an application since it requires the use of locking and other synchronization mechanisms to ensure data consistency.

- **Cost:** The cost of purchasing, maintaining, and upgrading a DBMS can be high, especially for large or complex systems.
- **Limited Use Cases:** Not all use cases are suitable for a DBMS, some solutions do not need high reliability, consistency or security and may be better served by other types of data storage.

These are the main reasons which made a shift from file system to DBMS.
Also, see.
A Database Management System (DBMS) is a software system that allows users to create, maintain, and manage databases. It is a collection of programs that enables users to access and manipulate data in a database. A DBMS is used to store, retrieve, and manipulate data in a way that provides security, privacy, and reliability.

Several Types of DBMS

- **Relational DBMS (RDBMS):** An RDBMS stores data in tables with rows and columns and uses SQL (Structured Query Language) to manipulate the data.
- **Object-Oriented DBMS (OODBMS):** An OODBMS stores data as objects, which can be manipulated using object-oriented programming languages.
- **NoSQL DBMS:** A NoSQL DBMS stores data in non-relational data structures, such as key-value pairs, document-based models, or graph models.

2. Database Architecture

A Database stores a lot of critical information to access data quickly and securely. Hence it is important to select the correct architecture for efficient data management. DBMS Architecture helps users to get their requests done while connecting to the database. We choose database architecture depending on several factors like the size of the database, number of users, and relationships between the users. There are two types of database models that we generally use, logical model and physical model. Several types of architecture are there in the database which we will deal with in the next section.

Types of DBMS Architecture

There are several types of DBMS Architecture that we use according to the usage requirements. Types of DBMS Architecture are discussed here.

- 1-Tier Architecture
- 2-Tier Architecture
- 3-Tier Architecture

➢ 1-Tier Architecture

In 1-Tier Architecture the database is directly available to the user, the user can directly sit on the DBMS and use it that is, the client, server, and Database are all present on the same machine. For Example: to learn SQL we set up an SQL server and the database on the local system. This enables us to directly interact with the relational database and execute operations. The industry won't use this architecture they logically go for 2-tier and 3-tier Architecture.

Advantages of 1-Tier Architecture

Below mentioned are the advantages of 1-Tier Architecture.

- **Simple Architecture:** 1-Tier Architecture is the simplest architecture to set up, as only a single machine is required to maintain it.
- **Cost-Effective:** No additional hardware is required for implementing 1-Tier Architecture, which makes it cost-effective.
- **Easy to Implement:** 1-Tier Architecture can be easily deployed, and hence it is mostly used in small projects.

➢ 2-Tier Architecture

The 2-tier architecture is like a basic client-server model. The application at the client end directly communicates with the database on the server side. APIs like ODBC and JDBC 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 to communicate with the DBMS.

An advantage of this type is that maintenance and understanding are easier, and compatible with existing systems. However, this model gives poor performance when there are many users.

Figure 1: 2-Tire Architecture

Advantages of 2-Tier Architecture

- **Easy to Access:** 2-Tier Architecture makes easy access to the database, which makes fast retrieval.
- **Scalable:** We can scale the database easily, by adding clients or upgrading hardware.
- **Low Cost:** 2-Tier Architecture is cheaper than 3-Tier Architecture and Multi-Tier Architecture.

- **Easy Deployment:** 2-Tier Architecture is easier to deploy than 3-Tier Architecture.
- **Simple:** 2-Tier Architecture is easily understandable as well as simple because of only two components.

3-Tier Architecture

In 3-Tier Architecture, there is another layer between the client and the server. 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 the exchange of partially processed data between the server and the client. This type of architecture is used in the case of large web applications.

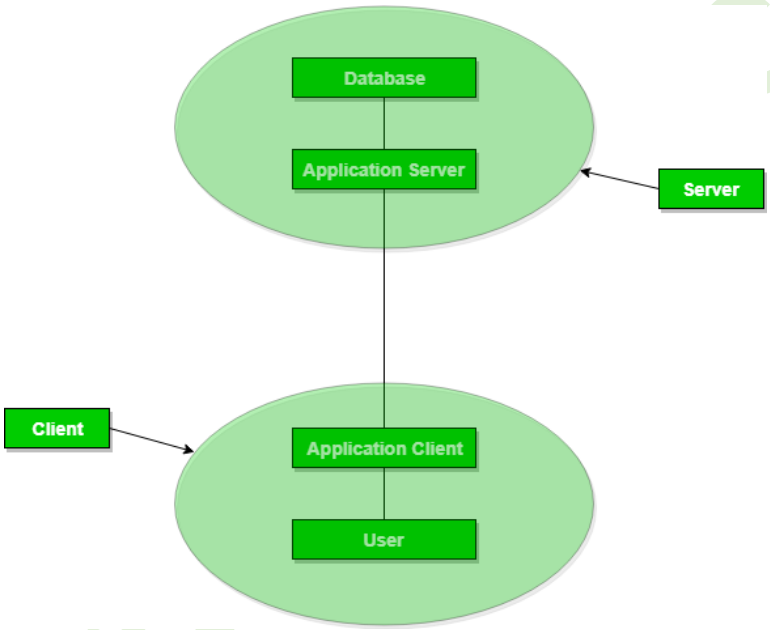


Figure 1: 3-Tire Architecture

Advantages of 3-Tier Architecture

- **Enhanced scalability:** Scalability is enhanced due to the distributed deployment of application servers. Now, individual connections need not be made between the client and server.
- **Data Integrity:** 3-Tier Architecture maintains Data Integrity. Since there is a middle layer between the client and the server, data corruption can be avoided/removed.
- **Security:** 3-Tier Architecture Improves Security. This type of model prevents direct interaction of the client with the server thereby reducing access to unauthorized data.

Disadvantages of 3-Tier Architecture

- **More Complex:** 3-Tier Architecture is more complex in comparison to 2-Tier Architecture. Communication Points are also doubled in 3-Tier Architecture.
- **Difficult to Interact:** It becomes difficult for this sort of interaction to take place due to the presence of middle layers.

3. Need for DBMS

A **D**ata **B**ase **M**anagement **S**ystem is a system software for easy, efficient and reliable data processing and management. It can be used for:

- Creation of a database.
- Retrieval of information from the database.
- Updating the database.
- Managing a database.
    - **Multiple User Interface**
    - **Data scalability, expandability, and flexibility**: We can change schema of the database, all schema will be updated according to it.
    - Overall, the time for developing an application is reduced.
    - **Security:** Simplifies data storage as it is possible to assign security permissions allowing restricted access to data.

Data organization: DBMS allow users to organize large amounts of data in a structured and systematic way. Data is organized into tables, fields, and records, making it easy to manage, store, and retrieve information.
Data scalability: DBMS are designed to handle large amounts of data and are scalable to meet the growing needs of organizations. As organizations grow, DBMS can scale up to handle increasing amounts of data and user traffic.

- **Data Organization and Management:**

One of the primary needs for a DBMS is data organization and management. DBMSs allow data to be stored in a structured manner, which helps in easier retrieval and analysis. A well-designed database schema enables faster access to information, reducing the time required to find relevant data. A DBMS also provides features like indexing and searching, which make it easier to locate specific data within the database. This allows organizations to manage their data more efficiently and effectively.

- **Data Security and Privacy:**

DBMSs provide a robust security framework that ensures the confidentiality, integrity, and availability of data. They offer authentication and authorization features that control access to the database. DBMSs also provide encryption

capabilities to protect sensitive data from unauthorized access. Moreover, DBMSs comply with various data privacy regulations such as the GDPR, HIPAA, and CCPA, ensuring that organizations can store and manage their data in compliance with legal requirements.

- **Data Integrity and Consistency:**

Data integrity and consistency are crucial for any database. DBMSs provide mechanisms that ensure the accuracy and consistency of data. These mechanisms include constraints, triggers, and stored procedures that enforce data integrity rules. DBMSs also provide features like transactions that ensure that data changes are atomic, consistent, isolated, and durable (ACID).

- **Concurrent Data Access:**

A DBMS provides a concurrent access mechanism that allows multiple users to access the same data simultaneously. This is especially important for organizations that require real-time data access. DBMSs use locking mechanisms to ensure that multiple users can access the same data without causing conflicts or data corruption.

- **Data Analysis and Reporting:**

DBMSs provide tools that enable data analysis and reporting. These tools allow organizations to extract useful insights from their data, enabling better decision-making. DBMSs support various data analysis techniques such as OLAP, data mining, and machine learning. Moreover, DBMSs provide features like data visualization and reporting, which enable organizations to present their data in a visually appealing and understandable way.

- **Scalability and Flexibility:**

DBMSs provide scalability and flexibility, enabling organizations to handle increasing amounts of data. DBMSs can be scaled horizontally by adding more servers or vertically by increasing the capacity of existing servers. This makes it easier for organizations to handle large amounts of data without compromising performance. Moreover, DBMSs provide flexibility in terms of data modeling, enabling organizations to adapt their databases to changing business requirements.

- **Cost-Effectiveness:**

DBMSs are cost-effective compared to traditional file-based systems. They reduce storage costs by eliminating redundancy and optimizing data storage. They also reduce development costs by providing tools for database design, maintenance, and administration. Moreover, DBMSs reduce operational costs by automating routine tasks and providing self-tuning capabilities.

4. Challenges of Database Security in DBMS

The vast increase in volume and speed of threats to databases and many information assets, research efforts need to be consider to the following issues such as data quality, intellectual property rights, and database survivability.

Let us discuss them one by one.

i) Data quality –

- The database community basically needs techniques and some organizational solutions to assess and attest the quality of data. These techniques may include the simple mechanism such as quality stamps that are posted on different websites. We also need techniques that will provide us more effective integrity semantics verification tools for assessment of data quality, based on many techniques such as record linkage.
- We also need application-level recovery techniques to automatically repair the incorrect data.
- The ETL that is extracted transform and load tools widely used for loading the data in the data warehouse are presently grappling with these issues.

ii) Intellectual property rights –

As the use of Internet and intranet is increasing day by day, legal and informational aspects of data are becoming major concerns for many organizations. To address this concerns watermark technique are used which will help to protect content from unauthorized duplication and distribution by giving the provable power to the ownership of the content. Traditionally they are dependent upon the availability of a large domain within which the objects can be altered while retaining its essential or important properties. However, research is needed to access the robustness of many such techniques and the study and investigate many different approaches or methods that aimed to prevent intellectual property rights violation.

iii) Database survivability –

Database systems need to operate and continued their functions even with the reduced capabilities, despite disruptive events such as information warfare attacks A DBMS in addition to making every effort to prevent an attack and detecting one in the event of the occurrence should be able to do the following:

- **Confident:** We should take immediate action to eliminate the attacker's access to the system and to isolate or contain the problem to prevent further spread.
- **Damage assessment:** Determine the extent of the problem, including failed function and corrupted data.
- **Recover:** Recover corrupted or lost data and repair or reinstall failed function to reestablish a normal level of operation.

- **Reconfiguration:** Reconfigure to allow the operation to continue in a degraded mode while recovery proceeds.
- **Fault treatment:** To the extent possible, identify the weakness exploited in the attack and takes steps to prevent a recurrence.

Database security

It is an essential aspect of database management systems (DBMS) as it involves protecting the confidentiality, integrity, and availability of the data stored in the database. The challenges of database security in DBMS include:

- **Authentication and Authorization:** One of the biggest challenges of database security is ensuring that only authorized users can access the database. The DBMS must authenticate users and grant them appropriate access rights based on their roles and responsibilities.
- **Encryption:** Data encryption is an effective way to protect sensitive data in transit and at rest. However, it can also be a challenge to implement and manage encryption keys and ensure that encrypted data is not compromised.
- **Access Control:** Access control involves regulating the access to data within the database. It can be challenging to implement access control mechanisms that allow authorized users to access the data they need while preventing unauthorized users from accessing it.
- **Auditing and Logging:** DBMS must maintain an audit trail of all activities in the database. This includes monitoring who accesses the database, what data is accessed, and when it is accessed. This can be a challenge to implement and manage, especially in large databases.
- **Database Design:** The design of the database can also impact security. A poorly designed database can lead to security vulnerabilities, such as SQL injection attacks, which can compromise the confidentiality, integrity, and availability of data.
- **Malicious attacks:** Cyberattacks such as hacking, malware, and phishing pose a significant threat to the security of databases. DBMS must have robust security measures in place to prevent and detect such attacks.
- **Physical Security:** Physical security of the database is also important, as unauthorized physical access to the server can lead to data breaches.

Features that are used to enhance database security:

- **Backup and Recovery:** DBMS systems include backup and recovery features that ensure that data can be restored in the event of a system failure or security breach. Backups can be created at regular intervals and stored securely to prevent unauthorized access.
- **Access Controls:** Access controls can be used to restrict access to certain parts of the database based on user roles or permissions. For example, a

DBMS can enforce rules such as not allowing a user to drop tables or granting read-only access to some users.

- **Database Auditing and Testing Tools:** Database auditing and testing tools allow security personnel to monitor and test the security of the database. This helps in identifying security gaps and weaknesses in the system.
- **Data Masking:** DBMS systems support data masking features which are used to protect sensitive data by obscuring it from view. This is especially useful in cases where sensitive data needs to be accessed by third-party vendors or contractors.

5. Advantage of DBMS over File system

File System: A File Management system is a DBMS that allows access to single files or tables at a time. In a File System, data is directly stored in a set of files. It contains flat files that have no relation to other files (when only one table is stored in a single file, then this file is known as a flat file).
DBMS: A Database Management System (DBMS) is application software that allows users to efficiently define, create, maintain, and share databases. Defining a database involves specifying the data types, structures and constraints of the data to be stored in the database. Creating a database involves storing the data on some storage medium that is controlled by DBMS. Maintaining a database involves updating the database whenever required to evolve and reflect changes in the Mini world and also generating reports for each change. Sharing a database involves allowing multiple users to access the database. DBMS also serves as an interface between the database and end users or application programs. It provides control access to the data and ensures that data is consistent and correct by defining rules on them.
An application program accesses the database by sending queries or requests for data to the DBMS. A query causes some data to be retrieved from the database.

Advantages of DBMS over File system are:

- **Data redundancy and inconsistency:** Redundancy is the concept of repetition of data i.e. each data may have more than a single copy. The file system cannot control the redundancy of data as each user defines and maintains the needed files for a specific application to run. There may be a possibility that two users are maintaining the data of the same file for different applications. Hence changes made by one user do not reflect in files used by second users, which leads to inconsistency of data. Whereas DBMS controls redundancy by maintaining a single repository of data that is defined once and is accessed by many users. As there is no or less redundancy, data remains consistent.

- **Data sharing:** The file system does not allow sharing of data or sharing is too complex. Whereas in DBMS, data can be shared easily due to a centralized system.
- **Data concurrency:** Concurrent access to data means more than one user is accessing the same data at the same time. Anomalies occur when changes made by one user get lost because of changes made by another user. The file system does not provide any procedure to stop anomalies. Whereas DBMS provides a locking system to stop anomalies to occur.
- **Data searching:** For every search operation performed on the file system, a different application program has to be written. While DBMS provides inbuilt searching operations. The user only has to write a small query to retrieve data from the database.
- **Data integrity:** There may be cases when some constraints need to be applied to the data before inserting it into the database. The file system does not provide any procedure to check these constraints automatically. Whereas DBMS maintains data integrity by enforcing user-defined constraints on data by itself.
- **System crashing:** In some cases, systems might have crashed due to various reasons. It is a bane in the case of file systems because once the system crashes, there will be no recovery of the data that's been lost. A DBMS will have the recovery manager which retrieves the data making it another advantage over file systems.
- **Data security:** A file system provides a password mechanism to protect the database but how long can the password be protected? No one can guarantee that. This doesn't happen in the case of DBMS. DBMS has specialized features that help provide shielding to its data.
- **Backup:** It creates a backup subsystem to restore the data if required.
- **Interfaces**: It provides different multiple user interfaces like graphical user interface and application program interface.
- **Easy Maintenance**: It is easily maintainable due to its centralized nature.

6. Data Abstraction and Data Independence

Database systems comprise complex data structures. To make the system efficient in terms of retrieval of data, and reduce complexity in terms of usability of users, developers use abstraction i.e. hide irrelevant details from the users. This approach simplifies database design.

Level of Abstraction in a DBMS
There are mainly 3 levels of data abstraction:

- Physical or Internal Level
- Logical or Conceptual Level
- View or External Level

Physical or Internal Level

This is the lowest level of data abstraction. It tells us how the data is stored in memory. Access methods like sequential or random access and file organization methods like B+ trees and hashing are used for the same. Usability, size of memory, and the number of times the records are factors that we need to know while designing the database.

Suppose we need to store the details of an employee. Blocks of storage and the amount of memory used for these purposes are kept hidden from the user.

Logical or Conceptual Level

This level comprises the information that is stored in the database in the form of tables. It also stores the relationship among the data entities in relatively simple structures. At this level, the information available to the user at the view level is unknown.

We can store the various attributes of an employee and relationships, e.g. with the manager can also be stored.

View or External Level

This is the highest level of abstraction. Only a part of the actual database is viewed by the users. This level exists to ease the accessibility of the database by an individual user. Users view data in the form of rows and columns. Tables and relations are used to store data. Multiple views of the same database may exist. Users can just view the data and interact with the database, storage and implementation details are hidden from them.

Example: In case of storing customer data,

- **Physical level** – it will contain block of storages (bytes, GB, TB, etc)
- **Logical level –** it will contain the fields and the attributes of data.
- **View level** – it works with CLI or GUI access of database

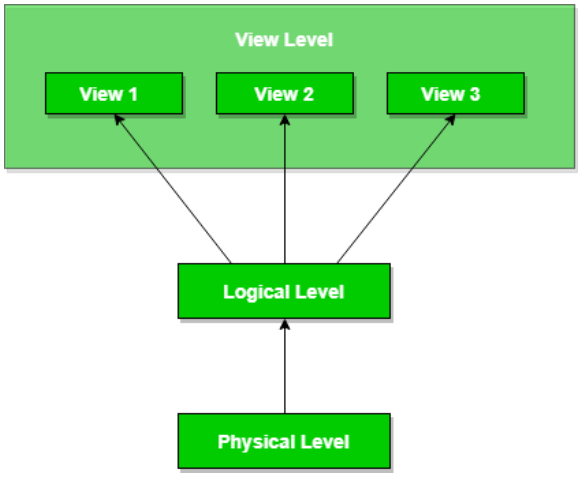


Figure 3: Data Abstraction

The main purpose of data abstraction is to achieve data independence in order to save the time and cost required when the database is modified or altered.

Data Independence

It is mainly defined as a property of DBMS that helps you to change the database schema at one level of a system without requiring to change the schema at the next level. it helps to keep the data separated from all programs that makes use of it.

We have namely two levels of data independence arising from these levels of abstraction:

- Physical level data independence
- Logical level data independence

Physical Level Data Independence

It refers to the characteristic of being able to modify the physical schema without any alterations to the conceptual or logical schema, done for optimization purposes, e.g., the Conceptual structure of the database would not be affected by any change in storage size of the database system server. Changing from sequential to random access files is one such example. These alterations or modifications to the physical structure may include:

- Utilizing new storage devices.
- Modifying data structures used for storage.
- Altering indexes or using alternative file organization techniques etc.

Logical Level Data Independence

It refers characteristic of being able to modify the logical schema without affecting the external schema or application program. The user view of the data would not be affected by any changes to the conceptual view of the data. These changes may include insertion or deletion of attributes, altering table structures entities or relationships to the logical schema, etc.

7. Introduction to ER-Model

The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.

The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.

Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

- **Rectangles:** Rectangles represent Entities in the ER Model.
- **Ellipses:** Ellipses represent Attributes in the ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

Components of ER Diagram

ER Model consists of Entities, Attributes, and Relationships among Entities in a Database System.

i) Entity

An Entity may be an object with a physical existence – a particular person, car, house, or employee – or it may be an object with a conceptual existence – a company, a job, or a university course.

Entity Set: An Entity is an object of Entity Type, and a set of all entities is called an entity set. For Example, E1 is an entity having Entity Type Student and the set of all students is called Entity Set.

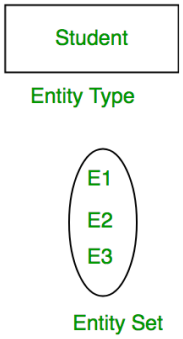


Figure 4: Entity Set

a. Strong Entity

A Strong Entity is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

b. Weak Entity

An Entity type has a key attribute that uniquely identifies each entity in the entity set. But some entity type exists for which key attributes cannot be defined. These are called Weak Entity types.

For Example, A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents do not have existed without the employee. So Dependent will be a **Weak Entity Type** and Employee will be Identifying Entity type for Dependent, which means it is **Strong Entity Type**.

A weak entity type is represented by a Double Rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.

1) Attributes

Attributes are the properties that define the entity type. For example, Roll_No, Name, DOB, Age, Address, and Mobile_No are the attributes that define entity type Student. In ER diagram, the attribute is represented by an oval.

Figure 5: Attribute

a) Key Attribute

The attribute which **uniquely identifies each entity** in the entity set is called the key attribute. For example, Roll_No will be unique for each student. In ER diagram, the key attribute is represented by an oval with underlying lines.

Figure 6: Key Attribute

b) Composite Attribute

An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.

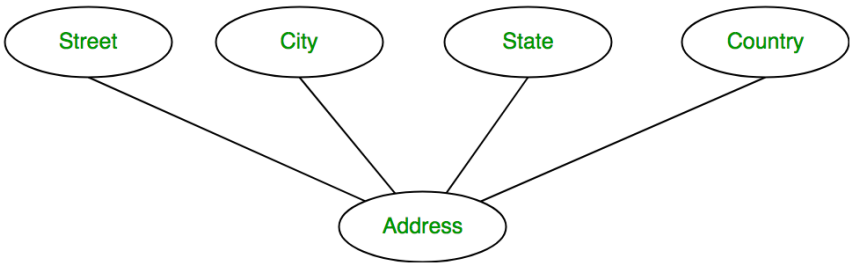


Figure 7: Composite Attribute

c) Multivalued Attribute

An attribute consisting of more than one value for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.

Figure 8. **Multivalued Attribute**

d) Derived Attribute

An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g., Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.

7.1. Recursive Relationships in ER diagram

A relationship between two entities of a similar entity type is called a **recursive** relationship. Here the same entity type participates more than once in a relationship type with a different role for each instance. In other words, a relationship has always been between occurrences in two different entities. However, the same entity can participate in the relationship. This is termed a **recursive** relationship.

Recursive relationships are often used to represent hierarchies or networks, where an entity can be connected to other entities of the same type.

For example, in an organizational chart, an employee can have a relationship with other employees who are also in a managerial position. Similarly, in a social network, a user can have a relationship with other users who are their friends.

To represent a recursive relationship in an ER diagram, we use a self-join, which is a join between a table and itself. In other words, we create a relationship between the same entity type. The self-join involves creating two instances of the same entity and connecting them with a relationship. One instance is considered the parent, and the other instance is considered the child.

We use cardinality constraints to specify the number of instances of the entity that can participate in the relationship. For example, in an organizational chart, an employee can have many subordinates, but each subordinate can only have one manager. This is represented as a one-to-many (1:N) relationship between the employee entity and itself.

Overall, recursive relationships are an important concept in ER modeling, and they allow us to represent complex relationships between entities of the same type. They are particularly useful in modeling hierarchical data structures and networks.

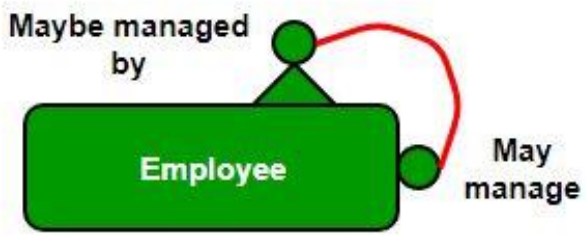


Figure 9: Recursive Relationship

Example: Let us suppose that we have an employee table. A manager supervises a subordinate. Every employee can have a supervisor except the CEO and there can be at most one boss for each employee. One employee may be the boss of more than one employee. Let us suppose that REPORTS_TO is a recursive relationship on the Employee entity type where each Employee plays two roles.

i) Supervisor
ii) Subordinate

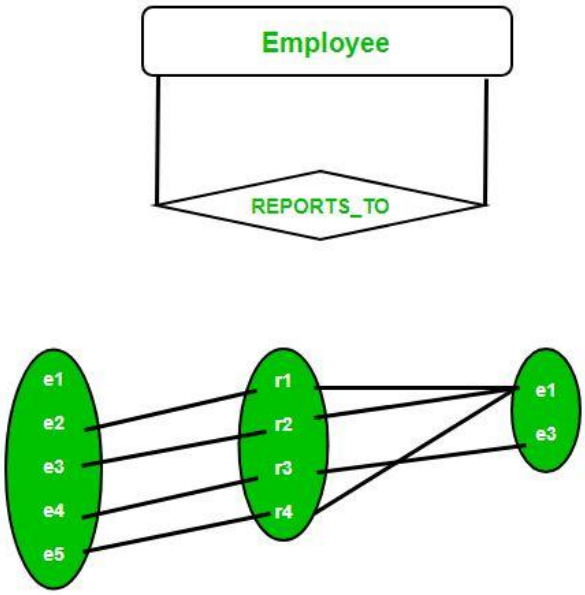


Figure 10: Recursive Relation

Supervisors and subordinates are called **"Role Names."** Here the degree of the REPORTS_TO relationship is 1 i.e., a unary relationship.

- The minimum cardinality of the Supervisor entity is ZERO since the lowest level employee may not be a manager for anyone.
- The maximum cardinality of the Supervisor entity is N since an employee can manage many employees.
- Similarly, the Subordinate entity has a minimum cardinality of ZERO to account for the case where CEO can never be a subordinate.
- Its maximum cardinality is ONE since a subordinate employee can have at most one supervisor.

***Note** – Here none of the participants have total participation since both minimum cardinalities are Zero. Hence, the relationships are connected by a single line instead of a double line in the ER diagram.*

Example

```
CREATE TABLE employee (
    id INT PRIMARY KEY,
    name VARCHAR (50),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employee(id)
);
```

Here, the employee table has a foreign key column called **manager_id** that references the **id** column of the same **employee** table. This allows you to create a recursive relationship where an employee can have a manager who is also an employee.

7.2. Minimization of ER Diagram

Entity-Relationship (ER) Diagram is a diagrammatic representation of data in databases, it shows how data is related to one another. In this article, we require previous knowledge of ER diagrams and how to draw ER diagrams.
Minimization of ER Diagram simply means reducing the quantity of the tables in the ER Diagram. When there are so many tables present in the ER DIagram, it decreases the readability and understandability of the ER Diagram, and it also becomes difficult for the admin also to understand these. Minimizing the ER Diagram helps in better understanding. We reduce tables depending on the cardinality.

Cardinality
The number of times an entity of an entity set participates in a relationship set is known as cardinality. Cardinality can be of different types:

i) One-to-One:

When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one. Let, us assume that a male can marry one female and a female can marry one male. So, the relationship will be one-to-one.
the total number of tables that can be used in this is 2.

Figure 11. **one to one cardinality**

Using Sets, it can be represented as:

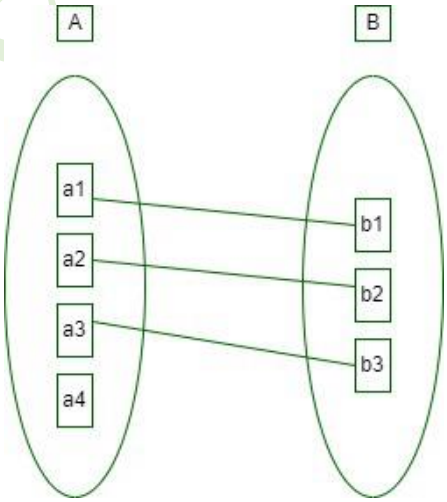


Figure 12. **Set Representation of One-to-One**

ii) One-to-Many:

In one-to-many mapping as well where each entity can be related to more than one relationship and the total number of tables that can be used in this is 2. Let us assume that one surgeon department can accommodate many doctors. So, the Cardinality will be 1 to M. It means one department has many Doctors.
total number of tables that can used is 3.

Figure 13. **one to many cardinality**

iii) Many-to-One:

When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So, the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.
The total number of tables that can be used in this is 3.

Figure 14. **many to one cardinality**

Using Sets, it can be represented as:

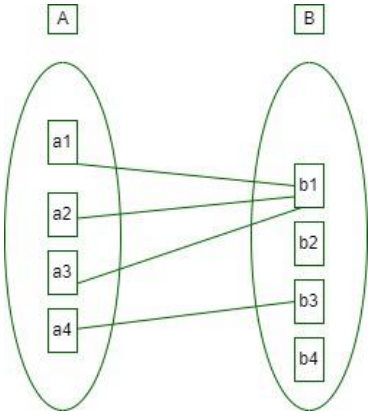


Figure 15. **Set Representation of Many-to-One**

iv) Many-to-Many:

When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So, the relationship will be many to many.
the total number of tables that can be used in this is 3.

Figure 16: **many to many cardinality**

Using Sets, it can be represented as:

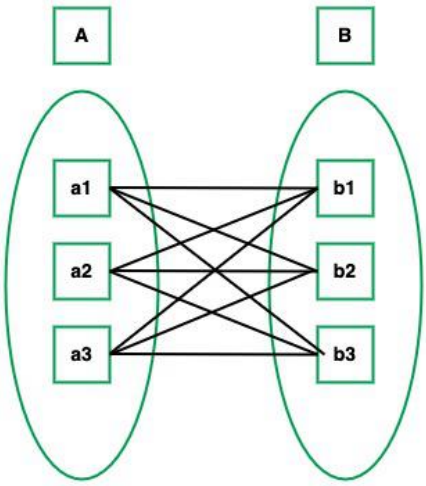


Figure 17: **Many-to-Many Set Representation**

In this example, student S1 is enrolled in C1 and C3 and Course C3 is enrolled by S1, S3, and S4. So, it is many-to-many relationships.

7.3. Enhanced ER Model

Today the complexity of the data is increasing so it becomes more and more difficult to use the traditional ER model for database modeling. To reduce this complexity of modeling we must make improvements or enhancements to the existing ER model to make it able to handle the complex application in a better way.

Enhanced entity-relationship diagrams are advanced database diagrams very similar to regular ER diagrams which represent the requirements and complexities of complex databases.

It is a diagrammatic technique for displaying the Sub Class and Super Class; Specialization and Generalization; Union or Category; Aggregation etc.

Generalization and Specialization: These are very common relationships found in real entities. However, this kind of relationship was added later as an enhanced extension to the classical ER model. **Specialized** classes are often called **subclass** while a **generalized class** is called a superclass, probably inspired by object-oriented programming. A sub-class is best understood by **"IS-A analysis"**. The following statements hopefully make some sense to your mind "Technician IS-A Employee", and "Laptop IS-A Computer".
An entity is a specialized type/class of another entity. For example, a Technician is a special Employee in a university system Faculty is a special class of Employees. We call this phenomenon generalization/specialization. In the example here Employee is a generalized entity class while the Technician and Faculty are specialized classes of Employee.

Example:
This example instance of **"sub-class"** relationships. Here we have four sets of employees: Secretary, Technician, and Engineer. The employee is a super-class of the rest three sets of individual sub-class is a subset of Employee set.

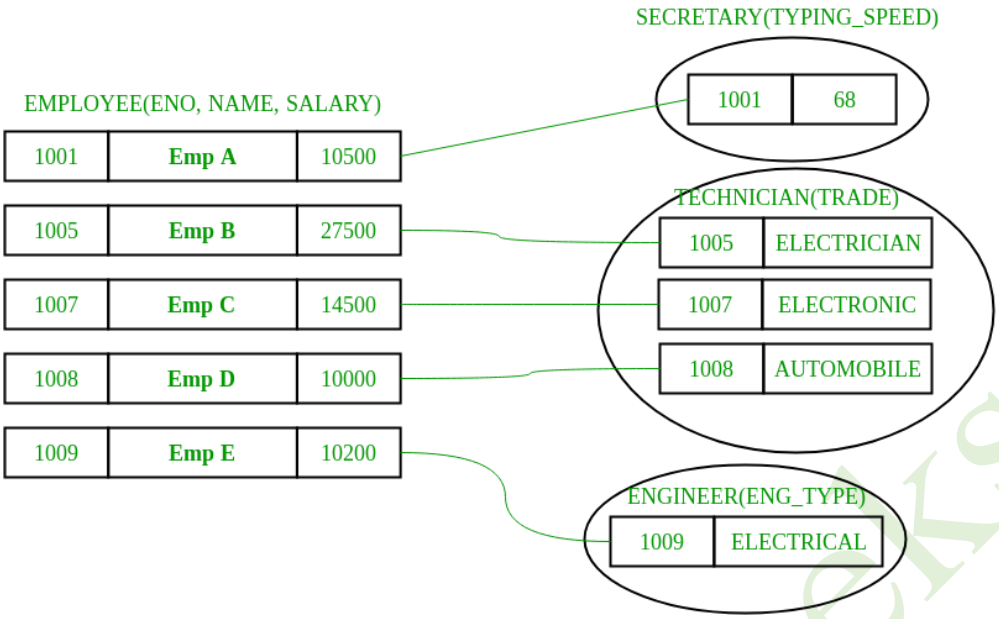


Figure 18: ER Model

- An entity belonging to a sub-class is related to some super-class entity. For instance, emp, no 1001 is a secretary, and his typing speed is 68. Emp no 1009 is an engineer (sub-class) and her trade is “Electrical”, so forth.
- Sub-class entity “inherits” all attributes of super-class; for example, employee 1001 will have attributes eno, name, salary, and typing speed.

Enhanced ER model of above example

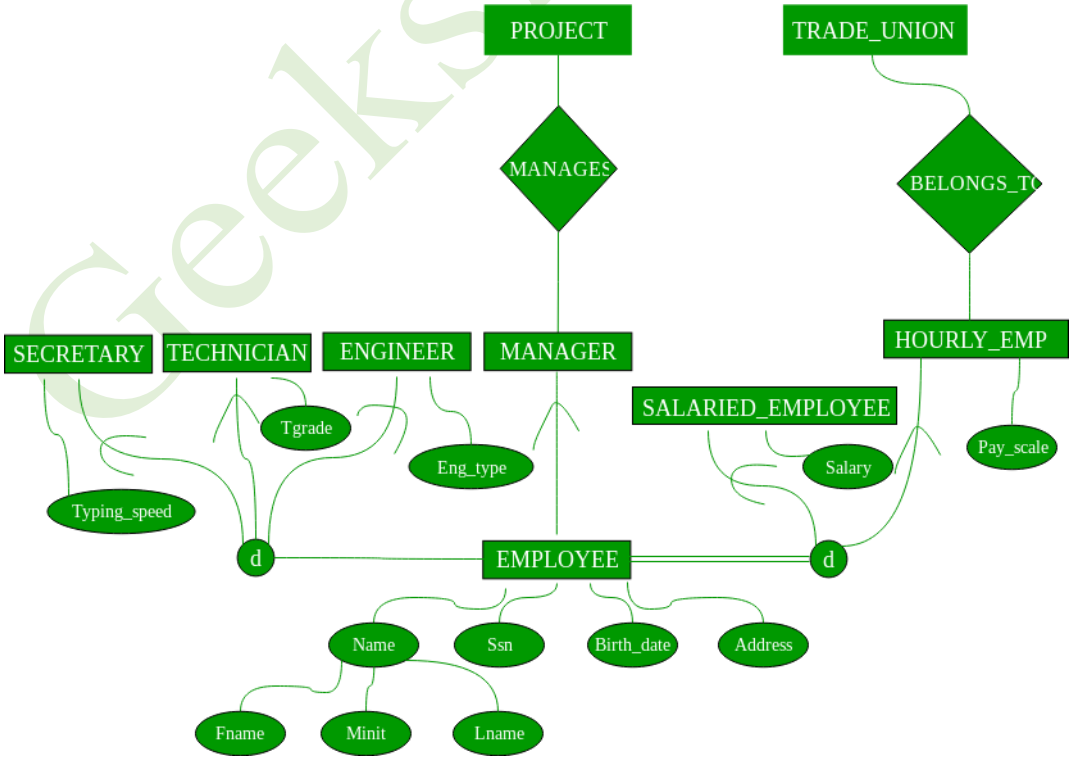


Figure 19: Enhanced ER Model

Constraints – There are two types of constraints on the "Sub-class" relationship.

i) **Total or Partial –**

A sub-classing relationship is total if every super-class entity is to be associated with some sub-class entity, otherwise partial. Sub-class "job type-based employee category" is partial sub-classing – not necessary every employee is one of (secretary, engineer, and technician), i.e. union of these three types is a proper subset of all employees. Whereas other sub-classing "Salaried Employee AND Hourly Employee" is total; the union of entities from sub-classes is equal to the total employee set, i.e. every employee necessarily has to be one of them.

ii) **Overlapped or Disjoint –**

If an entity from a super-set can be related (can occur) in multiple sub-class sets, then it is overlapped sub-classing, otherwise disjoint. Both the examples: job-type based, and salaries/hourly employee sub-classing are disjoint.

Note – *These constraints are independent of each other: can be "overlapped and total or partial" or "disjoint and total or partial." Also, sub-classing has transitive properties.*

Multiple Inheritance (sub-class of multiple superclasses) –

An entity can be a sub-class of multiple entity types; such entities are sub-class of multiple entities and have multiple super-classes; Teaching Assistant can subclass of Employee and Student both. A faculty in a university system can be a subclass of Employee and Alumnus. In multiple inheritances, attributes of sub-class are the union of attributes of all super-classes.

Union –

- Set of Library Members is **UNION** of Faculty, Student, and Staff. A union relationship indicates either type; for example, a library member is either Faculty or Staff or Student.
- Below are two examples that show how **UNION** can be depicted in ERD – Vehicle Owner is UNION of PERSON and Company, and RTO Registered Vehicle is UNION of Car and Truck.

Here are some of the key features of the EER model:

- **Subtypes and Supertypes:** The EER model allow for the creation of subtypes and supertypes. A supertype is a generalization of one or more subtypes, while a subtype is a specialization of a supertype. For example, a vehicle could be a supertype, while car, truck, and motorcycle could be subtypes.
- **Generalization and Specialization:** Generalization is the process of identifying common attributes and relationships between entities and creating a supertype based on these common features. Specialization is the process of identifying unique attributes and relationships between entities and creating subtypes based on these unique features.

- **Inheritance:** Inheritance is a mechanism that allows subtypes to inherit attributes and relationships from their supertype. This means that any attribute or relationship defined for a supertype is automatically inherited by all its subtypes.
- **Constraints:** The EER model allows for the specification of constraints that must be satisfied by entities and relationships. Examples of constraints include cardinality constraints, which specify the number of relationships that can exist between entities, and participation constraints, which specify whether an entity is required to participate in a relationship.
- Overall, the EER model provides a powerful and flexible way to model complex data relationships, making it a popular choice for database design. An Enhanced Entity-Relationship (EER) model is an extension of the traditional Entity-Relationship (ER) model that includes additional features to represent complex relationships between entities more accurately. Some of the main features of the EER model are:
- **Subclasses and Super classes:** EER model allows for the creation of a hierarchical structure of entities where a superclass can have one or more subclasses. Each subclass inherits attributes and relationships from its superclass, and it can also have its unique attributes and relationships.
- **Specialization and Generalization:** EER model uses the concepts of specialization and generalization to create a hierarchy of entities. Specialization is the process of defining subclasses from a superclass, while generalization is the process of defining a superclass from two or more subclasses.
- **Attribute Inheritance:** EER model allows attributes to be inherited from a superclass to its subclasses. This means that attributes defined in the superclass are automatically inherited by all its subclasses.
- **Union Types: E**ER model allows for the creation of a union type, which is a combination of two or more entity types. The union type can have attributes and relationships that are common to all the entity types that make up the union.
- **Aggregation:** EER model allows for the creation of an aggregate entity that represents a group of entities as a single entity. The aggregate entity has its unique attributes and relationships.
- **Multi-valued Attributes:** EER model allows an attribute to have multiple values for a single entity instance. For example, an entity representing a person may have multiple phone numbers.
- **Relationships with Attributes:** EER model allows relationships between entities to have attributes. These attributes can describe the nature of the relationship or provide additional information about the relationship.

7.4. Mapping from ER Model to Relational Model

After designing the ER diagram of system, we need to convert it to Relational models which can directly be implemented by any RDBMS like Oracle, MySQL etc. In this article we will discuss how to convert ER diagram to Relational Model for different scenarios.

Case 1: Binary Relationship with 1:1 cardinality with total participation of an entity

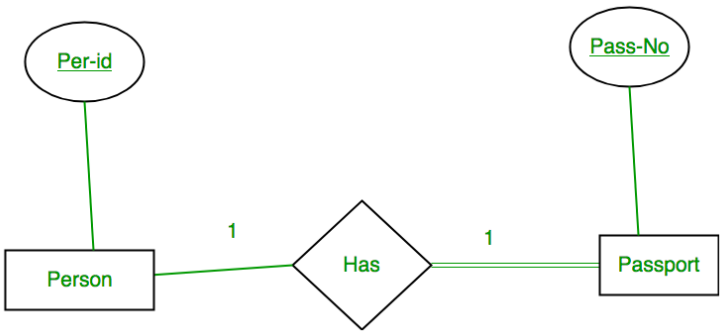


Figure 20: Mapping ER Model to Relational Model

A person has 0 or 1 passport number and Passport is always owned by 1 person. So, it is 1:1 cardinality with full participation constraint from Passport.

First Convert each entity and relationship to tables. Person table corresponds to Person Entity with key as Per-Id. Similarly, Passport table corresponds to Passport Entity with key as Pass-No. Has Table represents relationship between Person and Passport (Which person has which passport). So, it will take attribute Per-Id from Person and Pass-No from Passport.

Person			**Has**			**Passport**	
Per-Id	**Other Person Attribute**		**Per-Id**	**Pass-No**		**Pass-No**	**Other PassportAttribute**
PR1	–		PR1	PS1		PS1	–
PR2	–		PR2	PS2		PS2	–
PR3	–						

Table 1

As we can see from Table 1, each Per-Id and Pass-No has only one entry in Has Table. So we can merge all three tables into 1 with attributes shown in Table 2. Each Per-Id will be unique and not null. So it will be the key. Pass-No can't be key because for some person, it can be NULL.

Per-ID	Other Person Attribute	Pass-No	Other Passport Attribute

Table 2

Case 2: Binary Relationship with 1:1 cardinality and partial participation of both entities

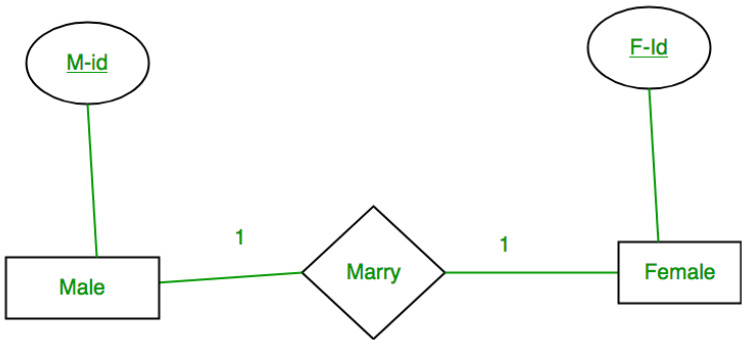


Figure 21: Binary Relationship 1:1

A male marries 0 or 1 female and vice versa as well. So, it is 1:1 cardinality with partial participation constraint from both. First Convert each entity and relationship to tables. Male table corresponds to Male Entity with key as M-Id. Similarly Female table corresponds to Female Entity with key as F-Id. Marry Table represents relationship between Male and Female (Which Male marries which female). So, it will take attribute M-Id from Male and F-Id from Female.

Male			**Marry**			**Female**	
M-Id	Other Male Attribute		M-Id	F-Id		F-Id	Other FemaleAttribute
M1	–		M1	F2		F1	–
M2	–		M2	F1		F2	–
M3	–					F3	–

Table 3

As we can see from Table 3, some males and some females do not marry. If we merge 3 tables into 1, for some M-Id, F-Id will be NULL. So, there is no attribute which is always not NULL. So, we cannot merge all three tables into 1. We can convert into 2 tables. In table 4, M-Id who are married will have F-Id associated. For others, it will be NULL. Table 5 will have information of all females. Primary Keys have been underlined.

M-Id	Other male Attribute	F-Id

Table 4

F-Id	Other Female Attribute

Table 5

Case 3: Binary Relationship with n: 1 cardinality

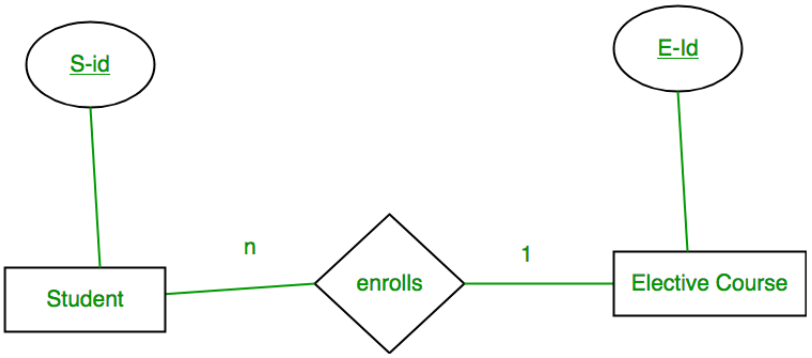


Figure 22: Binary Relationship with n:1 cardinality

In this scenario, every student can enroll only in one elective course but for an elective course there can be more than one student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Elective_Course table corresponds to Elective_Course Entity with key as E-Id. Enrolls Table represents relationship between Student and Elective_Course (Which student enrolls in which course). So it will take attribute S-Id from Student and E-Id from Elective_Course.

Student			**Enrolls**			**Elective_Course**	
S-Id	Other Student Attribute		S-Id	E-Id		E-Id	Other Elective CourseAttribute
S1	–		S1	E1		E1	–
S2	–		S2	E2		E2	–
S3	–		S3	E1		E3	–
S4	–		S4	E1			

Table 6

As we can see from Table 6, S-Id is not repeating in Enrolls Table. So, it can be considered as a key of Enrolls table. Both Student and Enrolls Table's key is same; we can merge it as a single table. The resultant tables are shown in Table 7 and Table 8. Primary Keys have been underlined.

S-Id	Other Student Attribute	E-Id

Table 7

E-Id	Other Elective Course Attribute

Table 8

Case 4: Binary Relationship with m: n cardinality

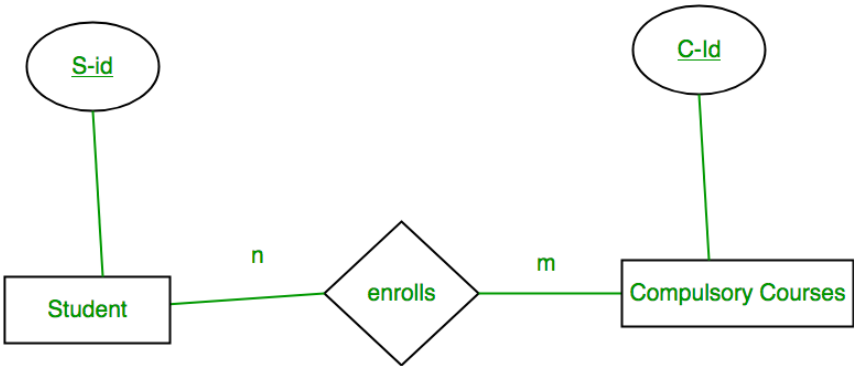


Figure 23: Binary Relation with m: n cardinality

In this scenario, every student can enroll in more than 1 compulsory course and for a compulsory course there can be more than 1 student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Compulsory_Courses table corresponds to Compulsory Courses Entity with key as C-Id. Enrolls Table represents relationship between Student and Compulsory_Courses (Which student enrolls in which course). So, it will take attribute S-Id from Person and C-Id from Compulsory_Courses.

Student			**Enrolls**			**Compulsory_Courses**	
S-Id	Other Student Attribute		S-Id	C-Id		C-Id	Other Compulsory CourseAttribute
S1	–		S1	C1		C1	–
S2	–		S1	C2		C2	–
S3	–		S3	C1		C3	–
S4	–		S4	C3		C4	–
			S4	C2			
			S3	C3			

Table 9

As we can see from Table 9, S-Id and C-Id both are repeating in Enrolls Table. But its combination is unique; so it can be considered as a key of Enrolls table. All tables' keys are different, these can't be merged. Primary Keys of all tables have been underlined.

Case 5: Binary Relationship with weak entity

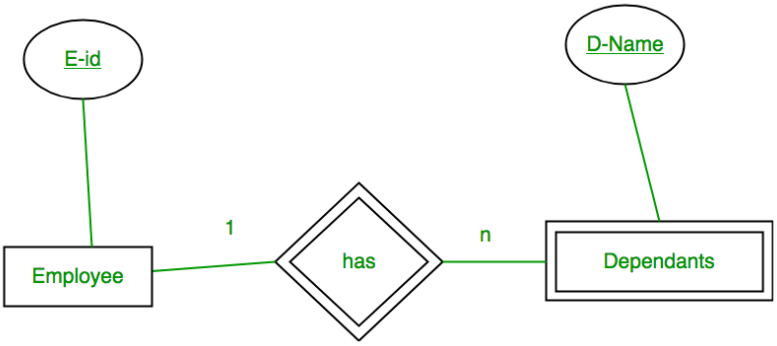


Figure 24: Binary Relationship with weak entity

In this scenario, an employee can have many dependents and one dependent can depend on one employee. A dependent does not have any existence without an employee (e.g; you as a child can be dependent of your father in his company). So, it will be a weak entity and its participation will always be total. Weak Entity does not have key of its own. So, its key will be combination of key of its identifying entity (E-Id of Employee in this case) and its partial key (D-Name). First Convert each entity and relationship to tables. Employee table corresponds to Employee Entity with key as E-Id. Similarly, Dependents table corresponds to Dependent Entity with key as D-Name and E-Id. Has Table represents relationship between Employee and Dependents (Which employee has which dependents). So, it will take attribute E-Id from Employee and D-Name from Dependents.

Employee			**Has**			**Dependents**		
E-Id	Other Employee Attribute		E-Id	D-Name		D-Name	E-Id	Other DependentsAttribute
E1	–		E1	RAM		RAM	E1	–
E2	–		E1	SRINI		SRINI	E1	–
E3	–		E2	RAM		RAM	E2	–
			E3	ASHISH		ASHISH	E3	–

Table 10

As we can see from Table 10, E-Id, D-Name is key for **Has** as well as Dependents Table. So, we can merge these two into 1. So the resultant tables are shown in Tables 11 and 12. Primary Keys of all tables have been underlined.

E-Id	Other Employee Attribute

Table 11

D-Name	E-Id	Other DependentsAttribute

Table 12

Relational Model

8. Introduction to Relational Model

E.F. Codd proposed the relational Model to model data in the form of relations or tables. After designing the conceptual model of the Database using ER diagram, we need to convert the conceptual model into a relational model which can be implemented using any RDBMS language like Oracle SQL, MySQL, etc. So, we will see what the Relational Model is.

What is the Relational Model?

The relational model represents how data is stored in Relational Databases. A relational database consists of a collection of tables, each of which is assigned a unique name. Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE, and AGE shown in the table.

Table Student

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	**RAM**	**DELHI**	9455123451	**18**
2	**RAMESH**	**GURGAON**	9652431543	**18**
3	**SUJIT**	**ROHTAK**	9156253131	**20**
4	**SURESH**	**DELHI**		**18**

Important Terminologies

- **Attribute:** Attributes are the properties that define an entity. e.g., **ROLL_NO**, **NAME, ADDRESS**
- **Relation Schema:** A relation schema defines the structure of the relation and represents the name of the relation with its attributes. e.g., STUDENT (ROLL_NO, NAME, ADDRESS, PHONE, and AGE) is the relation schema for STUDENT. If a schema has more than 1 relation, it is called Relational Schema.
- **Tuple:** Each row in the relation is known as a tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	**9772667313**	18

- **Relation Instance:** The set of tuples of a relation at a particular instance of time is called a relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is an insertion, deletion, or update in the database.
- **Degree:** The number of attributes in the relation is known as the degree of the relation. The **STUDENT** relation defined above has degree 5.

- **Cardinality:** The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.
- **Column:** The column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from the relation STUDENT.

ROLL_NO
1
2
3
4

- **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by blank space. e.g., PHONE of STUDENT having ROLL_NO 4 is NULL.
- **Relation Key:** These are basically the keys that are used to identify the rows uniquely or also help in identifying tables. These are of the following types.
    - Primary Key
    - Candidate Key
    - Super Key
    - Foreign Key
    - Alternate Key
    - Composite Key

8.1. Coddy Rule and it's Features

Features of the relational model and Codd's Rules :

Tables/Relations: The basic building block of the relational model is the table or relation, which represents a collection of related data. Each table consists of columns, also known as attributes or fields, and rows, also known as tuples or records.

Primary Keys: In the relational model, each row in a table must have a unique identifier, which is known as the primary key. This ensures that each row is unique, can be accessed, and manipulated easily.

Foreign Keys: Foreign keys are used to link tables together and enforce referential integrity. They ensure that data in one table is consistent with data in another table.

Normalization: The process of organizing data into tables and eliminating redundancy is known as normalization. Normalization is important in the relational model because it helps to ensure that data is consistent and easy to maintain.

Codd's Rules –
Codd's Rules are a set of 12 rules that define the characteristics of a true relational DBMS. These rules ensure that the DBMS is consistent, reliable, and easy to use.
Atomicity, Consistency, Isolation, Durability (ACID): The ACID properties are a set of properties that ensure that transactions are processed reliably in the relational model. Transactions are sets of operations that are executed as a single unit, ensuring that data is consistent and accurate.

Advantages of Relational Algebra
Relational Algebra is a formal language used to specify queries to retrieve data from a relational database. It has several advantages that make it a popular choice for managing and manipulating data. Here are some of the advantages of

Relational Algebra:

- **Simplicity:** Relational Algebra provides a simple and easy-to-understand set of operators that can be used to manipulate data. It is based on a set of mathematical concepts and principles, which makes it easy to learn and use.
- **Formality:** Relational Algebra is a formal language that provides a standardized and rigorous way of expressing queries. This makes it easier to write and debug queries, and also ensures that queries are correct and consistent.
- **Abstraction:** Relational Algebra provides a high-level abstraction of the underlying database structure, which makes it easier to work with large and complex databases. It allows users to focus on the logical structure of the data, rather than the physical storage details.
- **Portability:** Relational Algebra is independent of any specific database management system, which means that queries can be easily ported to other systems. This makes it easy to switch between different databases or vendors without having to rewrite queries.
- **Efficiency:** Relational Algebra is optimized for efficiency and performance, which means that queries can be executed quickly and with minimal resources. This is particularly important for large and complex databases, where performance is critical.
- **Extensibility:** Relational Algebra provides a flexible and extensible framework that can be extended with new operators and functions. This allows developers to customize and extend the language to meet their specific needs.

Disadvantages of Relational Algebra

While Relational Algebra has many advantages, it also has some limitations and disadvantages that should be considered when using it.

Here are some of the disadvantages of Relational Algebra:

- **Complexity:** Although Relational Algebra is based on mathematical principles, it can be complex and difficult to understand for non-experts. The syntax and semantics of the language can be challenging, and it may require significant training and experience to use it effectively.
- **Limited Expressiveness:** Relational Algebra has a limited set of operators, which can make it difficult to express certain types of queries. It may be necessary to use more advanced techniques, such as subqueries or joins, to express complex queries.
- **Lack of Flexibility:** Relational Algebra is designed for use with relational databases, which means that it may not be well-suited for other types of data storage or management systems. This can limit its flexibility and applicability in certain contexts.
- **Performance Limitations:** While Relational Algebra is optimized for efficiency and performance, it may not be able to handle large or complex datasets. Queries can become slow and resource-intensive when dealing with large amounts of data or complex queries.
- **Limited Data Types:** Relational Algebra is designed for use with simple data types, such as integers, strings, and dates. It may not be well-suited for more complex data types, such as multimedia files or spatial data.
- **Lack of Integration:** Relational Algebra is often used in conjunction with other programming languages and tools, which can create integration challenges. It may require additional programming effort to integrate Relational Algebra with other systems and tools.

Relational Algebra is a powerful and useful tool for managing and manipulating data in relational databases, it has some limitations and disadvantages that should be carefully considered when using it.

Codd's Twelve Rules of Relational Database

Codd rules were proposed by E.F. Codd which should be satisfied by the relational model. Codd's Rules are basically used to check whether DBMS has the quality to become Relational Database Management System (RDBMS). But, it is rare to find that any product has fulfilled all the rules of Codd. They generally follow the 8-9 rules of Codd. E.F. Codd has proposed 13 rules which are popularly known as Codd's 12 rules. These rules are stated as follows:

- **Rule 0: Foundation Rule**– For any system that is advertised as, or claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities.

- **Rule 1: Information Rule**– Data stored in the Relational model must be a value of some cell of a table.
- **Rule 2: Guaranteed Access Rule**– Every data element must be accessible by the table name, its primary key, and the name of the attribute whose value is to be determined.
- **Rule 3: Systematic Treatment of NULL values**– NULL value in the database must only correspond to missing, unknown, or not applicable values.
- **Rule 4: Active Online Catalog**– The structure of the database must be stored in an online catalog that can be queried by authorized users.
- **Rule 5: Comprehensive Data Sub-language Rule-** A database should be accessible by a language supported for definition, manipulation, and transaction management operation.
- **Rule 6: View Updating Rule-** Different views created for various purposes should be automatically updatable by the system.
- **Rule 7: High-level insert, update and delete rule-** Relational Model should support insert, delete, update, etc. operations at each level of relations. Also, set operations like Union, Intersection, and minus should be supported.
- **Rule 8: Physical data independence-** Any modification in the physical location of a table should not enforce modification at the application level.
- **Rule 9: Logical data independence-** Any modification in the logical or conceptual schema of a table should not enforce modification at the application level. For example, merging two tables into one should not affect the application accessing it which is difficult to achieve.
- **Rule 10: Integrity Independence-** Integrity constraints modified at the database level should not enforce modification at the application level.
- **Rule 11: Distribution Independence-** Distribution of data over various locations should not be visible to end-users.
- **Rule 12: Non-Subversion Rule- Low-level** access to data should not be able to bypass the integrity rule to change data.

Question. Given the basic ER and relational models, which of the following is INCORRECT? [GATE CS 2012]
(A) An attribute of an entity can have more than one value.
(B) An attribute of an entity can be a composite.
(C) In a row of a relational table, an attribute can have more than one value.
(D) In a row of a relational table, an attribute can have exactly one value or a NULL value.
Answer: In the relation model, an attribute can't have more than one value. So, option 3 is the answer.

Constraints in Relational Model

While designing the Relational Model, we define some conditions which must hold for data present in the database are called Constraints. These constraints are checked before performing any operation (insertion, deletion, and updation) in the database. If there is a violation of any of the constraints, the operation will fail.

Domain Constraints

These are attribute-level constraints. An attribute can only take values that lie inside the domain range. e.g.; If a constraint AGE>0 is applied to STUDENT relation, inserting a negative value of AGE will result in failure.

Key Integrity

Every relation in the database should have at least one set of attributes that defines a tuple uniquely. Those set of attributes is called keys. e.g.; ROLL_NO in STUDENT is key. No two students can have the same roll number. So, a key has two properties:

- It should be unique for all tuples.
- It cannot have NULL values.

Referential Integrity

When one attribute of a relation can only takes values from another attribute of the same relation or any other relation, it is called referential integrity. Let us suppose we have 2 relations.

Table Student

ROLL_NO	NAME	ADDRESS	PHONE	AGE	BRANCH_CODE
1	RAM	DELHI	9455123451	18	CS
2	RAMESH	GURGAON	9652431543	18	CS
3	SUJIT	ROHTAK	9156253131	20	ECE
4	SURESH	DELHI		18	IT

Table Branch

BRANCH_CODE	BRANCH_NAME
CS	COMPUTER SCIENCE
IT	INFORMATION TECHNOLOGY
ECE	ELECTRONICS AND COMMUNICATION ENGINEERING
CV	CIVIL ENGINEERING

BRANCH_CODE of STUDENT can only take the values which are present in BRANCH_CODE of BRANCH which is called referential integrity constraint. The relation which is referencing another relation is called REFERENCING RELATION (STUDENT in this case) and the relation to which other relations refer is called REFERENCED RELATION (BRANCH in this case).

Relational Schema and Instances

Relational Schema

In the context of databases, a relational schema represents the blueprint or structure of a relational database. It defines the organization of data in the form of tables, specifying the names of the tables, the names of the attributes (columns), and the data types associated with each attribute. The relational schema essentially outlines the framework that governs how data is stored and organized within the database.

Instances

Instances, refer to the actual data contained within the tables of a relational database at a specific point in time. They represent the rows or records within the tables, each row corresponding to a unique set of values that align with the defined attributes in the schema. These instances reflect the real-world data that is stored, retrieved, and manipulated within the database system.

Understanding the distinction between the relational schema, which outlines the structure, and instances, which represent the actual data entries, is crucial for comprehending how data is organized and managed within a relational database system.

8.2. Anomalies in Relational Model

Anomalies in the relational model refer to inconsistencies or errors that can arise when working with relational databases, specifically in the context of data insertion, deletion, and modification. There are different types of anomalies that can occur in referencing and referenced relations which can be discussed as:

These anomalies can be categorized into three types:

- Insertion Anomalies
- Deletion Anomalies
- Update Anomalies.

How Are Anomalies Caused in DBMS?

Database anomalies are the faults in the database caused due to poor management of storing everything in the flat database. It can be removed with the process of Normalization, which generally splits the database which results in reducing the anomalies in the database.

STUDENT Table 1

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD-COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajasthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUDENT_COURSE

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

Table 2

Insertion Anomaly: If a tuple is inserted in referencing relation and referencing attribute value is not present in referenced attribute, it will not allow insertion in referencing relation.

Example:

If we try to insert a record in STUDENT_COURSE with STUD_NO =7, it will not allow it.

Deletion and Updation Anomaly: If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will not allow deleting the tuple from referenced relation.

Example: If we want to update a record from STUDENT_COURSE with STUD_NO =1, We must update it in both rows of the table. If we try to delete a record from STUDENT with STUD_NO =1, it will not allow it.

To avoid this, the following can be used in query:

- **ON DELETE/UPDATE SET NULL:** If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and set the value of referencing attribute to NULL.
- **ON DELETE/UPDATE CASCADE:** If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and referencing relation as well.

How These Anomalies Occur?

- **Insertion Anomalies:** These anomalies occur when it is not possible to insert data into a database because the required fields are missing or because the data is incomplete. For example, if a database requires that every record has a primary key, but no value is provided for a particular record, it cannot be inserted into the database.
- **Deletion anomalies:** These anomalies occur when deleting a record from a database and can result in the unintentional loss of data. For example, if a database contains information about customers and orders, deleting a customer record may also delete all the orders associated with that customer.

- **Update anomalies:** These anomalies occur when modifying data in a database and can result in inconsistencies or errors. For example, if a database contains information about employees and their salaries, updating an employee's salary in one record but not in all related records could lead to incorrect calculations and reporting.

Removal of Anomalies

These anomalies can be avoided or minimized by designing databases that adhere to the principles of normalization. Normalization involves organizing data into tables and applying rules to ensure data is stored in a consistent and efficient manner. By reducing data redundancy and ensuring data integrity, normalization helps to eliminate anomalies and improve the overall quality of the database

According to **E.F.Codd**, who is the inventor of the Relational Database, the goals of Normalization include:

- It helps in vacating all the repeated data from the database.
- It helps in removing undesirable deletion, insertion, and update anomalies.
- It helps in making a proper and useful relationship between tables.

Advantages Anomalies in Relational Model

- **Data Integrity:** Relational databases enforce data integrity through various constraints such as primary keys, foreign keys, and referential integrity rules, ensuring that the data is accurate and consistent.
- **Scalability:** Relational databases are highly scalable and can handle large amounts of data without sacrificing performance.
- **Flexibility:** The relational model allows for flexible querying of data, making it easier to retrieve specific information and generate reports.
- **Security:** Relational databases provide robust security features to protect data from unauthorized access.

Disadvantages of Anomalies in Relational Model

- **Redundancy:** When the same data is stored in various locations, a relational architecture may cause data redundancy. This can result in inefficiencies and even inconsistent data.
- **Complexity:** Establishing and keeping up a relational database calls for specific knowledge and abilities and can be difficult and time-consuming.
- **Performance:** Because more tables must be joined in order to access information, performance may degrade as a database gets larger.
- **Incapacity to manage unstructured data:** Text documents, videos, and other forms of semi-structured or unstructured data are not well-suited for the relational paradigm.

8.3. Key and Superkey Concepts

Key:
In the context of a database, a key is a unique attribute or a set of attributes that can uniquely identify each record in a table. It helps in maintaining the integrity and consistency of data by ensuring that there are no duplicate records. In simpler terms, a key act as a unique identifier for each row in a table, enabling efficient retrieval, updating, and deletion of data. It serves as a primary means of establishing relationships between different tables within the database.

Superkey:
A superkey refers to a set of one or more attributes that, when taken collectively, can uniquely identify each record within a table. Unlike a key, a superkey may contain additional attributes that are not necessary for uniquely identifying each record. This means that a superkey can have more attributes than the minimum required for uniqueness. In essence, a superkey acts as a broader identifier encompassing one or more keys within it.
Understanding the distinction between a key and a superkey is crucial for designing efficient and reliable databases. Keys are the minimal set of attributes required for unique identification, while superkey represent a broader set of attributes that include the key and possibly more. The proper identification and utilization of keys and superkey are essential for maintaining data integrity and establishing relationships within the database.
Any set of attributes that allows us to identify unique rows (tuples) in a given relationship is known as super keys. Out of these super keys, we can always choose a proper subset among these that can be used as a primary key. Such keys are known as Candidate keys. If there is a combination of two or more attributes that are being used as the primary key then we call it a Composite key.

8.4. Tuple Relational Calculus

Tuple Relational Calculus (TRC) is a non-procedural query language used in relational database management systems (RDBMS) to retrieve data from tables. TRC is based on the concept of tuples, which are ordered sets of attribute values that represent a single row or record in a database table.
TRC is a declarative language, meaning that it specifies what data is required from the database, rather than how to retrieve it. TRC queries are expressed as logical formulas that describe the desired tuples.
Syntax: The basic syntax of TRC is as follows:
{ t | P(t) }

where it is a **tuple variable** and P(t) is a **logical formula** that describes the conditions that the tuples in the result must satisfy. The **curly braces {}** are used to indicate that the expression is a set of tuples.

For example, let's say we have a table called "Employees" with the following attributes:

Employee ID
Name
Salary
Department ID

To retrieve the names of all employees who earn more than $50,000 per year, we can use the following TRC query:

{t | Employees(t) ∧ t.Salary > 50000 }

In this query, the "Employees(t)" expression specifies that the tuple variable t represents a row in the "Employees" table. The "∧" symbol is the logical AND operator, which is used to combine the condition "t.Salary > 50000" with the table selection.

The result of this query will be a set of tuples, where each tuple contains the Name attribute of an employee who earns more than $50,000 per year.

TRC can also be used to perform more complex queries, such as joins and nested queries, by using additional logical operators and expressions.

While TRC is a powerful query language, it can be more difficult to write and understand than other SQL-based query languages, such as Structured Query Language (SQL). However, it is useful in certain applications, such as in the formal verification of database schemas and in academic research.
Tuple Relational Calculus is a **non-procedural query language,** unlike relational algebra. Tuple Calculus provides only the description of the query but it does not provide the methods to solve it. Thus, it explains what to do but not how to do it.

Tuple Relational Query
In Tuple Calculus, a query is expressed as {**t| P(t)}**
where t = resulting tuples,
P(t) = known as Predicate and these are the conditions that are used to fetch t.
Thus, it generates a set of all tuples t, such that Predicate P(t) is true for t.
P(t) may have various conditions logically combined with OR (∨), AND (∧), NOT(¬).
It also uses quantifiers:
∃ t ∈ r (Q(t)) =" there exists" a tuple in t in relation r such that predicate Q(t) is true.
∀ t ∈ r (Q(t)) = Q(t) is true "for all" tuples in relation r.

5.2 Tuple Relational Calculus Examples

Table Customer

Customer Name	Street	City
Saurabh	A7	Patiala
Mehak	B6	Jalandhar
Sumiti	D9	Ludhiana
Ria	A5	Patiala

Table Branch

Branch Name	Branch City
ABC	Patiala
DEF	Ludhiana
GHI	Jalandhar

Table Account

Account number	Branch name	Balance
1111	ABC	50000
1112	DEF	10000
1113	GHI	9000
1114	ABC	7000

Table Loan

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L49	GHI	9000
L98	DEF	65000

Table Borrower

Customer name	Loan number
Saurabh	L33
Mehak	L49
Ria	L98

Table Depositor

Customer name	Account number
Saurabh	1111
Mehak	1113
Suniti	1114

Example 1: Find the loan number, branch, and amount of loans greater than or equal to 10000 amounts.
{t| t ∈ loan ∧ t[amount]>=10000}
Resulting relation:

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L98	DEF	65000

In the above query, t[amount] is known as a tuple variable.

8.5. Extended Operation in Relational Algebra

Extended operators are those operators which can be derived from basic operators. There are mainly three types of extended operators in Relational Algebra:

- **Join**
- **Intersection**
- **Divide**

The relations used to understand extended operators are STUDENT, STUDENT_SPORTS, ALL_SPORTS and EMPLOYEE which are shown in Table 1, Table 2, Table 3, and Table 4, respectively.

STUDENT Table (Table 1)

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

STUDENT_SPORTS (Table 2)

ROLL_NO	SPORTS
1	Badminton
2	Cricket
2	Badminton
4	Badminton

ALL_SPORTS (Table 3)

SPORTS
Badminton
Cricket

EMPLOYEE (Table 4)

EMP_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
5	NARESH	HISAR	9782918192	22
6	SWETA	RANCHI	9852617621	21
4	SURESH	DELHI	9156768971	18

Intersection (?): Intersection on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relations should have same number of attributes and corresponding attributes in two relations have same domain). Intersection operator when applied on two relations as R1**?** R2 will give a relation with tuples which are in R1 as well as R2. Syntax:

Relation1? Relation2

Example: Find a person who is student as well as employee- STUDENT**?** **EMPLOYEE**

In terms of basic operators (union and minus):

STUDENT ? EMPLOYEE = STUDENT + EMPLOYEE - (STUDENT U EMPLOYEE)

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18

Conditional Join(?$_c$): Conditional Join is used when you want to join two or more relation based on some conditions. Example: Select students whose ROLL_NO is greater than EMP_NO of employees

STUDENT?*$_c$ STUDENT.ROLL_NO>EMPLOYEE.EMP_NOEMPLOYEE*

In terms of basic operators (cross product and selection) :

? (STUDENT.ROLL_NO>EMPLOYEE.EMP_NO)(STUDENT×EMPLOYEE)

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI

Equijoin(?): Equijoin is a **special case of conditional join** where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result. Example: Select students whose ROLL_NO is equal to EMP_NO of employees.

STUDENT?STUDENT.ROLL_NO=EMPLOYEE.EMP_NOEMPLOYEE

In terms? of basic operators (cross product, selection and projection) :

?(STUDENT.ROLL_NO, STUDENT.NAME, STUDENT.ADDRESS, STUDENT.PHONE, STUDENT.AGE EMPLOYEE.NAME, EMPLOYEE.ADDRESS, EMPLOYEE.PHONE, EMPLOYEE>AGE)(? (STUDENT.ROLL_NO=EMPLOYEE.EMP_NO) (STUDENT×EMPLOYEE))

RESULT:

RO LL_NO	NAME	ADD RESS	PHONE	AGE	NAME	ADD RESS	PHONE	AG
1	RAM	DELHI	9455123451	18	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	SURESH	DELHI	9156768971	18

Natural Join(?): It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations R and S (relations on which join operation is applied). While applying natural join on two relations, there is no need to write equality condition explicitly. Natural Join will also return the similar attributes only once as their value will be same in resulting relation. Example: Select students whose ROLL_NO is equal to ROLL_NO of STUDENT_SPORTS as:

STUDENT?STUDENT_SPORTS

In terms of basic operators (cross product, selection and projection) :

?(STUDENT.ROLL_NO, STUDENT.NAME, STUDENT.ADDRESS, STUDENT.PHONE, STUDENT.AGE STUDENT_SPORTS.SPORTS)(? (STUDENT.ROLL_NO=STUDENT_SPORTS.ROLL_NO) (STUDENT×STUDENT_SPORTS))

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	SPORTS
1	RAM	DELHI	9455123451	18	Badminton
2	RAMESH	GURGAON	9652431543	18	Cricket
2	RAMESH	GURGAON	9652431543	18	Badminton
4	SURESH	DELHI	9156768971	18	Badminton

Natural Join is by default inner join because the tuples which does not satisfy the conditions of join does not appear in result set. e.g.; The tuple having

ROLL_NO 3 in STUDENT does not match with any tuple in STUDENT_SPORTS, so it has not been a part of result set.

Left Outer Join(?): When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Left Outer Joins gives all tuples of R in the result set. The tuples of R which do not satisfy join condition will have values as NULL for attributes of S. Example:Select students whose ROLL_NO is greater than EMP_NO of employees and details of other students as well

STUDENT?STUDENT.ROLL_NO>EMPLOYEE.EMP_NOEMPLOYEE

RESULT

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

Right Outer Join(?): When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Right Outer Joins gives all tuples of S in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R. Example: Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well

STUDENT?STUDENT.ROLL_NO>EMPLOYEE.EMP_NOEMPLOYEE

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18

Full Outer Join(?): When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Full Outer Joins gives all tuples of S and all tuples of R in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R and vice versa. Example:
Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well and other Students as well
STUDENT?STUDENT.ROLL_NO>EMPLOYEE.EMP_NOEMPLOYEE
RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

Advantages:

- **Expressive Power:** Extended operators allow for more complex queries and transformations that cannot be easily expressed using basic relational algebra operations.
- **Data Reduction:** Aggregation operators, such as SUM, AVG, COUNT, and MAX, can reduce the amount of data that needs to be processed and displayed.
- **Data Transformation:** Extended operators can be used to transform data into different formats, such as pivoting rows into columns or vice versa.
- **More Efficient:** Extended operators can be more efficient than expressing the same query in terms of basic relational algebra operations, since they can take advantage of specialized algorithms and optimizations.

Disadvantages:

- **Complexity:** Extended operators can be more difficult to understand and use than basic relational algebra operations. They require a deeper understanding of the underlying data and the operators themselves.
- **Performance:** Some extended operators, such as outer joins, can be expensive in terms of performance, especially when dealing with large data sets.
- **Non-standardized:** There is no universal set of extended operators, and different relational database management systems may implement them differently or not at all.
- **Data Integrity:** Some extended operators, such as aggregate functions, can introduce potential problems with data integrity if not used properly. For example, using AVG on a column that contains null values can result in unexpected or incorrect results.

9. Introduction to Database Normalization

Database normalization is the process of organizing the attributes of the database to reduce or eliminate **data redundancy (having the same data but at different places)**.

Problems because of data redundancy: Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete and update operations.

Functional Dependency: Functional Dependency is a constraint between two sets of attributes in relation to a database. A functional dependency is denoted by an arrow (?). If an attribute A functionally determines B, then it is written as A ? B.

For example, employee_id ? name means employee_id functionally determines the name of the employee. As another example in a timetable database, {student_id, time} ? {lecture_room}, student ID and time determine the lecture room where the student should be.

Advantages of Functional Dependency

- The database's data quality is maintained using it.
- It communicates the database design's facts.
- It aids in precisely outlining the limitations and implications of databases.
- It is useful to recognize poor designs.
- Finding the potential keys in the relationship is the first step in the normalization procedure. Identifying potential keys and normalizing the database without functional dependencies is impossible.

What does functionally dependent mean?

A function dependency A ? B means for all instances of a particular value of A, there is the same value of B. For example, in the below table A ? B is true, but B ? A is not true as there are different values of A for B = 3.

A	B
1	3
2	3
4	0
1	3
4	0

The features of database normalization are as follows:

- **Elimination of Data Redundancy:** One of the main features of normalization is to eliminate the data redundancy that can occur in a database. Data redundancy refers to the repetition of data in different parts of the database. Normalization helps in reducing or eliminating this redundancy, which can improve the efficiency and consistency of the database.
- **Ensuring Data Consistency:** Normalization helps in ensuring that the data in the database is consistent and accurate. By eliminating redundancy, normalization helps in preventing inconsistencies and contradictions that can arise due to different versions of the same data.
- **Simplification of Data Management:** Normalization simplifies the process of managing data in a database. By breaking down a complex data structure into simpler tables, normalization makes it easier to manage the data, update it, and retrieve it.
- **Improved Database Design:** Normalization helps in improving the overall design of the database. By organizing the data in a structured and systematic way, normalization makes it easier to design and maintain the database. It also makes the database more flexible and adaptable to changing business needs.
- **Avoiding Update Anomalies:** Normalization helps in avoiding update anomalies, which can occur when updating a single record in a table affects multiple records in other tables. Normalization ensures that each table contains only one type of data and that the relationships between the tables are clearly defined, which helps in avoiding such anomalies.
- **Standardization:** Normalization helps in standardizing the data in the database. By organizing the data into tables and defining relationships between them, normalization helps in ensuring that the data is stored in a consistent and uniform manner.

9.1. Normal Forms & Dependency

Important Points Regarding Normal Forms in DBMS

- **First Normal Form (1NF):** This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
- **Second Normal Form (2NF):** 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.

- **Third Normal Form (3NF):** 3NF builds on 2NF by requiring that all non-key attributes be independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
- **Boyce-Codd Normal Form (BCNF):** BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.
- **Fourth Normal Form (4NF):** 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies.
- **Fifth Normal Form (5NF):** 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity.

Normal forms help to reduce data redundancy, increase data consistency, and improve database performance. However, higher levels of normalization can lead to more complex database designs and queries. It is important to strike a balance between normalization and practicality when designing a database.

First, Second and Third Normal Form

❖ First Normal Form

If a relation contains composite or multi-valued attribute, it violates first normal form, or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- **Example 1 –** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	INDIA
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

❖ Second Normal Form

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e.**,** no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table. **Partial Dependency –** If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

- **Example 1 –** Consider table-3 as following below.

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

{Note that, there are many courses having the same course fee} Here, COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO; COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO; COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO; Hence, COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ; But, COURSE_NO -> COURSE_FEE, i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF. To convert the above relation to 2NF, we need to split the table into two tables such as :

Table 1: *STUD_NO, COURSE_NO* ***Table 2****: COURSE_NO, COURSE_FEE*

Table 1

STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
4	C3
3	C1

Table 2

COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

NOTE: 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we don't need to store its Fee as 1000 for all the 100 records, instead, once we can store it in the second table as the course fee for C1 is 1000.

❖ Third Normal Form

A relation is said to be in third normal form, if we did not have any transitive dependency for non-prime attributes. The basic condition with the Third Normal Form is that, the relation must be in Second Normal Form.

Below mentioned is the basic condition that must be hold in the non-trivial functional dependency X -> Y:

- X is a Super Key.
- Y is a Prime Attribute (this means that element of Y is some part of Candidate Key).

Boyce-Codd Normal Form (BCNF), Fourth and Fifth Normal Form

❖ BCNF (Boyce-Codd Normal Form)

BCNF (Boyce-Codd Normal Form) is just an advanced version of Third Normal Form. Here we have some additional rules than Third Normal Form. The basic condition for any relation to be in BCNF is that it must be in Third Normal Form.

We must focus on some basic rules that are for BCNF:

- Table must be in Third Normal Form.
    - In relation X->Y, X must be a superkey in a relation.

❖ Fourth Normal Form

Fourth Normal Form contains no non-trivial multivalued dependency except candidate key. The basic condition with Fourth Normal Form is that the relation must be in BCNF.

The basic rules are mentioned below.

- It must be in BCNF.
- It does not have any multi-valued dependency.

❖ Fifth Normal Form

Fifth Normal Form is also called as Projected Normal Form. The basic conditions of Fifth Normal Form are mentioned below.

Relation must be in Fourth Normal Form.

The relation must not be further non loss decomposed.

❖ How To Find the Highest normal Form of a relation

Steps to find the highest normal form of relation:

Step 1. Find all possible candidate keys of the relation.

Step 2. Divide all attributes into two categories: prime attributes and non-prime attributes.

Step 3. Check for 1^{st} normal form then 2^{nd} and so on. If it fails to satisfy the n^{th} normal form condition, the highest normal form will be n-1.

Example 1. Find the highest normal form of a relation R(A,B,C,D,E) with FD set {A->D, B->A, BC->D, AC->BE}

Step 1. As we can see, $(AC)^+$ ={A, C, B, E, D} but none of its subsets can determine all attributes of relation, So AC will be the candidate key. A can be derived from B, so we can replace A in AC with B. So BC will also be a candidate key. So, there will be two candidate keys {AC, BC}.
Step 2. The prime attribute is that attribute which is part of candidate key {A, B, C} in this example and others will be non-prime {D, E} in this example.
Step 3. The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attributes.
The relation is not in the 2nd Normal form because A->D is partial dependency (A which is a subset of candidate key AC is determining non-prime attribute D) and the 2nd normal form does not allow partial dependency.
So, the highest normal form will be the 1st Normal Form.

Function Dependency

A functional dependency A->B in a relation holds if two tuples having the same value of attribute A also have the same value for attribute B. For Example, in relation to STUDENT shown in Table 1, Functional Dependencies STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE **hold** but STUD_NAME->STUD_STATE **do not hold.**

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

Advantages of Functional Dependencies

- Through the identification and removal of redundant or unneeded data, they aid in the reduction of data redundancy in databases.
- By guaranteeing that data is correct and consistent throughout the database, they enhance data integrity.
- They make it simpler to add, edit, and remove data, which helps with database management.

Disadvantages of Functional Dependencies

- The process of identifying functional dependencies can be time-consuming and complex, especially in large databases with many tables and relationships.

- Overly restrictive functional dependencies can result in slow query performance or data inconsistencies, as data that should be related may not be properly linked.
- Functional dependencies do not consider the semantic meaning of data and may not always reflect the true relationships between data elements.

9.2. Attribute Clouser and Candidate Key

Attribute Closure

Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.

How to find attribute closure of an attribute set?

To find attribute closure of an attribute set:

- Add elements of attribute set to the result set.
- Recursively add elements to the result set which can be functionally determined from the elements of the result set.

Using FD set of table 1, attribute closure can be determined as:

(STUD_NO) + = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}

(STUD_STATE) + = {STUD_STATE, STUD_COUNTRY}

Advantages of Attribute Closure

- Attribute closures help to identify all possible attributes that can be derived from a set of given attributes.
- They facilitate database design by identifying relationships between attributes and tables, which can help to optimize query performance.
- They ensure data consistency by identifying all possible combinations of attributes that can exist in the database.

Disadvantages of Attribute Closure

- The process of calculating attribute closures can be computationally expensive, especially for large datasets.
- Attribute closures can become too complex to manage, especially as the number of attributes and tables in a database grows.
- Attribute closures do not take into account the semantic meaning of data and may not always accurately reflect the relationships between data elements.

How to Find Candidate Keys and Super Keys Using Attribute Closure?

- If attribute closure of an attribute set contains all attributes of relation, the attribute set will be super key of the relation.
- If no subset of this attribute set can functionally determine all attributes of the relation, the set will be candidate key as well. For Example, using FD set of tables 1,

(STUD_NO, STUD_NAME) + = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}
(STUD_NO)+ = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}
(STUD_NO, STUD_NAME) will be super key but not candidate key because its subset (STUD_NO)+ is equal to all attributes of the relation. So, STUD_NO will be a candidate key.

Prime and Non-Prime Attributes

Attributes which are parts of any candidate key of relation are called as prime attribute, others are non-prime attributes. For Example, STUD_NO in STUDENT relation is prime attribute, others are non-prime attribute.

Candidate Key

Candidate keys play an essential role in Database Management Systems (DBMS) by ensuring data integrity and efficient retrieval. A candidate key refers to a set of attributes that can uniquely identify each record in a table. In this article, we will explore the concept of candidate keys, their significance in DBMS, and their crucial role in optimizing databases.

What is a Candidate Key?

A candidate key is a minimal set of attributes that uniquely identifies each tuple within a table. In other words, there should not be any two rows in a table that can have the same values for the columns that are the part of candidate key. It is very important for establishing relationships between tables and maintaining data integrity. Candidate keys play a pivotal role in database normalization as they help us to eliminate data redundancy and anomalies.

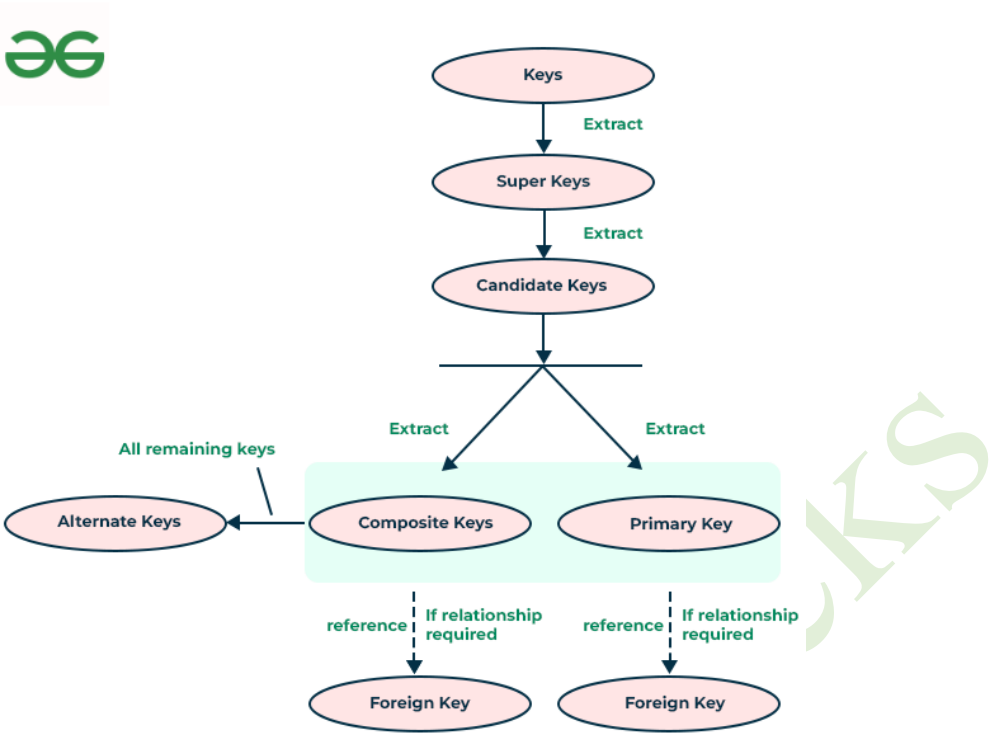


Figure 25: Key in DBMS

Example of Candidate Key

Let us try to understand, the concept of the candidate key with an example of a student table.

Student_id	Roll_no	Name	Mobile_no	Email_id
A1	1	Dipak	9120	a@gmail.cpm
A2	2	Raja	8732	b@gmail.com
A3	3	Dipak	8344	c@gmail.com

In this table, each student can uniquely identify by any of the following attribute: Student_id, Roll_no, Mobile_no, Email_id. So let primary key is Student_id and Candidate keys are Student_id**,** Roll_no**,** Mobile_no**,** Email_id.

9.3. Lossless Decomposition and Dependency Preserving Decomposition

What is Lossless Decomposition?

Lossless join decomposition is a decomposition of a relation R into relations R1, and R2 such that if we perform a natural join of relation R1 and R2, it will return the original relation R. This is effective in removing redundancy from databases while preserving the original data.

In other words by lossless decomposition, it becomes feasible to reconstruct the relation R from decomposed tables R1 and R2 by using Joins.

0 seconds of 17 seconds Volume 0%

Only **1NF,2NF,3NF,** and **BCNF** are valid for lossless join decomposition.
In Lossless Decomposition, we select the common attribute and the criteria for selecting a common attribute is that the common attribute must be a candidate key or super key in either relation R1, R2, or both.
Decomposition of a relation R into R1 and R2 is a lossless-join decomposition if at least one of the following functional dependencies is in F+ (Closure of functional dependencies)

Example of Lossless Decomposition
— Employee (Employee_Id, Ename, Salary, Department_Id, Dname)
Can be decomposed using lossless decomposition as,
— Employee_desc (Employee_Id, Ename, Salary, Department_Id)
— Department_desc (Department_Id, Dname)
Alternatively the lossy decomposition would be as joining these tables is not possible so not possible to get back original data.
– Employee_desc (Employee_Id, Ename, Salary)
– Department_desc (Department_Id, Dname)
$R1 \cap R2 \rightarrow R1$
OR
$R1 \cap R2 \rightarrow R2$
In a database management system (DBMS), a lossless decomposition is a process of decomposing a relation schema into multiple relations in such a way that it preserves the information contained in the original relation. Specifically, a lossless decomposition is one in which the original relation can be reconstructed by joining the decomposed relations.
To achieve lossless decomposition, a set of conditions known as Armstrong's axioms can be used. These conditions ensure that the decomposed relations will retain all the information present in the original relation. Specifically, the two most important axioms for lossless decomposition are the reflexivity and the decomposition axiom.

The reflexivity axiom states that if a set of attributes is a subset of another set of attributes, then the larger set of attributes can be inferred from the smaller set. The decomposition axiom states that if a relation R can be decomposed into two relations R1 and R2, then the original relation R can be reconstructed by taking the natural join of R1 and R2.
There are several algorithms available for performing lossless decomposition in DBMS, such as the BCNF (Boyce-Codd Normal Form) decomposition and the 3NF (Third Normal Form) decomposition. These algorithms use a set of rules to decompose a relation into multiple relations while ensuring that the original relation can be reconstructed without any loss of information.

Advantages of Lossless Decomposition

- **Reduced Data Redundancy:** Lossless decomposition helps in reducing the data redundancy that exists in the original relation. This helps in improving the efficiency of the database system by reducing storage requirements and improving query performance.
- **Maintenance and Updates:** Lossless decomposition makes it easier to maintain and update the database since it allows for more granular control over the data.
- **Improved Data Integrity:** Decomposing a relation into smaller relations can help to improve data integrity by ensuring that each relation contains only data that is relevant to that relation. This can help to reduce data inconsistencies and errors.
- **Improved Flexibility:** Lossless decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Lossless Decomposition

- **Increased Complexity:** Lossless decomposition can increase the complexity of the database system, making it harder to understand and manage.
- **Increased Processing Overhead:** The process of decomposing a relation into smaller relations can result in increased processing overhead. This can lead to slower query performance and reduced efficiency.
- **Join Operations:** Lossless decomposition may require additional join operations to retrieve data from the decomposed relations. This can also result in slower query performance.
- **Costly:** Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.

Dependency Prevention Decomposition

Dependency Preservation

A Decomposition D = {R1, R2, R3…Rn } of R is dependency preserving wrt a set F of Functional dependency if

(F1 ? F2 ? … ? Fm) + = F+.

Consider a relation R

R ---> F {...with some functional dependency (FD)....}

R is decomposed or divided into R1 with FD { f1 } and R2 with { f2 }, then there can be three cases:

f1 U f2 = F -----> Decomposition is dependency preserving.

f1 U f2 is a subset of F -----> Not Dependency preserving.

f1 U f2 is a super set of F -----> This case is not possible.

Problem:
Let a relation R (A, B, C, D) and functional dependency {AB –> C, C –> D, D –> A}. Relation R is decomposed into R1(A, B, C) and R2(C, D). Check whether decomposition is dependency preserving or not.

Solution:
0 seconds of 17 seconds Volume 0%
R1(A, B, C) and R2(C, D)

Let us find closure of F1 and F2

To find closure of F1, consider all combination of

ABC. i.e., find closure of A, B, C, AB, BC and AC

Note ABC is not considered as it is always ABC

closure(A) = { A } // Trivial

closure(B) = { B } // Trivial

closure(C) = {C, A, D} but D can't be in closure as D is not present R1.

= {C, A}

C--> A // Removing C from right side as it is trivial attribute

closure(AB) = {A, B, C, D} = {A, B, C}

AB --> C // Removing AB from right side as these are trivial attributes

closure(BC) = {B, C, D, A} = {A, B, C}

BC --> A // Removing BC from right side as these are trivial attributes

closure(AC) = {A, C, D}

NULL SET

F1 {C--> A, AB --> C, BC --> A}.

Similarly F2 { C--> D }

In the original Relation Dependency { AB --> C , C --> D , D --> A}.

AB --> C is present in F1.

C --> D is present in F2.

D --> A is not preserved.

F1 U F2 is a subset of F. So **given decomposition is not dependency preserving**.

Lossless Join Decomposition
If we decompose a relation R into relations R1 and R2,
Decomposition is lossy if R1 ⋈ R2 ⊃ R
Decomposition is lossless if R1 ⋈ R2 = R

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

i) The Union of Attributes of R1 and R2 must be equal to the attribute of R. Each attribute of R must be either in R1 or in R2.

0 seconds of 17 secondsVolume 0%

Att(R1) U Att(R2) = Att(R)

ii) The intersection of Attributes of R1 and R2 must not be NULL.

Att(R1) ∩ Att(R2) ≠ Φ

iii) The common attribute must be a key for at least one relation (R1 or R2)

Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)

Dependency Preserving Decomposition
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 a combination of functional dependency of R1 and R2. For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of R1(ABC).

Advantages of Lossless Join and Dependency Preserving Decomposition

- **Improved Data Integrity:** Lossless join and dependency preserving decomposition help to maintain the data integrity of the original relation by ensuring that all dependencies are preserved.
- **Reduced Data Redundancy:** These techniques help to reduce data redundancy by breaking down a relation into smaller, more manageable relations.
- **Improved Query Performance:** By breaking down a relation into smaller, more focused relations, query performance can be improved.
- **Easier Maintenance and Updates:** The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.
- **Better Flexibility:** Lossless join and dependency preserving decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Lossless Join and Dependency Preserving Decomposition

- **Increased Complexity:** Lossless join and dependency-preserving decomposition can increase the complexity of the database system, making it harder to understand and manage.
- **Costly:** Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.
- **Reduced Performance:** Although query performance can be improved in some cases, in others, lossless join and dependency-preserving decomposition can result in reduced query performance due to the need for additional join operations.
- **Limited Scalability:** These techniques may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

9.4. Equivalence of Function Dependencies

For understanding the equivalence of Functional Dependencies Sets (FD sets), the basic idea about Attribute Closure is given in this article Given a Relation with different FD sets for that relation, we have to find out whether one FD set is a subset of another or both are equal.

How To Find the Relationship Between Two Functional Dependency Sets?

Let FD1 and FD2 be two FD sets for a relation R.

i) If all FDs of FD1 can be derived from FDs present in FD2, we can say that FD2 ⊃ FD1.

ii) If all FDs of FD2 can be derived from FDs present in FD1, we can say that FD1 ⊃ FD2.

iii) If 1 and 2 both are true, FD1=FD2.

Why We Need to Compare Functional Dependencies?

Suppose in the designing process we convert the ER diagram to a relational model and this task is given to two different engineers.

Now those two engineers give two different sets of functional dependencies. So, being an administrator, we need to ensure that we must have a good set of Functional Dependencies. To ensure this we require to study the equivalence of Functional Dependencies.

Advantages

- It can help to identify redundant functional dependencies, which can be eliminated to reduce data redundancy and improve database performance.

- It can help to optimize database design by identifying equivalent sets of functional dependencies that can be used interchangeably.
- It can ensure data consistency by identifying all possible combinations of attributes that can exist in the database.

Disadvantages

- The process of determining the equivalence of functional dependencies can be computationally expensive, especially for large datasets.
- The process may require testing multiple candidates sets of functional dependencies, which can be time-consuming and complex.
- The equivalence of functional dependencies may not always accurately reflect the semantic meaning of data and may not always reflect the true relationships between data elements.

Question.1 Let us take an example to show the relationship between two FD sets. A relation R(A,B,C,D) having two FD sets FD1 = {A->B, B->C, AB->D} and FD2 = {A->B, B->C, A->C, A->D}

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

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

As all FDs in set FD1 also hold in set FD2, **FD2 ⊃ FD1** is true.

9.5. Canonical Cover of Functional Dependencies

Whenever a user updates the database, the system must check whether any of the functional dependencies are getting violated in this process. If there is a violation of dependencies in the new database state, the system must roll back. Working with a huge set of functional dependencies can cause unnecessary added computational time. This is where the canonical cover comes into play. 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.

An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies

Canonical Cover

In DBMS, a canonical cover is a set of functional dependencies that is equivalent to a given set of functional dependencies but is minimal in terms of the number of dependencies. The process of finding the canonical cover of a set of functional dependencies involves three main steps:

- **Reduction:** The first step is to reduce the original set of functional dependencies to an equivalent set that has the same closure as the original set, but with fewer dependencies. This is done by removing redundant dependencies and combining dependencies that have common attributes on the left-hand side.
- **Elimination:** The second step is to eliminate any extraneous attributes from the left-hand side of the dependencies. An attribute is considered extraneous if it can be removed from the left-hand side without changing the closure of the dependencies.
- **Minimization:** The final step is to minimize the number of dependencies by removing any dependencies that are implied by other dependencies in the set.
- To illustrate the process, let's consider a set of functional dependencies: A -> BC, B -> C, and AB -> C. Here are the steps to find the canonical cover:
- **Reduction:** We can reduce the set by removing the redundant dependency B -> C and combining the two remaining dependencies into one: A -> B, A -> C.
- **Elimination:** We can eliminate the extraneous attribute B from the dependency A -> B, resulting in A -> C.
- **Minimization:** We can minimize the set by removing the dependency AB -> C, which is implied by A -> C.
- The resulting canonical cover for the original set of functional dependencies is A -> C.

A canonical cover Fc of a set of functional dependencies F such that all the following properties are satisfied:

- F logically implies all dependencies in Fc.
- Fc logically implies all dependencies in F.
- No functional dependency in Fc contains an extraneous attribute.
- Each left side of a functional dependency in Fc is unique. That is, there are no two dependencies $\alpha1 \rightarrow \beta1$ and $\alpha2 \rightarrow \beta2$ in such that $\alpha1 \rightarrow \alpha2$.

The canonical cover is useful because it provides a simplified representation of the original set of functional dependencies that can be used to determine the key, superkey, and candidate key for a relation, as well as to check for normalization violations and perform other database design tasks.

How to Find Canonical Cover?

Below mentioned is the algorithm to compute canonical cover for set F.

i) **Repeat**
Use the union rule to replace any dependencies in $\alpha 1 \rightarrow \beta 1$ and $\alpha 2 \rightarrow \beta 2$ with $\alpha 1 \rightarrow \beta 1\beta 2$

ii) Find a functional dependency $\alpha \rightarrow \beta$ with an extraneous attribute either in α or in β.

iii) If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$. **until F does not change.**

Example 1:

Consider the following set *F* of functional dependencies:
F= {A → BC, B → C A → B, AB → C}. Below mentioned are the steps to find the canonical cover of the functional dependency given above.

Step 1: There are two functional dependencies with the same attributes on the left: A → BC, A → B. These two can be combined to get A → BC. Now, the revised set F becomes F= {A → BC, B → C, AB → C}.

Step 2: There is an extraneous attribute in AB → C because even after removing AB → C from the set F, we get the same closures. This is because B → C is already a part of F. Now, the revised set F becomes: F= {A → BC, B → C}

Step 3: C is an extraneous attribute in A → BC, also A → B is logically implied by A → B and B → C (by transitivity). F= {A → B B → C}

Step 4: After this step, F does not change anymore. So, hence the required canonical cover is, Fc = {A → B, B → C}

Features of the Canonical Cover

- **Minimal:** The canonical cover is the smallest set of dependencies that can be derived from a given set of dependencies, i.e., it has the minimum number of dependencies required to represent the same set of constraints.
- **Lossless:** The canonical cover preserves all the functional dependencies of the original set of dependencies, i.e., it does not lose any information.
- **Unique:** The canonical cover is unique, i.e., there is only one canonical cover for a given set of dependencies.
- **Deterministic:** The canonical cover is deterministic, i.e., it does not contain any redundant or extraneous dependencies.
- **Reduces data redundancy:** The canonical cover helps to reduce data redundancy by eliminating unnecessary dependencies that can be inferred from other dependencies.
- **Improves query performance:** The canonical cover helps to improve query performance by reducing the number of joins and redundant data in the database.

Structured Query language (SQL)

10. SQL Overview

Structured Query Language is a standard Database language that is used to create, maintain, and retrieve the relational database. In this article, we will discuss this in detail about SQL. Following are some interesting facts about SQL. Let us focus on that.

SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc.) in capital letters and use user-defined things (like table name, column name, etc.) in small letters.

We can write comments in SQL using "–" (double hyphen) at the beginning of any line. SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgrad, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc. do not use SQL.

Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So, we may encounter queries that work in SQL Server but do not work in MySQL.

What is Relational Database?

A relational database means the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which
stores **ROLL_NO**, **NAME**, **ADDRESS**, **PHONE,** and **AGE** of students.

STUDENT Table

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

Important Terminologies

These are some important terminologies that are used in terms of relation.

- **Attribute:** Attributes are the properties that define a relation.
  e.g.; **ROLL_NO**, **NAME** etc.
- **Tuple:** Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18

- **Degree:** The number of attributes in the relation is known as degree of the relation. The **STUDENT** relation defined above has degree 5.
- **Cardinality:** The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.

- **Column:** Column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from relation STUDENT.

ROLL_NO
1
2
3
4

How Queries can be Categorized in Relational Database?

The queries to deal with relational database can be categorized as:

- **Data Definition Language:** It is used to define the structure of the database. e.g., CREATE TABLE, ADD COLUMN, DROP COLUMN and so on.
- **Data Manipulation Language:** It is used to manipulate data in the relations. e.g., INSERT, DELETE, UPDATE and so on.
- **Data Query Language:** It is used to extract the data from the relations. e.g., SELECT So first we will consider the Data Query Language. A generic query to retrieve data from a relational database.

i) **SELECT** [**DISTINCT**] Attribute_List **FROM** R1, R2….RM
ii) [**WHERE** condition]
iii) [**GROUP BY** (Attributes)[**HAVING** condition]]
iv) [**ORDER BY**(Attributes)[**DESC**]];

Different Query Combinations

Case 1: If we want to retrieve attributes **ROLL_NO** and **Name** of all students, the query will be:

SELECT ROLL_NO, NAME **FROM** STUDENT;

ROLL_NO	**NAME**
1	RAM
2	RAMESH
3	SUJIT
4	SURESH

Case 2: If we want to retrieve **ROLL_NO** and **NAME** of the students whose **ROLL_NO** is greater than 2, the query will be:

SELECT ROLL_NO, NAME **FROM** STUDENT
WHERE ROLL_NO>2;

ROLL_NO	**NAME**
3	SUJIT
4	SURESH

CASE 3: If we want to retrieve all attributes of students, we can write * in place of writing all attributes as:
SELECT * FROM STUDENT
WHERE ROLL_NO>2;

ROLL_NO	NAME	ADDRESS	PHONE	AGE
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

CASE 4: If we want to represent the relation in ascending order by **AGE**, we can use ORDER BY clause as:
0 seconds of 0 secondsVolume 0%

SELECT * FROM STUDENT **ORDER BY** AGE;

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
4	SURESH	DELHI	9156768971	18
3	SUJIT	ROHTAK	9156253131	20

Note:
ORDER BY AGE is equivalent to ORDER BY AGE ASC.
If we want to retrieve the results in descending order of AGE, we can use ORDER BY AGE DESC.

CASE 5: If we want to retrieve distinct values of an attribute or group of attributes, DISTINCT is used as in:
SELECT DISTINCT ADDRESS **FROM** STUDENT;

ADDRESS
DELHI
GURGAON
ROHTAK

If DISTINCT is not used, DELHI will be repeated twice in result set. Before understanding GROUP BY and HAVING, we need to understand aggregations functions in SQL.

Aggregation Functions
Aggregation functions are used to perform mathematical operations on data values of a relation. Some of the common aggregation functions used in SQL are:

- **COUNT:** Count function is used to count the number of rows in a relation. e.g.,

SELECT COUNT (PHONE) **FROM** STUDENT;

COUNT(PHONE)
4

- **SUM:** SUM function is used to add the values of an attribute in a relation. e.g.,

SELECT SUM(AGE) FROM STUDENT;

SUM (AGE)
74

In the same way, MIN, MAX and AVG can be used. As we have seen above, all aggregation functions return only 1 row. **AVERAGE:** It gives the average values of the tuples. It is also defined as sum divided by count values.

Syntax:

AVG (attributename)

OR

SUM (attributename)/COUNT (attributename)

The above-mentioned syntax also retrieves the average value of tuples.

- **MAXIMUM:** It extracts the maximum value among the set of tuples.

Syntax:

MAX (attributename)

- **MINIMUM:** It extracts the minimum value amongst the set of all the tuples.

Syntax:

MIN (attributename)

- **GROUP BY:** Group by is used to group the tuples of a relation based on an attribute or group of attributes. It is always combined with aggregation function which is computed on group. e.g.,

SELECT ADDRESS, **SUM**(AGE) **FROM** STUDENT
GROUP BY (ADDRESS);

In this query, SUM(**AGE**) will be computed but not for entire table but for each address. i.e., sum of AGE for address DELHI (18+18=36) and similarly for other address as well. The output is:

ADDRESS	SUM(AGE)
DELHI	36
GURGAON	18
ROHTAK	20

If we try to execute the query given below, it will result in error because although we have computed SUM(AGE) for each address, there are more than 1 ROLL_NO for each address we have grouped. So, it cannot be displayed in result set. We need to use aggregate functions on columns after SELECT statement to make sense of the resulting set whenever we are using GROUP BY.

SELECT ROLL_NO, ADDRESS, **SUM**(AGE) **FROM** STUDENT
GROUP BY (ADDRESS);

NOTE: *An attribute which is not a part of GROUP BY clause can't be used for selection.*
Any attribute which is part of GROUP BY CLAUSE can be used for selection but it is not mandatory.
But we could use attributes which are not a part of the GROUP BY clause in an aggregate function.

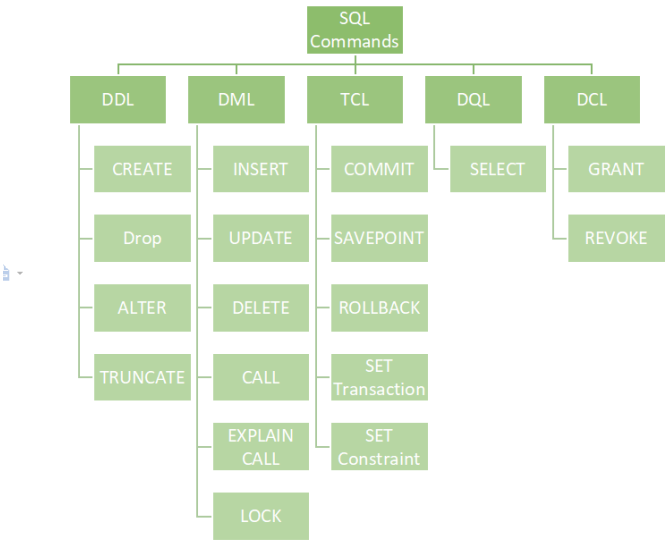
10.1. SQL Commands

Looking for one place where you can find all the SQL commands or SQL sublanguage commands like DDL, DQL, DML, DCL, and TCL, then bookmark this article. In this write-up, you will explore all the Structured Query Language (SQL) commands with accurate syntax.

Structured Query Language (SQL), as we all know, is the database language by the use of which we can perform certain operations on the existing database, and also, we can use this language to create a database. SQL uses certain commands like CREATE, DROP, INSERT, etc. to carry out the required tasks.
SQL commands are like instructions to a table. It is used to interact with the database with some operations. It is also used to perform specific tasks, functions, and queries of data. SQL can perform various tasks like creating a table, adding data to tables, dropping the table, modifying the table, set permission for users.

These SQL commands are mainly categorized into five categories:

- DDL – Data Definition Language
- DQL – Data Query Language
- DML – Data Manipulation Language
- DCL – Data Control Language
- TCL – Transaction Control Language



i) DDL (Data Definition Language)

DDL or Data Definition Language 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. DDL is a set of SQL commands used to create, modify, and delete database structures but not data. These commands are normally not used by a general user, who should be accessing the database via an application.
List of DDL commands:

- **CREATE**: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- **DROP**: This command is used to delete objects from the database.
- **ALTER:** This is used to alter the structure of the database.
- **TRUNCATE:** This is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT**: This is used to add comments to the data dictionary.
- **RENAME:** This is used to rename an object existing in the database.

ii) DQL (Data Query Language)

DQL statements are used for performing queries on the data within schema objects. The purpose of the DQL Command is to get some schema relation based on the query passed to it. We can define DQL as follows it is a component of SQL statement that allows getting data from the database and imposing order upon it. It includes the SELECT statement. This command allows getting the data out of the database to perform operations with it. When a SELECT is fired against a table or tables the result is compiled into a further temporary table, which is displayed or perhaps received by the program i.e., front-end.
List of DQL:

- **SELECT:** It is used to retrieve data from the database.

iii) DML (Data Manipulation Language)

The SQL commands that deal with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

List of DML commands:

- **INSERT**: It is used to insert data into a table.
- **UPDATE:** It is used to update existing data within a table.
- **DELETE**: It is used to delete records from a database table.
- **LOCK:** Table control concurrency.
- **CALL:** Call a PL/SQL or JAVA subprogram.
- **EXPLAIN PLAN:** It describes the access path to data.

iv) DCL (Data Control Language)

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

- **GRANT:** This command gives users access privileges to the database.

Syntax:*GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOT HER_USER;*

- **REVOKE:** This command withdraws the user's access privileges given by using the GRANT command.

Syntax:

REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

v) TCL (Transaction Control Language)

Transactions group a set of tasks into a single execution unit. Each transaction begins with a specific task and ends when all the tasks in the group are successfully completed. If any of the tasks fail, the transaction fails. Therefore, a transaction has only two results: success or failure. Hence, the following TCL commands are used to control the execution of a transaction:

BEGIN: Opens a Transaction.

COMMIT: Commits a Transaction.

Syntax:

COMMIT;

ROLLBACK: Rollbacks a transaction in case of any error occurs.

Syntax:

ROLLBACK;

SAVEPOINT: Sets a save point within a transaction.

Syntax: *SAVEPOINT SAVEPOINT_NAME;*

10.2. SQL | Join

SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- NATURAL JOIN

Consider the two tables below as follows:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	xxxxxxxxxx	18
2	PRATIK	BIHAR	xxxxxxxxxx	19
3	RIYANKA	SILIGURI	xxxxxxxxxx	20
4	DEEP	RAMNAGAR	xxxxxxxxxx	18
5	SAPTARHI	KOLKATA	xxxxxxxxxx	19
6	DHANRAJ	BARABAJAR	xxxxxxxxxx	20
7	ROHIT	BALURGHAT	xxxxxxxxxx	18
8	NIRAJ	ALIPUR	xxxxxxxxxx	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

The simplest Join is INNER JOIN.

i) INNER JOIN

The INNER JOIN keyword selects all rows from both the tables if the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e., value of the common field will be the same.

Syntax:

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

table1: First table.
table2: Second table
matching_column: Column common to both the tables.

***Note**: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*

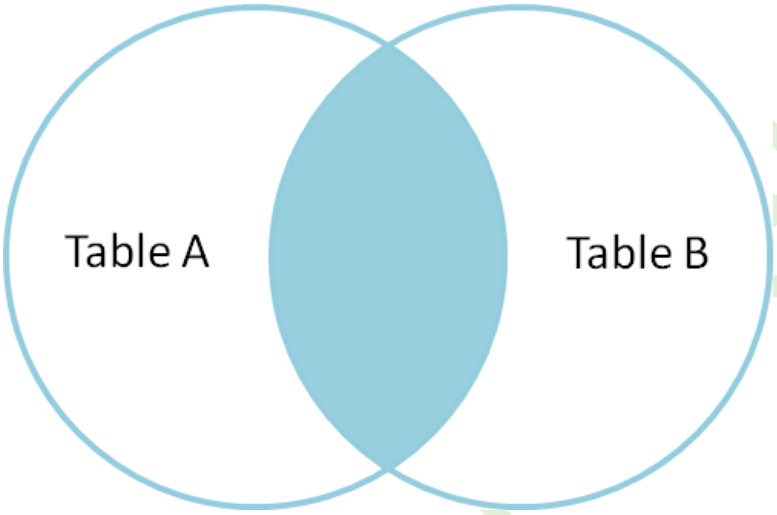


Figure 26: Venn Diagram Inner Join

Example Queries (INNER JOIN)
This query will show the names and age of students enrolled in different courses.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM
Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	Age
1	Harsh	18
2	Pratik	19
2	Riyanka	20
3	Deep	18
1	Saptarhi	19

ii) LEFT JOIN

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.

***Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*

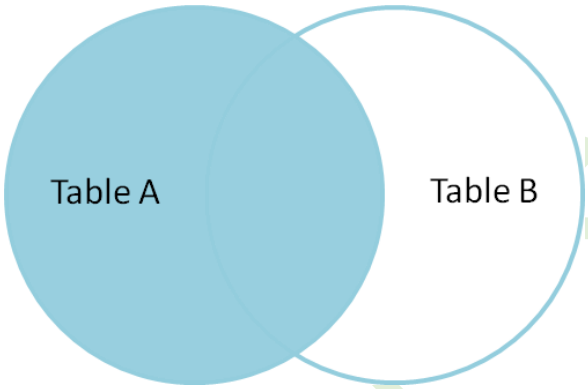


Figure 27: Venn Diagram (Left Join)

Example Queries (LEFT JOIN):
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	**COURSE_ID**
Harsh	1
Pratik	2
Riyanka	2
Deep	3
Saptarhi	1
Dhanraj	NULL
Rohit	NULL
Niraj	NULL

iii) RIGHT JOIN

RIGHT JOIN is like LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
table1: First table.
table2: Second table
matching_column: Column common to both the tables.

***Note**: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*

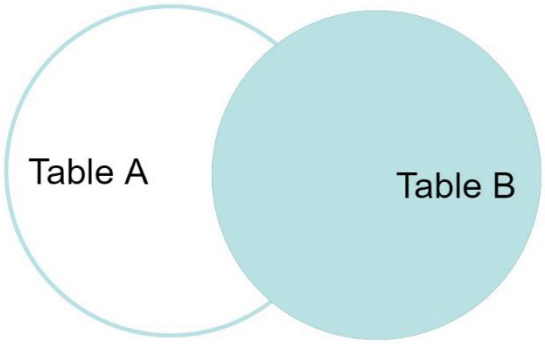


Figure 28: Venn Diagram (Right Join)

Example Queries (RIGHT JOIN):
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
Output:

NAME	COURSE_ID
Harsh	1
Pratik	2
Riyanka	2
Deep	3
Saptarhi	1
NULL	4
NULL	5
NULL	4

iv) FULL JOIN

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.

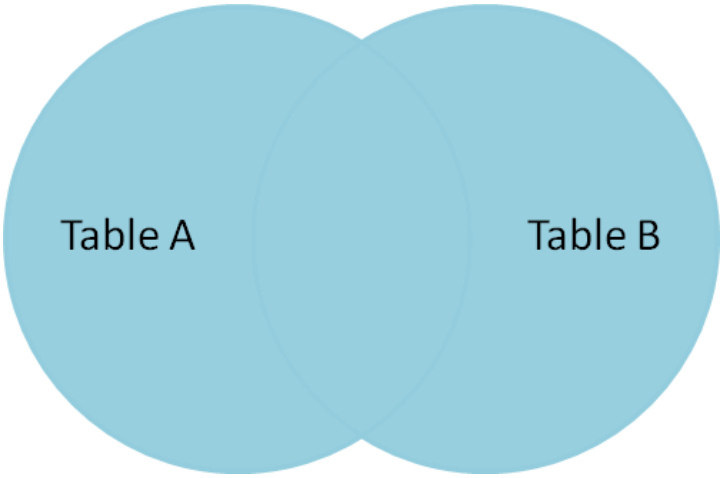


Figure 29: Venn Diagram (Full Join)

Syntax:
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
table1: First table.
table2: Second table
matching_column: Column common to both the tables.

Example Queries (FULL JOIN):
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4
NULL	5
NULL	4

10.3. Clause in SQL

Having vs. Where Clause

The difference between the having and where clause in SQL is that where clause cann*ot* be used with aggregates, but the having clause can.

The **where** clause works on row's data, not on aggregated data. Let us consider below table 'Marks'.

Student	Course	Score
a	c1	40
a	c2	50
b	c3	60
d	c1	70
e	c2	80

Consider the query.

```
SELECT Student, Score FROM Marks WHERE Score >= 40
```

This would select data row by row basis.

The **having** clause works on aggregated data.
For example, output of below query

```
SELECT Student, SUM (SCORE AS total FROM Marks GROUP BY Student
```

Student	Total
a	90
b	60
c	70
d	80

When we apply having in above query, we get

```
SELECT Student, SUM (score) AS total FROM Marks Group BY Student HAVING
```

Student	Total
a	90
c	80

Note: *It is not a predefined rule but in a good number of the SQL queries, we use WHERE prior to GROUP*

BY and HAVING after GROUP BY. The Where clause acts as a **pre filter** where as Having as a **post filter.**

10.4. Database Objects

A **database object** is any defined object in a database that is used to store or reference data. Anything which we make from **create command** is known as Database Object. It can be used to hold and manipulate the data. Some of the examples of database objects are : view, sequence, indexes, etc.

- **Table –** Basic unit of storage; composed rows and columns.
- **View –** Logically represents subsets of data from one or more tables.
- **Sequence –** Generates primary key values.
- **Index –** Improves the performance of some queries.
- **Synonym –** Alternative name for an object

Different database Objects:

i) **Table –** This database object is used to create a table in database.

Syntax:

CREATE TABLE [schema.] table (column datatype [DEFAULT expr][, ...]);

Example:

```
CREATE TABLE dept
        (deptno NUMBER (2),
         dname VARCHAR2(14),
         loc VARCHAR2(13)) ;
```

Output:

DESCRIBE dept;

Name	Null?	Type
DEPT_NO		NUMBER(2)
DNAME		VARCHAR(14)
LOC		VARCHAR(13)

ii) **View –** This database object is used to create a view in database. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.

Syntax:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
                    [(alias[, alias]...)]
                    AS subquery
                    [WITH CHECK OPTION [CONSTRAINT
constraint]]
                    [WITH READ ONLY [CONSTRAINT constraint]];
```

Example:

```
CREATE VIEW salvu50
        AS SELECT employee_id ID_NUMBER, last_name
NAME,
        salary*12 ANN_SALARY
        FROM employees
        WHERE department_id = 50;
```

Output:

```
SELECT *
FROM salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgous	69600
141	Rajs	42000
142	Davies	372000
143	Matos	312000
144	Vargas	300000

iii) Sequence –

This database object is used to create a sequence in database. A sequence is a user created database object that can be shared by multiple users to generate unique integers. A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine.

Syntax:

```
CREATE SEQUENCE sequence
                [INCREMENT BY n]
                [START WITH n]
                [{MAXVALUE n | NOMAXVALUE}]
                [{MINVALUE n | NOMINVALUE}]
                [{CYCLE | NOCYCLE}]
                [{CACHE n | NOCACHE}];
```

Example:

```
CREATE SEQUENCE dept_deptid_seq
                INCREMENT BY 10
                START WITH 120
                MAXVALUE 9999
                NOCACHE
                NOCYCLE;
```

Check if sequence is created by:

```
SELECT sequence_name, min_value, max_value,
                increment_by, last_number
                FROM   user_sequences;
```

iv) Index –

This database object is used to create an index in database. An Oracle server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle server. Once an index is created, no direct activity is required by the user. Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Syntax:

```
CREATE INDEX index
        ON table (column [, column] ...);
```

Example:

```
CREATE INDEX emp_last_name_idx
        ON  employees(last_name);
```

v) Synonym –

This database object is used to create a index in database. It simplify access to objects by creating a synonym (another name for an object). With synonyms, you can Ease referring to a table owned by another user and shorten lengthy object names. To refer to a table owned by another user, you need to prefix the table name with the name of the user who created it followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

PUBLIC: creates a synonym accessible to all users

synonym : is the name of the synonym to be created

object : identifies the object for which the synonym is created

Syntax:

```
CREATE [PUBLIC] SYNONYM synonym FOR object;
```

Example:

```
CREATE SYNONYM d_sum FOR dept_sum_vu;
```

10.5. Indexing

Indexing improves database performance by minimizing the number of disc visits required to fulfil a query. It is a data structure technique used to locate and quickly access data in databases. Several database fields are used to generate indexes. The main key or candidate key of the table is duplicated in the first column, which is the Search key. To speed up data retrieval, the values are also kept in sorted order. It should be highlighted that sorting the data is not required. The second column is the Data Reference or Pointer which contains a set of pointers holding the address of the disk block where that particular key value can be found.

Attributes of 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 a particular data element or set of elements.
- **Insertion Time:** It refers to the time taken to find the appropriate space and insert 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.

In general, there are two types of file organization mechanisms that are followed by the indexing methods to store the data:

Sequential File Organization or Ordered Index File

In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organizations might store the data in a dense or sparse format.

- **Dense Index**

For every search key value in the data file, there is an index record.
This record contains the search key and a reference to the first data record with that search key value.

- **Sparse Index**
    - The index record appears only for a few items in the data file. Each item points to a block as shown.
    - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
    - We start at that record pointed to by the index record and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
    - Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)

Hash File Organization

Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned are determined by a function called a hash function. There are primarily three methods of indexing:

- **Clustered Indexing**: When more than two records are stored in the same file this type of storing is known as cluster indexing. By using cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored in one place and it also gives the frequent joining of more than two tables (records).
  The clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create an index out of them. This method is known as the clustering index. Essentially, records with similar properties are grouped together, and indexes for these groupings are formed.
  Students studying each semester.
- **Primary Indexing:** This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.
- **Non-clustered or Secondary Indexing**: A non-clustered index just tells us where the data lies, i.e., it gives us a list of virtual pointers or references to the location where the data is stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For e.g., the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here (information on each page of the book) is not organized but we have an ordered reference (contents page) to where the data points lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
  It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.

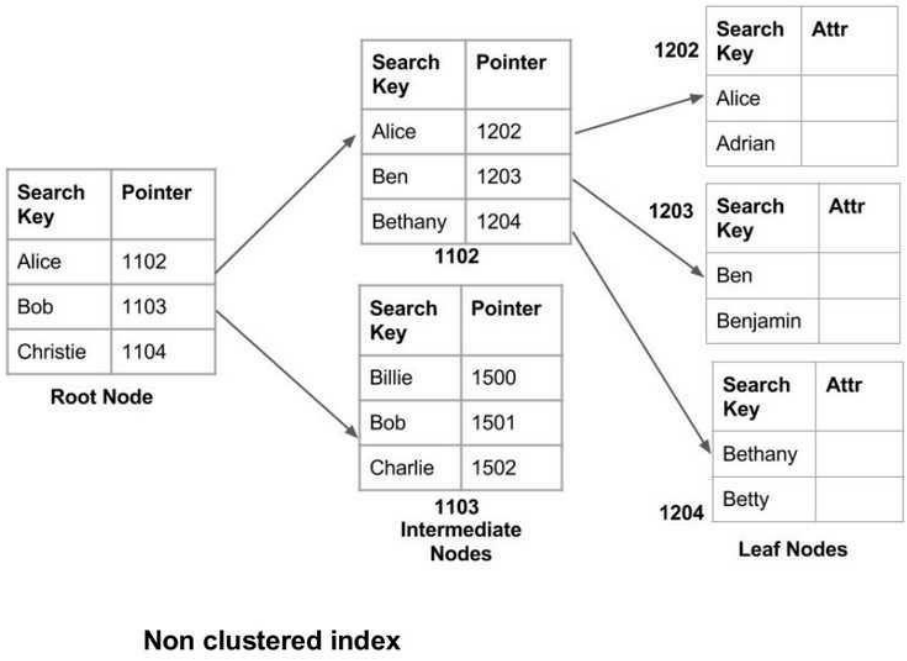


Figure 30: Non-Cluster Indexing

- **Multilevel 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. The multilevel indexing segregates the main block into various smaller blocks so that the same can be stored in a single block. 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.

10.6. SQL Views

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition. In this article we will learn about creating, deleting and updating Views.

Sample Tables:

Table 1. StudentDetails

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

Table 2. StudentMarks

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

CREATING VIEWS

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

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

view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows

Examples:

Creating View from a single table:

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

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

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

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

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

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as,

```
SELECT * FROM StudentNames;
```

Output:

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

Creating View from multiple tables: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

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

LISTING ALL VIEWS IN A DATABASE

We can list View using the SHOW FULL TABLES statement or using the information_schema table. A View can be created from a single table or multiple tables.

Syntax (Using SHOW FULL TABLES):

```
use "database_name";
show full tables where table_type like "%VIEW";
```

Syntax (Using information_schema) :

```
select * from information_schema.views where table_schema =
"database_name";

OR

select table_schema,table_name,view_definition from
information_schema.views where table_schema = "database_name";
```

DELETING VIEWS

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

```
DROP VIEW view_name;
```

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

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

```
DROP VIEW MarksView;
```

UPDATING VIEWS

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

- **i)** The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- **ii)** The SELECT statement should not have the DISTINCT keyword.
- **iii)** The View should have all NOT NULL values.
- **iv)** The view should not be created using nested queries or complex queries.
- **v)** The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

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

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,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.

Syntax:

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

view_name: Name of the View

Example: In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

- If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:

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

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

Syntax:

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

view_name:Name of view from where we want to delete rows

condition: Condition to select rows

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

```
DELETE FROM DetailsView
WHERE NAME="Suresh";
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

WITH CHECK OPTION

The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to an 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.

Example: In the below example we are creating a View SampleView from StudentDetails Table with WITH CHECK OPTION clause.

```
CREATE VIEW SampleView AS
SELECT S_ID, NAME
FROM  StudentDetails
WHERE NAME IS NOT NULL
WITH CHECK OPTION;
```

In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL. For example,though the View is updatable but then also the below query for this View is not valid:

```
INSERT INTO SampleView(S_ID)
VALUES(6);
```

***NOTE**: The default value of NAME column is null.*

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

i) **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.

ii) **Hiding data complexity –** A view can hide the complexity that exists in multiple tables join.

iii) **Simplify commands for the user –** Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.

iv) **Store complex queries –** Views can be used to store complex queries.

v) **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 hide the names of the columns of the base tables.

vi) **Multiple view facility –** Different views can be created on the same table for different users.

10.7. SQL Indexes

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly.
An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data. In this article, we will see how to create, delete, and use the INDEX in the database.

Creating an Index
Syntax

CREATE INDEX *index*
ON TABLE *column;*

where the **index** is the name given to that index **TABLE** is the name of the table on which that index is created, and **column** is the name of that column for which it is applied.

For Multiple Columns
Syntax:

CREATE INDEX *index*
ON TABLE *(column1, column2,.....);*

For Unique Indexes
Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter into the table.
Syntax:

CREATE UNIQUE INDEX *index*
ON TABLE *column;*

When Should Indexes be Created?

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When Should Indexes be Avoided?

- The table is small
- The columns are not often used as a condition in the query
- The column is updated frequently

Removing an Index
Remove an index from the data dictionary by using the **DROP INDEX** command.
Syntax

DROP INDEX *index;*

To drop an index, you must be the owner of the index or have the **DROP ANY INDEX** privilege.

Altering an Index
To modify an existing table's index by rebuilding, or reorganizing the index.
ALTER INDEX *IndexName*
ON *TableName* ***REBUILD****;*

Confirming Indexes
You can check the different indexes present in a particular table given by the user or the server itself and their uniqueness.
Syntax:

SELECT * *from* ***USER_INDEXES****;*

It will show you all the indexes present in the server, in which you can locate your own tables too.

Renaming an Index
You can use the system-stored procedure sp_rename to rename any index in the database.
Syntax:
EXEC sp_rename
index_name,
new_index_name,
N'INDEX';

SQL Server Database
Syntax:
DROP INDEX *TableName.IndexName;*
Why SQL Indexing is Important?
Indexing is an important topic when considering advanced MySQL, although most people know about its definition and usage they don't understand when and where to use it to change the efficiency of our queries or stored procedures by a huge margin.
Here are some scenarios along with their explanation related to Indexing:

- When executing a query on a table having huge data (> 100000 rows), MySQL performs a full table scan which takes much time and the server usually gets timed out. To avoid this always

check the explain option for the query within MySQL which tells us about the state of execution. It shows which columns are being used and whether it will be a threat to huge data. On basis of the columns repeated in a similar order in condition.

- The order of the index is of huge importance as we can use the saitions, we can create an index for them in the same order to maximize the speed of the query. me index in many scenarios. Using only one index we can utilize it in more than one query which different conditions. like for example, in a query, we make a join with a table based on customer_id wards we also join another join based on customer_id and order_date. Then we can simply create a single index by the order of customer_id, order_date which would be used in both cases. This also saves storage.
- We should also be careful to not make an index for each query as creating indexes also take storage and when the amount of data is huge it will create a problem. Therefore, it's important to carefully consider which columns to index based on the needs of your application. In general, it's a good practice to only create indexes on columns that are frequently used in queries and to avoid creating indexes on columns that are rarely used. It's also a good idea to periodically review the indexes in your database and remove any that are no longer needed.
- Indexes can also improve performance when used in conjunction with sorting and grouping operations. For example, if you frequently sort or group data based on a particular column, creating an index on that column can greatly improve performance. The index allows MySQL to quickly access and sort or group the data, rather than having to perform a full table scan.
- In some cases, MySQL may not use an index even if one exists. This can happen if the query optimizer determines that a full table scan is faster than using the index.

SQL Queries on Cluster and Non-Cluster Indexes

Indexing is a procedure that returns your requested data faster from the defined table. Without indexing, the SQL server has to scan the whole table for your data. By indexing, the SQL server will do the exact same thing you do when searching for content in a book by checking the index page. In the same way, a table's index allows us to locate the exact data without scanning the whole table. There are two types of indexing in SQL.

- Clustered index
- Non-clustered index

Clustered Index
A clustered index is the type of indexing that establishes a physical sorting order of rows.
Suppose you have a table *Student_info* which contains *ROLL_NO* as a primary key, then the clustered index which is self-created on that primary key will sort the *Student_info* table as per *ROLL_NO*. A clustered index is like a Dictionary; in the dictionary, the sorting order is alphabetical and there is no separate index page.
Examples:

```
CREATE TABLE Student_info
(
ROLL_NO int(10) primary key,
NAME varchar(20),
DEPARTMENT varchar(20),
);
INSERT INTO Student_info values(1410110405, 'H Agarwal', 'CSE');
INSERT INTO Student_info values(1410110404, 'S Samadder', 'CSE');
INSERT INTO Student_info values(1410110403, 'MD Irfan', 'CSE');

SELECT * FROM Student_info;
```

Output:

ROLL_NO	NAME	DEPARTMENT
1410110403	MD Irfan	CSE
1410110404	S Samadder	CSE
1410110405	H Agarwal	CSE

If we want to create a Clustered index on another column, first we have to remove the primary key, and then we can remove the previous index. Note that defining a column as a primary key makes that column the Clustered Index of that table. To make any other column, the clustered index, first we have to remove the previous one as follows below.

Syntax:

```
//Drop index
drop index table_name.index_name
//Create Clustered index index
create Clustered index IX_table_name_column_name
on table_name (column_name ASC)
```

Note: *We can create only one clustered index in a table.*

Non-Clustered Index

Non-Clustered index is an index structure separate from the data stored in a table that reorders one or more selected columns. The non-clustered index is created to improve the performance of frequently used queries not covered by a clustered index. It's like a textbook; the index page is created separately at the beginning of that book.

Examples:

```
CREATE TABLE Student_info
(
ROLL_NO int(10),
NAME varchar(20),
DEPARTMENT varchar(20),
);
INSERT INTO Student_info values(1410110405, 'H Agarwal', 'CSE');
INSERT INTO Student_info values(1410110404, 'S Samadder', 'CSE');
INSERT INTO Student_info values(1410110403, 'MD Irfan', 'CSE');

SELECT * FROM Student_info;
```

Output:

ROLL_NO	NAME	DEPARTMENT
1410110405	H Agarwal	CSE
1410110404	S Samadder	CSE
1410110403	MD Irfan	CSE

Note: We can create one or more Non_Clustered index in a table.

Syntax:

//Create Non-Clustered index

create NonClustered index IX_table_name_column_name
on table_name (column_name ASC)

Table: Student_info

ROLL_NO	NAME	DEPARTMENT
1410110405	H Agarwal	CSE
1410110404	S Samadder	CSE
1410110403	MD Irfan	CSE

Input: create NonClustered index IX_Student_info_NAME on Student_info (NAME ASC)

Output: Index

NAME	ROW_ADDRESS
H Agarwal	1
MD Irfan	3
S Samadder	2

Clustered vs Non-Clustered Index

- In a table, there can be only one clustered index or one or more than one non_clustered index.
- In Clustered index, there is no separate index storage but in Non-Clustered index, there is separate index storage for the index.
- Clustered index offers faster data access, on the other hand, the Non-clustered index is slower.

Clustered and non-clustered indexes in SQL Server can provide significant performance benefits when querying large tables. Here are some examples of SQL queries and the advantages of using clustered and non-clustered indexes:

SELECT Queries with WHERE Clause

- **Clustered Index:** When a SELECT query with a WHERE clause is executed on a table with a clustered index, SQL Server can use the clustered index to quickly locate the rows that match the WHERE condition. This can be very efficient for large tables, as it allows the database engine to minimize the number of disk reads required to retrieve the desired data.
- **Non-Clustered Index:** If there is no clustered index on the table or the WHERE clause references columns that are not part of the clustered index, SQL Server can use a non-clustered index to find the matching rows. However, this may require additional disk reads if the non-clustered index does not include all the columns required by the query.

UPDATE Queries

- **Clustered Index:** When an UPDATE query is executed on a table with a clustered index, SQL Server can use the index to quickly locate and modify the rows that match the query criteria. This can be very efficient for large tables, as it allows the database engine to minimize the number of disk writes required to modify the data.
- **Non-Clustered Index:** If the UPDATE query references columns that are not part of the clustered index, SQL Server may need to perform additional disk writes to update the non-clustered index as well.

JOIN Queries

- **Clustered Index:** When performing a JOIN operation between two large tables, SQL Server can use the clustered index on the join column(s) to efficiently match the rows from both tables. This can significantly reduce the time required to complete the query.

- **Non-Clustered Index:** If the JOIN operation references columns that are not part of the clustered index, SQL Server can use a non-clustered index to find the matching rows. However, this may require additional disk reads and slow down the query.
- In general, the advantage of using clustered indexes is that they can provide very efficient access to large tables, particularly when querying on the index columns. The advantage of using non-clustered indexes is that they can provide efficient access to columns that are not part of the clustered index, or when querying multiple tables with JOIN operations. However, non-clustered indexes can also require additional disk reads or writes, which can slow down queries. It is important to carefully design and tune indexes based on the specific query patterns and data access patterns of your application.

Advantages of Indexing

- **Improved Query Performance:** Indexing enables faster data retrieval from the database. The database may rapidly discover rows that match a specific value or collection of values by generating an index on a column, minimizing the amount of time it takes to perform a query.
- **Efficient Data Access:** Indexing can enhance data access efficiency by lowering the amount of disk I/O required to retrieve data. The database can maintain the data pages for frequently visited columns in memory by generating an index on those columns, decreasing the requirement to read from disk.
- **Optimized Data Sorting:** Indexing can also improve the performance of sorting operations. By creating an index on the columns used for sorting, the database can avoid sorting the entire table and instead sort only the relevant rows.
- **Consistent Data Performance:** Indexing can assist ensure that the database performs consistently even as the amount of data in the database rises. Without indexing, queries may take longer to run as the number of rows in the table grows, while indexing maintains a roughly consistent speed.
- By ensuring that only unique values are inserted into columns that have been indexed as unique, indexing can also be utilized to ensure the integrity of data. This avoids storing duplicate data in the database, which might lead to issues when performing queries or reports.

Disadvantages of Indexing

- Indexing necessitates more storage space to hold the index data structure, which might increase the total size of the database.
- **Increased database maintenance overhead:** Indexes must be maintained as data is added, destroyed, or modified in the table, which might raise database maintenance overhead.
- Indexing can reduce insert and update performance since the index data structure must be updated each time data is modified.
- **Choosing an index can be difficult:** It can be challenging to choose the right indexes for a specific query or application and may call for a detailed examination of the data and access patterns.

Features of Indexing

- The development of data structures, such as B-trees or hash tables, that provide quick access to certain data items is known as indexing. The data structures themselves are built on the values of the indexed columns, which are utilized to quickly find the data objects.
- The most important columns for indexing columns are selected based on how frequently they are used and the sorts of queries they are subjected to. The cardinality, selectivity, and uniqueness of the indexing columns can be taken into account.
- There are several different index types used by databases, including primary, secondary, clustered, and non-clustered indexes. Based on the needs of the database system, each form of index offers benefits and drawbacks.
- For the database system to function at its best, periodic index maintenance is required. According to changes in the data and usage patterns, maintenance work involves building, updating, and removing indexes.
- Database query optimization involves indexing, which is essential. The query optimizer utilizes the indexes to choose the best execution strategy for a particular query based on the cost of accessing the data and the selectivity of the indexing columns.
- Databases make use of a range of indexing strategies, including covering indexes, index-only scans, and partial indexes. These techniques maximize the utilization of indexes for particular types of queries and data access.
- When non-contiguous data blocks are stored in an index, it can result in index fragmentation, which makes the index less effective. Regular index maintenance, such as defragmentation and reorganization, can decrease fragmentation.

Transaction and Concurrency Control

11. Concurrency Control Overview

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly.
An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data. In this article, we will see how to create, delete, and use the INDEX in the database.

Creating an Index
Syntax

CREATE INDEX *index*
ON TABLE *column;*

where the **index** is the name given to that index **TABLE** is the name of the table on which that index is created, and **column** is the name of that column for which it is applied.

For Multiple Columns
Syntax:

CREATE INDEX *index*
ON TABLE *(column1, column2,.....);*

For Unique Indexes
Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter the table.
Syntax:

CREATE UNIQUE INDEX *index*
ON TABLE *column*

When Should Indexes be Created?

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause **or a join condition.**

When Should Indexes be Avoided?

- The table is small
- The columns are not often used as a condition in the query
- The column is updated frequently

Removing an Index

Remove an index from the data dictionary by using the **DROP INDEX** command.

Syntax

DROP INDEX *index;*

To drop an index, you must be the owner of the index or have the **DROP ANY INDEX** privilege.

Altering an Index

To modify an existing table's index by rebuilding or reorganizing the index.

ALTER INDEX *IndexName*
ON *TableName* ***REBUILD****;*

Confirming Indexes

You can check the different indexes present in a particular table given by the user or the server itself and their uniqueness.

Syntax:

SELECT ** from* ***USER_INDEXES****;*

It will show you all the indexes present in the server, in which you can locate your own tables too.

Renaming an Index

You can use the system-stored procedure sp_rename to rename any index in the database.

Syntax:

EXEC sp_rename
index_name,
new_index_name,
N'INDEX';
SQL Server Database

Syntax:

DROP INDEX *TableName.IndexName;*

Why SQL Indexing is Important?

Indexing is an important topic when considering advanced MySQL, although most people know about its definition and usage they don't understand when and where to use it to change the efficiency of our queries or stored procedures by a huge margin.

Here are some scenarios along with their explanation related to Indexing:

- When executing a query on a table having huge data (> 100000 rows), MySQL performs a full table scan which takes much time and the server usually gets timed out. To avoid this always check the explain option for the query within

MySQL which tells us about the state of execution. It shows which columns are being used and whether it will be a threat to huge data. On basis of the columns repeated in a similar order in condition.

- The order of the index is of huge importance as we can use the saitions, we can create an index for them in the same order to maximize the speed of the query. me index in many scenarios. Using only one index we can utilize it in more than one query which different conditions. like for example, in a query, we make a join with a table based on customer_id wards we also join another join based on customer_id and order_date. Then we can simply create a single index by the order of customer_id, order_date which would be used in both cases. This also saves storage.
- We should also be careful to not make an index for each query as creating indexes also take storage and when the amount of data is huge it will create a problem. Therefore, it's important to carefully consider which columns to index based on the needs of your application. In general, it's a good practice to only create indexes on columns that are frequently used in queries and to avoid creating indexes on columns that are rarely used. It's also a good idea to periodically review the indexes in your database and remove any that are no longer needed.
- Indexes can also improve performance when used in conjunction with sorting and grouping operations. For example, if you frequently sort or group data based on a particular column, creating an index on that column can greatly improve performance. The index allows MySQL to quickly access and sort or group the data, rather than having to perform a full table scan.
- In some cases, MySQL may not use an index even if one exists. This can happen if the query optimizer determines that a full table scan is faster than using the index.

Concurrency Control Problems

There are several problems that arise when numerous transactions are executed simultaneously in a random manner. The database transaction consist of two major operations "Read" and "Write". It is very important to manage these operations in the concurrent execution of the transactions in order to maintain the consistency of the data.

Dirty Read Problem (Write-Read conflict)

Dirty read problem occurs when one transaction updates an item but due to some unconditional events that transaction fails but before the transaction performs rollback, some other transaction reads the updated value. Thus creates an inconsistency in the database. Dirty read problem comes under the scenario of Write-Read conflict between the transactions in the database.

- The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.
- Transaction T1 modifies a database record without committing the changes.
- T2 reads the uncommitted data changed by T1
- T1 performs rollback
- T2 has already read the uncommitted data of T1 which is no longer valid, thus creating inconsistency in the database.

Lost Update Problem

Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.

- T1 reads the value of an item from the database.
- T2 starts and reads the same database item.
- T1 updates the value of that data and performs a commit.
- T2 updates the same data item based on its initial read and performs commit.
- This results in the modification of T1 gets lost by the T2's writes which causes a lost update problem in the database.

Concurrency Control Protocols

Concurrency control protocols are the set of rules which are maintained to solve the concurrency control problems in the database. It ensures that the concurrent transactions can execute properly while maintaining the database consistency. The concurrent execution of a transaction is provided with atomicity, consistency, isolation, durability, and serializability via the concurrency control protocols.

- Locked based concurrency control protocol.
- Timestamp based concurrency control protocol.

Locked based Protocol.

In locked based protocol, each transaction needs to acquire locks before they start accessing or modifying the data items. There are two types of locks used in databases.

- **Shared Lock:** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item, but it cannot modify the data item.
- **Exclusive Lock:** Exclusive lock is also known as the write lock. Exclusive lock allows a transaction to update a data item. Only one transaction can hold the exclusive lock on a data item at a time. While a transaction is holding an exclusive lock on a data item, no other transaction is allowed to acquire a shared/exclusive lock on the same data item.

There are two kind of lock based protocol mostly used in database:

- **Two Phase Locking Protocol:** Two phase locking is a widely used technique which ensures strict ordering of lock acquisition and release. Two phase locking protocol works in two phases.
- **Growing Phase:** In this phase, the transaction starts acquiring locks before performing any modification on the data items. Once a transaction acquires a lock, that lock can not be released until the transaction reaches the end of the execution.
- **Shrinking Phase :** In this phase, the transaction releases all the acquired locks once it performs all the modifications on the data item. Once the transaction starts releasing the locks, it can not acquire any locks further.
- **Strict Two Phase Locking Protocol :** It is almost similar to the two phase locking protocol the only difference is that in two phase locking the transaction can release its locks before it commits, but in case of strict two phase locking the transactions are only allowed to release the locks only when they performs commits.

Timestamp based Protocol

- In this protocol each transaction has a timestamp attached to it. Timestamp is nothing but the time in which a transaction enters into the system.
- The conflicting pairs of operations can be resolved by the timestamp ordering protocol through the utilization of the timestamp values of the transactions. Therefore, guaranteeing that the transactions take place in the correct order.

Advantages of Concurrency

In general, concurrency means, that more than one transaction can work on a system. The advantages of a concurrent system are:

- **Waiting Time:** It means if a process is in a ready state but still the process does not get the system to get execute is called waiting time. So, concurrency leads to less waiting time.
- **Response Time:** The time wasted in getting the response from the cpu for the first time, is called response time. So, concurrency leads to less Response Time.
- **Resource Utilization:** The amount of Resource utilization in a particular system is called Resource Utilization. Multiple transactions can run parallel in a system. So, concurrency leads to more Resource Utilization.
- **Efficiency:** The amount of output produced in comparison to given input is called efficiency. So, Concurrency leads to more Efficiency.

Disadvantages of Concurrency

- **Overhead:** Implementing concurrency control requires additional overhead, such as acquiring and releasing locks on database objects. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.
- **Deadlocks:** Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve, and can result in reduced throughput and increased latency.
- **Reduced concurrency:** Concurrency control can limit the number of users or applications that can access the database simultaneously. This can lead to reduced concurrency and slower performance in systems with high levels of concurrency.
- **Complexity:** Implementing concurrency control can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.
- **Inconsistency:** In some cases, concurrency control can lead to inconsistencies in the database. For example, a transaction that is rolled back may leave the database in an inconsistent state, or a long-running transaction may cause other transactions to wait for extended periods, leading to data staleness and reduced accuracy.

12. ACID Properties in DBMS

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
To maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

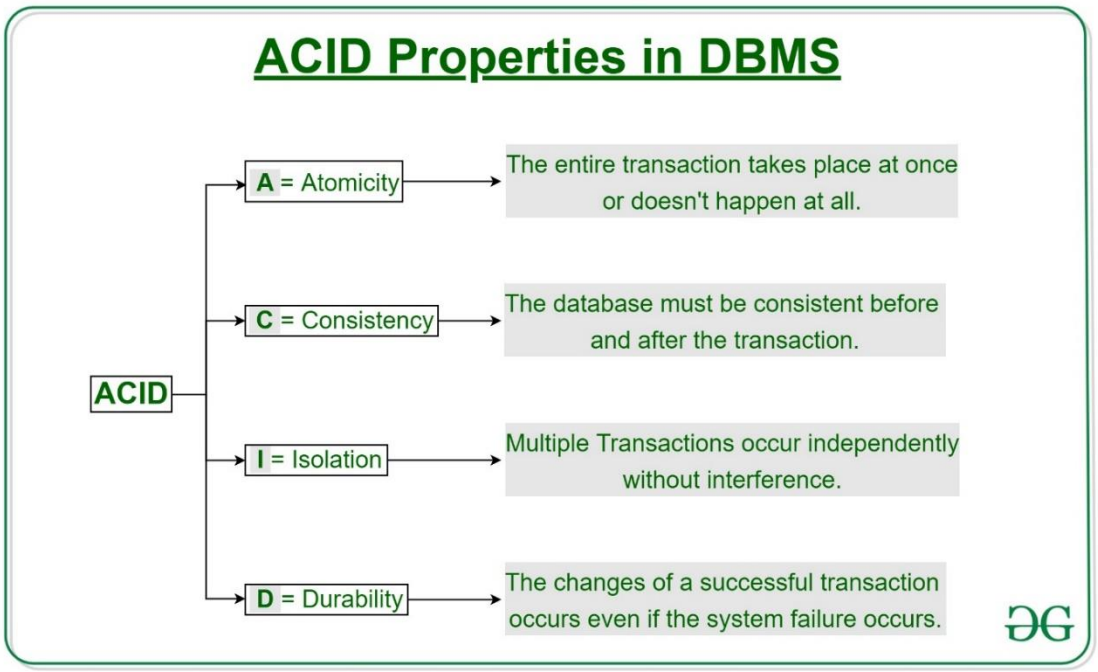


Figure 31: ACID Property

Atomicity:
By this, we mean that either the entire transaction takes place at once or does not happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
—**Abort**: If a transaction aborts, changes made to the database are not visible.
—**Commit**: If a transaction commits, changes made are visible.
Atomicity is also known as the 'All or nothing rule'.
Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Consistency:
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.
Total **before T** occurs = **500 + 200 = 700**.
Total **after T occurs** = **400 + 300 = 700**.
Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation:
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result, interleaving of operations takes place due to which **T''** reads the correct value of **X** but the incorrect value of **Y** and sum computed by
T'': (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of the transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Some important points:

Property	Responsibility for maintaining properties
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.
ACID properties are the four key characteristics that define the reliability and consistency of a transaction in a Database Management System (DBMS). The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. Here is a brief description of each of these properties:

- **Atomicity**: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its original state, ensuring data consistency and integrity.
- **Consistency**: Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database is in a consistent state both before and after the transaction is executed. Constraints, such as unique keys and foreign keys, must be maintained to ensure data consistency.
- **Isolation**: Isolation ensures that multiple transactions can execute concurrently without interfering with each other. Each transaction must be isolated from other transactions until it is completed. This isolation prevents dirty reads, non-repeatable reads, and phantom reads.
- **Durability**: Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The transaction's changes are saved to the database permanently, and even if the system crashes, the changes remain intact and can be recovered.

Overall, ACID properties provide a framework for ensuring data consistency, integrity, and reliability in DBMS. They ensure that transactions are executed in a reliable and consistent manner, even in the presence of system failures, network issues, or other problems. These properties make DBMS a reliable and efficient tool for managing data in modern organizations.

Advantages of ACID Properties in DBMS:

- Data Consistency: ACID properties ensure that the data remains consistent and accurate after any transaction execution.
- Data Integrity: ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
- Concurrency Control: ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
- Recovery: ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS:

- Performance: The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
- Scalability: The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
- Complexity: Implementing the ACID properties can increase the complexity of the system and require significant expertise and resources.
  Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data.
- management, ensuring data integrity, accuracy, and reliability. However, in some cases, the overhead of implementing ACID properties can cause performance and scalability issues. Therefore, it's important to balance the benefits of ACID properties against the specific needs and requirements of the system.

13. Database Recovery Techniques

Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS.

- **Rollback/Undo Recovery Technique**
- **Commit/Redo Recovery Technique**

Rollback/Undo Recovery Technique

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

Commit/Redo Recovery Technique

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

In addition to these two techniques, there is also a third technique called checkpoint recovery.

Checkpoint Recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Overall, recovery techniques are essential to ensure data consistency and availability in Database Management System, and each technique has its own

advantages and limitations that must be considered in the design of a recovery system.

Database Systems

There are both automatic and non-automatic ways for both, backing up data and recovery from any failure situations. The techniques used to recover lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect command execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred updates and immediate updates or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- **The log is kept on disk start_transaction(T):** This log entry records that transaction T starts the execution.
- **read_item(T, X):** This log entry records that transaction T reads the value of database item X.
- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted.
- **checkpoint:** A 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 a consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, the item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

- **Undoing:** If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database

to old-value. There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.

- **Deferred Update:** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm.**
- **Immediate Update:** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e., the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**
- **Caching/Buffering:** In these one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under the control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.
- **Shadow Paging:** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to the original are updated to refer new replacement page.
- **Backward Recovery:** The term "**Rollback**" and "**UNDO**" can also refer to backward recovery. When a backup of the data is not available and previous modifications need to be undone, this technique can be helpful. With the backward recovery method, unused modifications are removed, and the database is returned to its prior condition. All adjustments made during the previous traction are reversed during the backward recovery. In other words, it reprocesses valid transactions and undoes the erroneous database updates.

- **Forward Recovery:** "**Roll forward** "and "**REDO**" refers to forwarding recovery. When a database needs to be updated with all changes verified, this forward recovery technique is helpful. Some failed transactions in this database are applied to the database to roll those modifications forward. In other words, the database is restored using preserved data and valid transactions counted by their past saves.

Backup Techniques

There are different types of Backup Techniques. Some of them are listed below.

- **Full database Backup:** In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
- **Differential Backup:** It stores only the data changes that have occurred since the last full database backup. When some data has changed many times since the last full database backup, a differential backup stores the most recent version of the changed data. For this first, we need to restore a full database backup.
- **Transaction Log Backup:** In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transactions that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.

Log based recovery

The atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications done by an aborted transaction should not be visible to the database and the modifications done by the committed transaction should be visible. To achieve our goal of atomicity, the user must first output stable storage information describing the modifications, without modifying the database itself. This information can help us ensure that all modifications performed by committed transactions are reflected in the database. This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

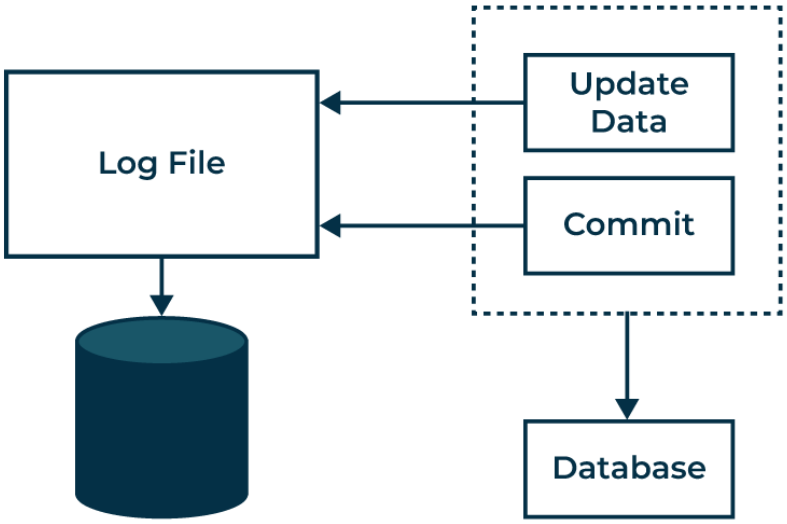


Figure 32: Log Based Recovery

Log and log records

The log is a sequence of log records, recording all the updated activities in the database. In stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded on the log. Prior to performing any modification to the database, an updated log record is created to reflect that modification. An update log record represented as: <Ti, Xj, V1, V2> has these fields:

i) **Transaction identifier:** Unique Identifier of the transaction that performed the write operation.
ii) **Data item:** Unique identifier of the data item written.
iii) **Old value:** Value of data item prior to write.
iv) **New value:** Value of data item after write operation.

Other types of log records are:

i) **<Ti start>**: It contains information about when a transaction Ti starts.
ii) **<Ti commit>**: It contains information about when a transaction Ti commits.
iii) **<Ti abort>**: It contains information about when a transaction Ti aborts.

Undo and Redo Operations

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

i) **Undo:** using a log record sets the data item specified in log record to old value.

ii) **Redo:** using a log record sets the data item specified in log record to new value.

The database can be modified using two approaches –

i) **Deferred Modification Technique:** If the transaction does not modify the database until it has partially committed, it is said to use deferred modification technique.
ii) **Immediate Modification Technique:** If database modification occur while the transaction is still active, it is said to use immediate modification technique.

Recovery using Log records

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone.

i) Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain either the record <Ti commit> or the record <Ti abort>.
ii) Transaction Ti needs to be redone if log contains record <Ti start> and either the record <Ti commit> or the record <Ti abort>.

Advantages of Log based Recovery

- **Durability:** In the event of a breakdown, the log file offers a dependable and long-lasting method of recovering data. It guarantees that in the event of a system crash, no committed transaction is lost.
- **Faster Recovery:** Since log-based recovery recovers databases by replaying committed transactions from the log file, it is typically faster than alternative recovery methods.
- **Incremental Backup:** Backups can be made in increments using log-based recovery. Just the changes made since the last backup are kept in the log file, rather than creating a complete backup of the database each time.
- **Lowers the Risk of Data Corruption:** By making sure that all transactions are correctly committed or canceled before they are written to the database, log-based recovery lowers the risk of data corruption.

Disadvantages of Log based Recovery

- **Additional overhead:** Maintaining the log file incurs an additional overhead on the database system, which can reduce the performance of the system.
- **Complexity:** Log-based recovery is a complex process that requires careful management and administration. If not managed properly, it can lead to data inconsistencies or loss.

- **Storage space:** The log file can consume a significant amount of storage space, especially in a database with many transactions.
- **Time-Consuming:** The process of replaying the transactions from the log file can be time-consuming, especially if there are many transactions to recover.

Why Recovery is Required in Database?

Here are some of the reasons why recovery is needed in DBMS.

- **System failures:** The DBMS can experience various types of failures, such as hardware failures, software bugs, or power outages, which can lead to data corruption or loss. Recovery mechanisms can help restore the database to a consistent state after such failures.
- **Transaction failures:** Transactions can fail due to various reasons, such as network failures, deadlock, or errors in application logic. Recovery mechanisms can help roll back or undo the effects of such failed transactions to ensure data consistency.
- **Human errors:** Human errors such as accidental deletion, updating or overwriting data, or incorrect data entry can cause data inconsistencies. Recovery mechanisms can help recover the lost or corrupted data and restore it to the correct state.
- **Security breaches:** Security breaches such as hacking or unauthorized access can compromise the integrity of data. Recovery mechanisms can help restore the database to a consistent state and prevent further data breaches.
- **Hardware upgrades:** When a DBMS is upgraded to a new hardware system, the migration process can potentially lead to data loss or corruption. Recovery mechanisms can help ensure that the data is successfully migrated and the integrity of the database is maintained.
- **Natural disasters:** Natural disasters such as earthquakes, floods, or fires can damage the hardware on which the database is stored, leading to data loss. Recovery mechanisms can help restore the data from backups and minimize the impact of the disaster.
- **Compliance regulations:** Many industries have regulations that require businesses to retain data for a certain period. Recovery mechanisms can help ensure that the data is available for compliance purposes even if it was deleted or lost accidentally.
- **Data corruption:** Data corruption can occur due to various reasons such as hardware failure, software bugs, or viruses. Recovery mechanisms can help restore the database to a consistent state and recover any lost or corrupted data.

14. Transaction and Isolation Levels

As we know, to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation, and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena:

- **Dirty Read –** A Dirty read is a situation when a transaction reads data that has not yet been committed. For example, Let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Non-Repeatable read** – Non-Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, now if transaction T1 rereads the same data, it will retrieve a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels:

- **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.
- **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.

- **Repeatable Read** – This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.
- **Serializable** – This is the highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The Table given below clearly depicts the relationship between isolation levels, read phenomena, and locks:

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Anomaly Serializable is not the same as Serializable. That is, it is necessary, but not sufficient that a Serializable schedule should be free of all three phenomena types.

Transaction isolation levels are used in database management systems (DBMS) to control the level of interaction between concurrent transactions.

The four standard isolation levels are:

- **Read Uncommitted:** This is the lowest level of isolation where a transaction can see uncommitted changes made by other transactions. This can result in dirty reads, non-repeatable reads, and phantom reads.
- **Read Committed:** In this isolation level, a transaction can only see changes made by other committed transactions. This eliminates dirty reads but can still result in non-repeatable reads and phantom reads.
- **Repeatable Read:** This isolation level guarantees that a transaction will see the same data throughout its duration, even if other transactions commit changes to the data. However, phantom reads are still possible.
- **Serializable:** This is the highest isolation level where a transaction is executed as if it were the only transaction in the system. All transactions must be executed sequentially, which ensures that there are no dirty reads, non-repeatable reads, or phantom reads.

The choice of isolation level depends on the specific requirements of the application. Higher isolation levels offer stronger data consistency but can also result in longer lock times and increased contention, leading to decreased concurrency and performance. Lower isolation levels provide more concurrency but can result in data inconsistencies.

In addition to the standard isolation levels, some DBMS may also support additional custom isolation levels or features such as snapshot isolation and multi-version concurrency control (MVCC) that provide alternative solutions to the problems addressed by the standard isolation levels.

Advantages of Transaction Isolation Levels:

Improved concurrency: Transaction isolation levels can improve concurrency by allowing multiple transactions to run concurrently without interfering with each other.
Control over data consistency: Isolation levels provide control over the level of data consistency required by a particular application.
Reduced data anomalies: The use of isolation levels can reduce data anomalies such as dirty reads, non-repeatable reads, and phantom reads.
Flexibility: The use of different isolation levels provides flexibility in designing applications that require different levels of data consistency.

Disadvantages of Transaction Isolation Levels:

- **Increased overhead:** The use of isolation levels can increase overhead because the database management system must perform additional checks and acquire more locks.
- **Decreased concurrency:** Some isolation levels, such as Serializable, can decrease concurrency by requiring transactions to acquire more locks, which can lead to blocking.
- **Limited support:** Not all database management systems support all isolation levels, which can limit the portability of applications across different systems.
- **Complexity:** The use of different isolation levels can add complexity to the design of database applications, making them more difficult to implement and maintain.

14.1. Types of Schedules in DBMS

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly. The basics of Transactions and Schedules is discussed in Concurrency Control (Introduction), and Transaction Isolation Levels in DBMS articles.
Here we will discuss various types of schedules.

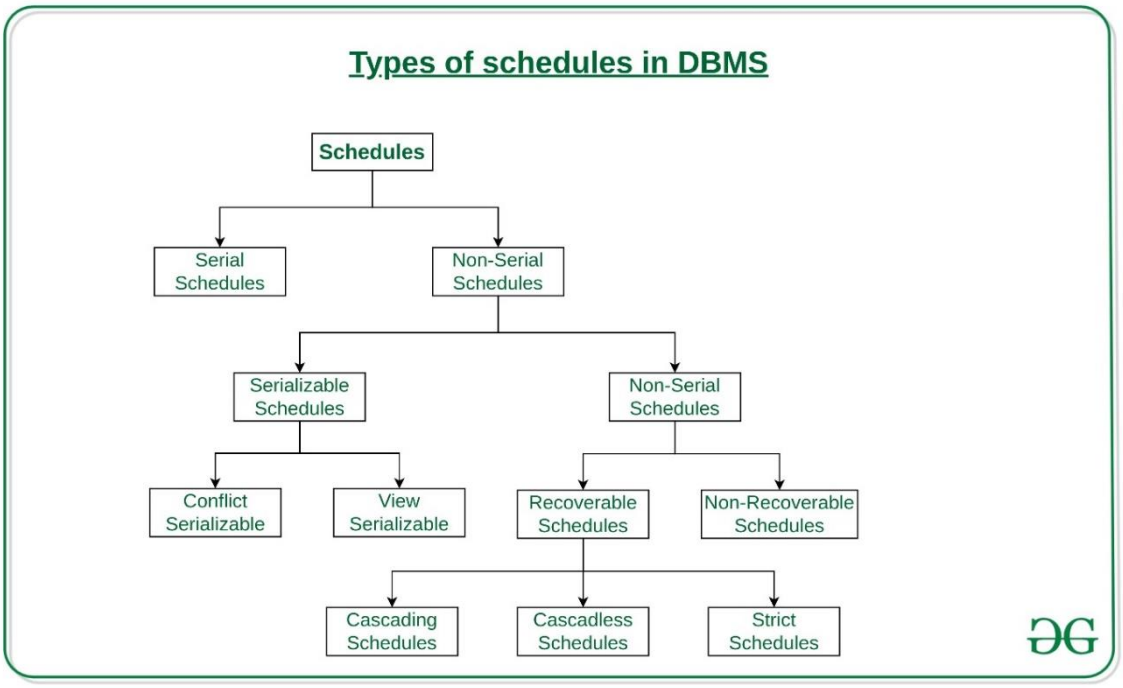


Figure 33: Types of Schedule in DBMS

1) Serial Schedules:

Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

Example:

Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

where R(A) denotes that a read operation is performed on some data item ‘A’
This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

2) Non-Serial Schedule:

This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.
The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

3) Serializable:

This is used to maintain the consistency of the database. It is mainly used in the non-serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

- **Conflict Serializable:**
  A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:
  - They belong to different transactions
  - They operate on the same data item
  - At Least one of them is a write operation

- **View Serializable:**
  A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

4) **Non-Serializable:**
The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

- **Recoverable Schedule:**
Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Example –
Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.
There can be three types of recoverable schedule:

a) **Cascading Schedule:**
Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

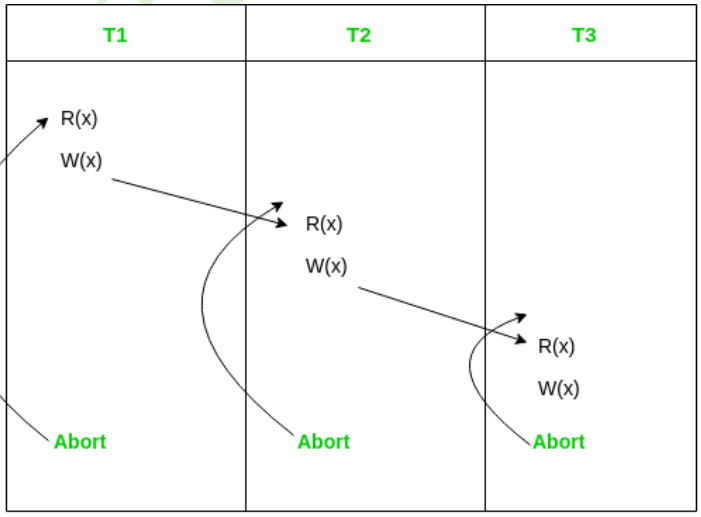


Figure - Cascading Abort

Figure 34: Cascading Abort

b) Cascadeless Schedule:

Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.

Example:

Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

c) Strict Schedule:

A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.

In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.

Example:

Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

- **Non-Recoverable Schedule:**

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

Note – It can be seen that:

- Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
- Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
- Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:

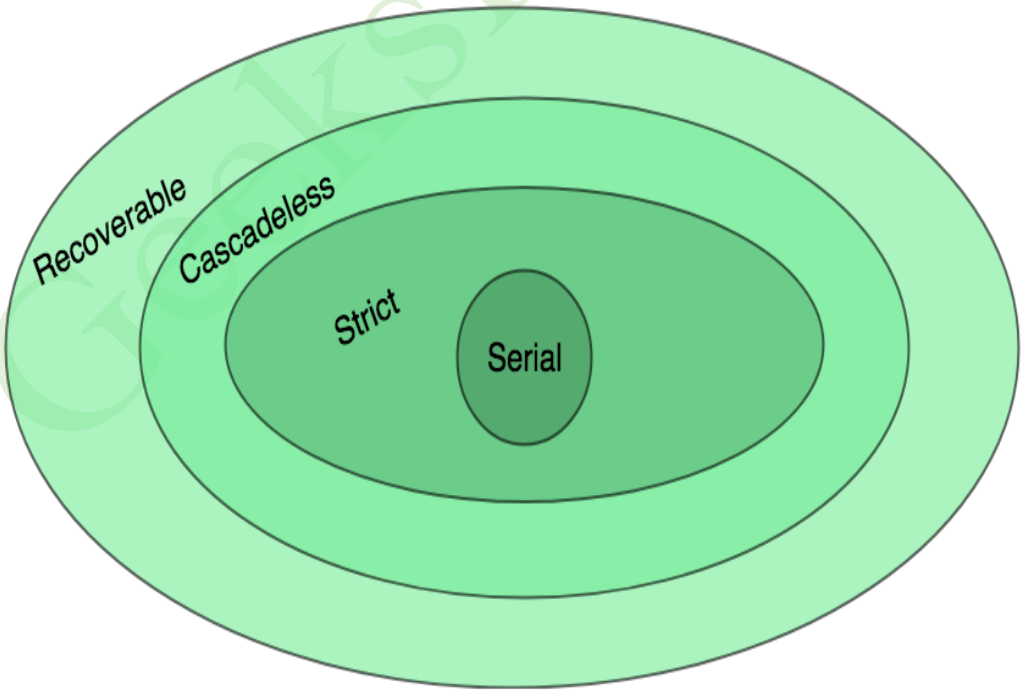


Figure 35: Relation between Schedules

14.2. Types of Schedules based Recoverability

we are going to deal with the types of Schedules based on the Recoverability in Database Management Systems (DBMS). Generally, there are three types of schedules given as follows:

Schedules Based on Recoverability

- **Recoverable Schedule:** A schedule is recoverable if it allows for the recovery of the database to a consistent state after a transaction failure. In a recoverable schedule, a transaction that has updated the database must commit before any other transaction reads or writes the same data. If a transaction fails before committing, its updates must be rolled back, and any transactions that have read its uncommitted data must also be rolled back.
- **Cascadeless Schedule:** A schedule is cascaded less if it does not result in a cascading rollback of transactions after a failure. In a cascade-less schedule, a transaction that has read uncommitted data from another transaction cannot commit before that transaction commits. If a transaction fails before committing, its updates must be rolled back, but any transactions that have read its uncommitted data need not be rolled back.
- **Strict Schedule:** A schedule is strict if it is both recoverable and cascades. In a strict schedule, a transaction that has read uncommitted data from another transaction cannot commit before that transaction commits, and a transaction that has updated the database must commit before any other transaction reads or writes the same data. If a transaction fails before committing, its updates must be rolled back, and any transactions that have read its uncommitted data must also be rolled back.

These types of schedules are important because they affect the consistency and reliability of the database system. It is essential to ensure that schedules are recoverable, cascaded, or strict to avoid inconsistencies and data loss in the database.

Recoverable Schedule

A schedule is said to be recoverable if it is recoverable as the name suggests. Only reads are allowed before write operations on the same data. Only reads (Ti->Tj) are permissible.

Example:

S1: R1(x), **W1(x)**, R2(x), R1(y), R2(y),
W2(x), W1(y), **C1**, **C2**;

The given schedule follows the order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chance of conflict occurring. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of the first transaction performed the first update on data item x, hence given schedule is recoverable.

Let us see an example of an **unrecoverable schedule** to clear the concept more.
S2: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x),
W3(y), R2(y), W2(z), **W2(y)**, C1, **C2**, **C3**;
Ti->Tj => C2->C3 but W3(y) executed before W2(y) which leads to conflicts thus it must be committed before the T2 transaction. So given schedule is unrecoverable. if **Ti->Tj => C3->C2** is given in the schedule then it will become a recoverable schedule.

Note: *A committed transaction should never be rollback. It means that reading value from uncommitted transaction and commit it will enter the current transaction into inconsistent or unrecoverable state this is called Dirty Read problem.*

Cascadeless Schedule
When no **read** or **write-write** occurs before the execution of the transaction then the corresponding schedule is called a cascadeless schedule.
Example:
S3: R1(x),
R2(z), R3(x), R1(z), R2(y), R3(y), W1(x), C1,
W2(z), **W3(y)**, **W2(y)**, **C3**, **C2**;

In this schedule **W3(y)** and **W2(y)** overwrite conflicts and there is no read, therefore given schedule is cascade less schedule.

Special Case: *A committed transaction desired to abort. As given below all the transactions are reading committed data hence it's cascadeless schedule.*

Strict Schedule
If the schedule contains no **read** or **write** before commit, then it is known as a strict schedule. A strict schedule is strict in nature.
Example:
S4: R1(x), R2(x), R1(z), R3(x), R3(y),
W1(x), **C1**, **W3(y)**, **C3**, R2(y), **W2(z)**, **W2(y)**, **C2**;
In this schedule, no read-write or write-write conflict arises before committing hence its strict schedule:

Cascading Abort:
Cascading Abort can also be rollback. If transaction T1 aborts as T2 read data that is written by T1 it is not committed. Hence its cascading rollback.

Co-Relation between Strict, Cascadeless, and Recoverable schedules
Below is the picture showing the correlation between Strict Schedules, Cascadeless Schedules, and Recoverable Schedules.
So, we can conclude that:

- Strict schedules are all recoverable and cascade schedules.
- All cascade-less schedules are recoverable.

14.3. Conflict Serializability

in Concurrency control, serial schedules have less resource utilization and low throughput. To improve it, two or more transactions are run concurrently. However, concurrency of transactions may lead to inconsistency in the database. To avoid this, we need to check whether these concurrent schedules are serializable or not.
Conflict Serializable
Concurrency serializability, also known as conflict serializability, is a type of concurrency control that guarantees that the outcome of concurrent transactions is the same as if the transactions were executed consecutively.
Conflict serializable schedules: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
Non-conflicting operations: When two operations operate on separate data items or the same data item but at least one of them is a read operation, they are said to be non-conflicting.

Conflicting Operations
Two operations are said to be conflicting if all conditions are satisfied:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

Example:
Conflicting operations pair (R1(A), W2(A)) because they belong to two different transactions on the same data item A and one of them is a write operation.
Similarly, (W1(A), W2(A)) and (W1(A), R2(A)) pairs are also **conflicting**.
On the other hand, the (R1(A), W2(B)) pair is **non-conflicting** because they operate on different data items.
Similarly, ((W1(A), W2(B)) pair is **non-conflicting.**

Consider the following schedule:
S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)
If Oi and Oj are two operations in a transaction and Oi< Oj (Oi is executed before Oj), same order will follow in the schedule as well. Using this property, we can get two transactions of schedule S1:
T1: R1(A), W1(A), R1(B), W1(B)
T2: R2(A), W2(A), R2(B), W2(B)

Possible Serial Schedules are: T1->T2 or T2->T1
-> **Swapping non-conflicting operation**s R2(A) and R1(B) in S1, the schedule becomes,
S11: R1(A), W1(A), R1(B), **W2(A),** R2(A), **W1(B),** R2(B), W2(B)
-> Similarly, **swapping non-conflicting operations** W2(A) and W1(B) in S11, the schedule becomes,
S12: R1(A), W1(A), R1(B), W1(B), R2(A), W2(A), R2(B), W2(B)
S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.
Let us take another Schedule:
S2: R2(A), W2(A), R1(A), W1(A), R1(B), W1(B), R2(B), W2(B)
Two transactions will be:
T1: R1(A), W1(A), R1(B), W1(B)
T2: R2(A), W2(A), R2(B), W2(B)

Possible Serial Schedules are: T1->T2 or T2->T1
Original Schedule is as:
S2: R2(A), W2(A), **R1(A),** W1(A), R1(B), W1(B), **R2(B),** W2(B)
Swapping non-conflicting operations R1(A) and R2(B) in S2, the schedule becomes,
S21: R2(A), W2(A), R2(B), **W1(A),** R1(B), W1(B), R1(A), **W2(B)**
Similarly, swapping non-conflicting operations W1(A) and W2(B) in S21, the schedule becomes,
S22: R2(A), W2(A), R2(B), W2(B), R1(B), W1(B), R1(A), W1(A)
In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be R1(A), W1(A), R1(B), W1(B)). So S2 is not conflict serializable.

Conflict Equivalent
Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12, and so on.

Note 1: *Although S2 is not conflict serializable, still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.*
Note 2: *The schedule which is conflict serializable is always conflict equivalent to one of the serial schedules. S1 schedule discussed above (which is conflict serializable) is equivalent to the serial schedule (T1->T2).*

14.4. Precedence Graph for Testing Conflict Serializability

A **Precedence Graph** or **Serialization Graph** is used commonly to test the Conflict Serializability of a schedule. It is a directed Graph (V, E) consisting of a set of nodes V = {T1, T2, T3……. Tn} and a set of directed edges E = {e1, e2, e3……em}. The graph contains one node for each Transaction Ti. An edge ei is of the form Tj –> Tk where Tj is the starting node of ei and Tk is the ending node of ei. An edge ei is constructed between nodes Tj to Tk if one of the operations in Tj appears in the schedule before some conflicting operation in Tk. The Algorithm can be written as:

- Create a node T in the graph for each participating transaction in the schedule.
- For the conflicting operation read_item(X) and write_item(X) – If a Transaction Tj executes a read_item (X) after Ti executes a write_item (X), draw an edge from Ti to Tj in the graph.
- For the conflicting operation write_item(X) and read_item(X) – If a Transaction Tj executes a write_item (X) after Ti executes a read_item (X), draw an edge from Ti to Tj in the graph.
- For the conflicting operation write_item(X) and write_item(X) – If a Transaction Tj executes a write_item (X) after Ti executes a write_item (X), draw an edge from Ti to Tj in the graph.
- Schedule S is serializable if there is no cycle in the precedence graph.

If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S. The serial schedule S' can be found by Topological Sorting of the acyclic precedence graph. Such schedules can be more than 1. For example, Consider the schedule S:
S: r1(x) r1(y) w2(x) w1(x) r2(y)

What are the Steps to Construct a Precedence Graph?
Step 1: Draw a node for each transaction in the schedule.

Step 2: For each pair of conflicting operations (i.e., operations on the same data item by different transactions), draw an edge from the transaction that performed the first operation to the transaction that performed the second operation. The edge represents a dependency between the two transactions.

Step 3: If there are multiple conflicting operations between two transactions, draw multiple edges between the corresponding nodes.

Step 4: If there are no conflicting operations between two transactions, do not draw an edge between them.

Step 5: Once all the edges have been added to the graph, check if the graph contains any cycles. If the graph contains cycles, then the schedule is not conflict serializable. Otherwise, the schedule is conflict serializable.

The precedence graph provides a visual representation of the dependencies between transactions in a schedule and allows us to determine whether the schedule is a conflict serializable or not. By constructing the precedence graph, we can identify the transactions that have conflicts and reorder them to produce a conflict serializable schedule, which is a schedule that can be transformed into a serial schedule by swapping non-conflicting operations.

Advantages of Precedence Graphs for Testing Conflict Serializability

- **Simple to comprehend:** Because precedence graphs show the connections between transactions visually, they are simple to comprehend.
- **Quick analysis:** You can rapidly ascertain whether or not a series of transactions can be conflict serialized by using precedence graphs.
- **Finding anomalies:** Anomalies like cycles or deadlocks that might not be seen right away might be found using precedence graphs.
- **Assists with optimization:** By identifying transactions that can be carried out in parallel, precedence graphs can be utilized to enhance a database system's performance.

Disadvantages of Precedence Graphs for Testing Conflict Serializability

- **Complex for large systems:** It can be challenging to discern dependencies between transactions in large database systems due to the complexity of precedence graphs.
- **Potential for inaccurate results:** It is possible that some conflicts between transactions will be unnoticed by precedence graphs.
- **Require Manual efforts:** Building precedence graphs by hand can be labour-intensive and time-consuming, particularly in the case of big systems.
- **Limited applicability:** Data races and deadlocks cannot be detected with precedence graphs; they are only useful for assessing conflict serializability.

14.5. Recoverability

Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.
The recoverability property is enforced through the use of transaction logs, which record all changes made to the database during transaction processing. When a failure occurs, the system uses the log to recover the database to a consistent state, which involves either undoing the effects of uncommitted transactions or redoing the effects of committed transactions.

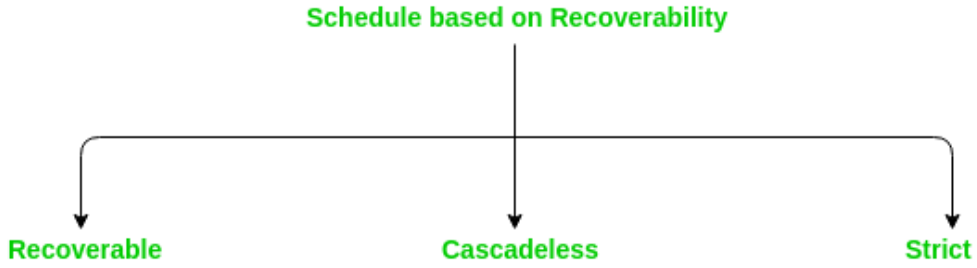
There are several levels of recoverability that can be supported by a database system:

- **No-undo logging:** This level of recoverability only guarantees that committed transactions are durable, but does not provide the ability to undo the effects of uncommitted transactions.
- **Undo logging:** This level of recoverability provides the ability to undo the effects of uncommitted transactions but may result in the loss of updates made by committed transactions that occur after the failed transaction.
- **Redo logging:** This level of recoverability provides the ability to redo the effects of committed transactions, ensuring that all committed updates are durable and can be recovered in the event of failure.
- **Undo-redo logging:** This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

In addition to these levels of recoverability, database systems may also use techniques such as checkpointing and shadow paging to improve recovery performance and reduce the overhead associated with logging.

Recoverable Schedules:
Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.
Example 1: S1: R1(x), **W1(x)**, R2(x), R1(y), R2(y),
W2(x), W1(y), **C1**, **C2**;
Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

14.6. Cascadeless in DBMS



Generally, there are 3 types of schedule based on recoverbility given as follows:

- **Recoverable schedule:**
  Transactions must be committed in order. Dirty Read problem and Lost Update problem may occur.
- **Cascadeless Schedule:**
  Dirty Read not allowed, means reading the data written by an uncommitted transaction is not allowed. Lost Update problem may occur.
- **Strict schedule:**
  Neither Dirty read nor Lost Update problem allowed, means reading or writing the data written by an uncommitted transaction is not allowed.

Cascading Rollback:
If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.

These Cascading Rollbacks occur because of **Dirty Read problems**.
For example, transaction T1 writes uncommitted x that is read by Transaction T2. Transaction T2 writes uncommitted x that is read by Transaction T3. Suppose at this point T1 fails.
T1 must be rolled back, since T2 is dependent on T1, T2 must be rolled back, and since T3 is dependent on T2, T3 must be rolled back.
Because of T1 rollback, all T2, T3, and T4 should also be rollback (Cascading dirty read problem).

This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.

Cascadeless Schedule:
This schedule avoids all possible *Dirty Read Problem*.
In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits. That means there must not be **Dirty Read**. Because Dirty Read Problem can cause *Cascading Rollback*, which is inefficient.
Cascadeless Schedule avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

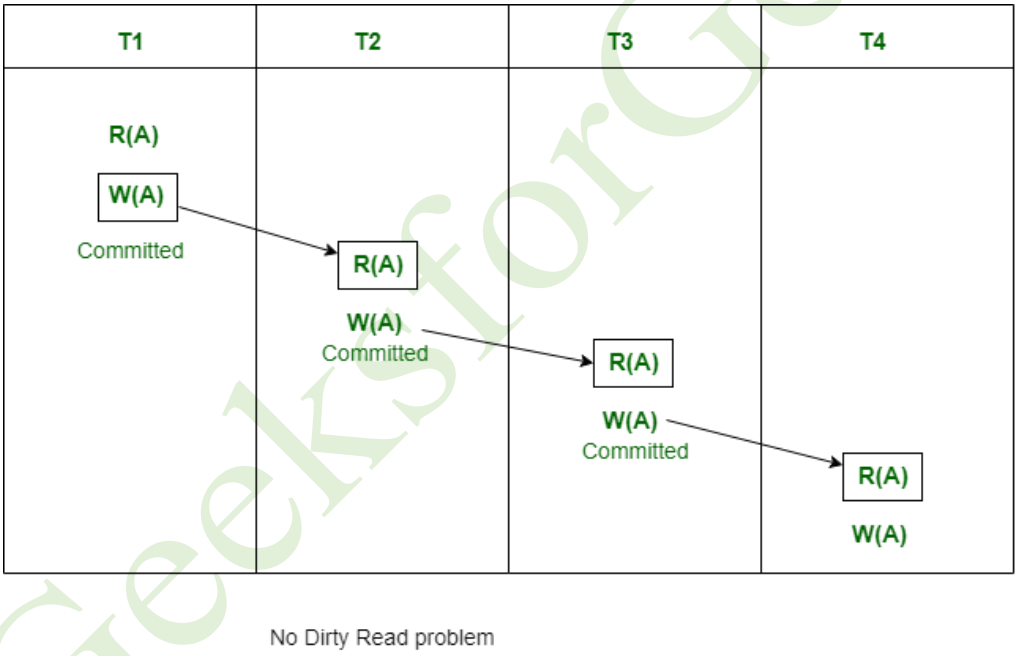


Figure 36: No Sirty Read Problem

Note: *Cascadeless schedule allows only committed read operations. However, it allows uncommitted write operations.*

14.7. Concurrency Control

Concurrently control is a very important concept of DBMS which ensures the simultaneous execution or manipulation of data by several processes or user without resulting in data inconsistency. Concurrency Control deals with **interleaved execution** of more than one transaction.

What is Transaction?

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department A to the account of department B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

The concept of a transaction has been applied broadly in database systems and applications. While the initial use of transactions was in financial applications, the concept is now used in real-time applications in telecommunication, as well as in the management of long-duration activities such as product design or administrative workflows.

A set of logically related operations is known as a transaction. The main operations of a transaction are:

- **Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in the main memory.
- **Write (A):** Write operation Write(A) or W(A) writes the value back to the database from the buffer.

(Note: It doesn't always need to write it to a database back it just writes the changes to buffer this is the reason where dirty read comes into the picture)

Let us take a debit transaction from an account that consists of the following operations:

- R(A);
- A=A-1000;
- W(A);

Assume A's value before starting the transaction is 5000.

- The first operation reads the value of A from the database and stores it in a buffer.
- the Second operation will decrease its value by 1000. So buffer will contain 4000.
- the Third operation will write the value from the buffer to the database. So A's final value will be 4000.

But it may also be possible that the transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power,** etc. For example, if the debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

- **Commit:** After all instructions of a transaction are successfully executed, the changes made by a transaction are made permanent in the database.
- **Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by a transaction are undone.

Properties of a Transaction

- **Atomicity:** As a transaction is a set of logically related operations, **either all of them should be executed or none**. A debit transaction discussed above should either execute all three operations or none. If the debit transaction fails after executing operations 1 and 2 then its new value of 4000 will not be updated in the database which leads to inconsistency.
- **Consistency:** If operations of debit and credit transactions on the same account are executed concurrently, it may leave the database in an inconsistent state.

Isolation: The result of a transaction should not be visible to others before the transaction is committed. For example, let us assume that A's balance is Rs. 5000 and T1 debits Rs. 1000 from A. A's new balance will be 4000. If T2 credits Rs. 500 to A's new balance, A will become 4500, and after this T1 fails. Then we have to roll back T2 as well because it is using the value produced by T1. So transaction results are not made visible to other transactions before it commits.

Durable: Once the database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, the bank can't say that the update has been lost. To avoid this problem, multiple copies of the database are stored at different locations.

What is a Schedule?

A schedule is a series of operations from one or more transactions.
A schedule can be of two types:

1) **Serial Schedule:** When one transaction completely executes before starting another transaction, the schedule is called a serial schedule. A serial schedule is always consistent. e.g.; If a schedule S has debit transaction T1 and credit transaction T2, possible serial schedules are T1 followed by T2 (T1->T2) or T2 followed by T1 ((T2->T1). A serial schedule has low throughput and less resource utilization.
2) **Concurrent Schedule:** When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called a Concurrent schedule. e.g.; the Schedule of debit and credit transactions shown in Table 1 is concurrent. But concurrency can lead to inconsistency in the database. The above example of a concurrent schedule is also inconsistent.

Difference between Serial Schedule and Serializable Schedule

Serial Schedule	Serializable Schedule
In Serial schedule, transactions will be executed one after other.	In Serializable schedule transaction are executed concurrently.
Serial schedule are less efficient.	Serializable schedule are more efficient.
In serial schedule only one transaction executed at a time.	In Serializable schedule multiple transactions can be executed at a time.
Serial schedule takes more time for execution.	In Serializable schedule execution is fast.

Concurrency Control in DBMS

- Executing a single transaction at a time will increase the waiting time of the other transactions which may result in delay in the overall execution. Hence for increasing the overall throughput and efficiency of the system, several transactions are executed.
- Concurrently control is a very important concept of DBMS which ensures the simultaneous execution or manipulation of data by several processes or user without resulting in data inconsistency.
- Concurrency control provides a procedure that is able to control concurrent execution of the operations in the database.
- The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable.

In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database

consistency constraints as the basis for a split of the database into sub databases on which concurrency can be managed separately.
Another is to treat some operations besides read and write as fundamental low-level operations and to extend concurrency control to deal with them.

Concurrency Control Problems
There are several problems that arise when numerous transactions are executed simultaneously in a random manner. The database transaction consist of two major operations “Read” and “Write”. It is very important to manage these operations in the concurrent execution of the transactions in order to maintain the consistency of the data.
Dirty Read Problem(Write-Read conflict)
Dirty read problem occurs when one transaction updates an item but due to some unconditional events that transaction fails but before the transaction performs rollback, some other transaction reads the updated value. Thus creates an inconsistency in the database. Dirty read problem comes under the scenario of

Write-Read conflict between the transactions in the database.

- The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.
- Transaction T1 modifies a database record without committing the changes.
- T2 reads the uncommitted data changed by T1.
- T1 performs rollback.
- T2 has already read the uncommitted data of T1 which is no longer valid, thus creating inconsistency in the database.

Lost Update Problem
Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.

- T1 reads the value of an item from the database.
- T2 starts and reads the same database item.
- T1 updates the value of that data and performs a commit.
- T2 updates the same data item based on its initial read and performs commit.
- This results in the modification of T1 gets lost by the T2’s writes which causes a lost update problem in the database.

Concurrency Control Protocols
Concurrency control protocols are the set of rules which are maintained in order to solve the concurrency control problems in the database. It ensures that the

concurrent transactions can execute properly while maintaining the database consistency. The concurrent execution of a transaction is provided with atomicity, consistency, isolation, durability, and serializability via the concurrency control protocols.

- Locked based concurrency control protocol
- Timestamp based concurrency control protocol

Locked based Protocol

In locked based protocol, each transaction needs to acquire locks before they start accessing or modifying the data items. There are two types of locks used in databases.

- **Shared Lock :** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item but it can not modify the data item.
- **Exclusive Lock :** Exclusive lock is also known as the write lock. Exclusive lock allows a transaction to update a data item. Only one transaction can hold the exclusive lock on a data item at a time. While a transaction is holding an exclusive lock on a data item, no other transaction is allowed to acquire a shared/exclusive lock on the same data item.

There are two kinds of lock-based protocol mostly used in database:

- **Two Phase Locking Protocol:** Two phase locking is a widely used technique which ensures strict ordering of lock acquisition and release. Two phase locking protocol works in two phases.
    - **Growing Phase:** In this phase, the transaction starts acquiring locks before performing any modification on the data items. Once a transaction acquires a lock, that lock cannot be released until the transaction reaches the end of the execution.
    - **Shrinking Phase:** In this phase, the transaction releases all the acquired locks once it performs all the modifications on the data item. Once the transaction starts releasing the locks, it cannot acquire any locks further.
- **Strict Two Phase Locking Protocol :** It is almost similar to the two phase locking protocol the only difference is that in two phase locking the transaction can release its locks before it commits, but in case of strict two phase locking the transactions are only allowed to release the locks only when they performs commits.

Timestamp based Protocol.

In this protocol each transaction has a timestamp attached to it. Timestamp is nothing but the time in which a transaction enters the system.

The conflicting pairs of operations can be resolved by the timestamp ordering protocol through the utilization of the timestamp values of the transactions. Therefore, guaranteeing that the transactions take place in the correct order.

Advantages of Concurrency

In general, concurrency means, that more than one transaction can work on a system. The advantages of a concurrent system are:

- **Waiting Time:** It means if a process is in a ready state but still the process does not get the system to get execute is called waiting time. So, concurrency leads to less waiting time.
- **Response Time:** The time wasted in getting the response from the CPU for the first time, is called response time. So, concurrency leads to less Response Time.
- **Resource Utilization:** The amount of Resource utilization in a particular system is called Resource Utilization. Multiple transactions can run parallel in a system. So, concurrency leads to more Resource Utilization.
- **Efficiency:** The amount of output produced in comparison to given input is called efficiency. So, Concurrency leads to more Efficiency.

Disadvantages of Concurrency

- **Overhead:** Implementing concurrency control requires additional overhead, such as acquiring and releasing locks on database objects. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.
- **Deadlocks:** Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve and can result in reduced throughput and increased latency.
- **Reduced concurrency:** Concurrency control can limit the number of users or applications that can access the database simultaneously. This can lead to reduced concurrency and slower performance in systems with high levels of concurrency.
- **Complexity:** Implementing concurrency control can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.

14.8. Concurrency Control Techniques

Concurrency control is provided in a database to:

- enforce isolation among transactions.
- preserve database consistency through consistency preserving execution of transactions.
- resolve read-write and write-read conflicts.

Various concurrency control techniques are:

i) Two-phase locking Protocol
ii) Time stamp ordering Protocol
iii) Multi version concurrency control
iv) Validation concurrency control

These are briefly explained below.

i) Two-Phase Locking Protocol:

Locking is an operation which secures permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two-phase update algorithm are:

- Lock Acquisition
- Modification of Data
- Release Lock

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention. A transaction in the Two-Phase Locking Protocol can assume one of the 2 phases:

1st Phase: **Growing Phase:** In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.

2nd Phase: **Shrinking Phase:** In this phase a transaction can only release locks but cannot acquire any.

iv) Time Stamp Ordering Protocol:

A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or the data item had been used in any way. A timestamp can be implemented in 2 ways. One is to directly

assign the current value of the clock to the transaction or data item. The other is to attach the value of a logical counter that keeps increment as new timestamps are required.

The timestamp of a data item can be of 2 types:

- **W-timestamp(X):** This means the latest time when the data item X has been written into.
- **R-timestamp(X):** This means the latest time when the data item X has been read from. These 2 timestamps are updated each time a successful read/write operation is performed on the data item X.

iii) Mult version Concurrency Control:

Mult version schemes keep old versions of data item to increase concurrency.
Mult version 2 phase locking: Each successful write results in the creation of a new version of the data item written. Timestamps are used to label the versions. When a read(X) operation is issued, select an appropriate version of X based on the timestamp of the transaction.

iv) Validation Concurrency Control:

The optimistic approach is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through 2 or 3 phases, referred to as read, validation and write.

- During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.
- During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to a write phase. If the validation test is negative, he transaction is restarted and the changes are discarded.
- During the write phase, the changes are permanently applied to the database.

15. Deadlock and Starvation

DEADLOCK

In a database management system (DBMS), a deadlock occurs when two or more transactions are waiting for each other to release resources, such as locks on database objects, that they need to complete their operations. As a result, none of the transactions can proceed, leading to a situation where they are stuck or "deadlocked."

Deadlocks can happen in multi-user environments when two or more transactions are running concurrently and try to access the same data in a different order. When this happens, one transaction may hold a lock on a resource that another transaction needs, while the second transaction may hold a lock on a resource that the first transaction needs. Both transactions are then blocked, waiting for the other to release the resource they need.

DBMSs often use various techniques to detect and resolve deadlocks automatically. These techniques include timeout mechanisms, where a transaction is forced to release its locks after a certain period of time, and deadlock detection algorithms, which periodically scan the transaction log for deadlock cycles and then choose a transaction to abort to resolve the deadlock.

It is also possible to prevent deadlocks by careful design of transactions, such as always acquiring locks in the same order or releasing locks as soon as possible. Proper design of the database schema and application can also help to minimize the likelihood of deadlocks.

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Example – let us understand the concept of Deadlock with an example:
Suppose, Transaction T1 holds a lock on some rows in the students table and **needs to update** some rows in the Grades table. Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the student table **held by Transaction T1**.
Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

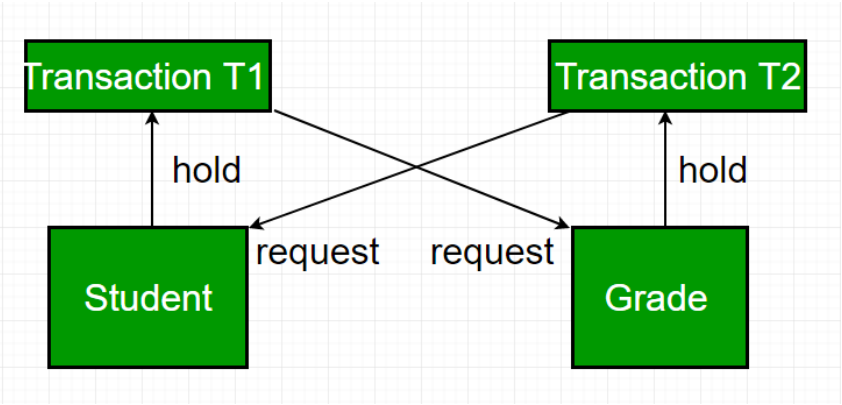


Figure 36: Deadlock in DBMS

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. The deadlock avoidance method is suitable for smaller databases whereas the deadlock prevention method is suitable for larger databases.
One method of avoiding deadlock is using application-consistent logic. In the above-given example, Transactions that access Students and Grades should always access the tables in the same order. In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins. When transaction T2 releases the lock, Transaction T1 can proceed freely.
Another method for avoiding deadlock is to apply both the row-level locking mechanism and the READ COMMITTED isolation level. However, It does not guarantee to remove deadlocks completely.

Deadlock Detection: When a transaction waits indefinitely to obtain a lock, The database management system should detect whether the transaction is involved in a deadlock or not.
Wait-for-graph is one of the methods for detecting the deadlock situation. This method is suitable for smaller databases. In this method, a graph is drawn based on the transaction and its lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.
For the above-mentioned scenario, the Wait-For graph is drawn below:

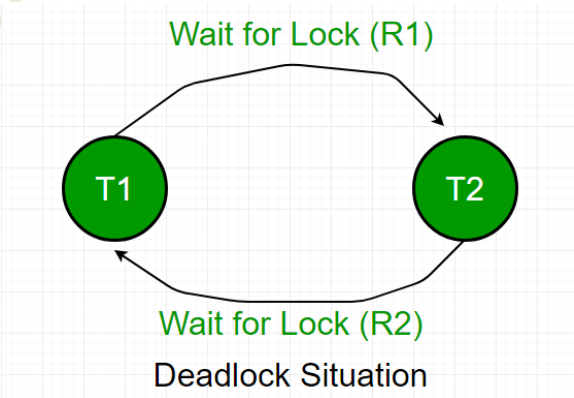


Figure 37: Deadlock Situation

Deadlock prevention: For a large database, the deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that a deadlock never occurs. The DBMS analyses the operations whether they can create a deadlock situation or not, If they do, that transaction is never allowed to be executed.
Deadlock prevention mechanism proposes two schemes:

- **Wait-Die Scheme:**

In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.
Suppose, there are two transactions T1 and T2, and Let the timestamp of any transaction T be TS (T). Now, If there is a lock on T2 by some other transaction and T1 is requesting resources held by T2, then DBMS performs the following actions:
Checks if TS (T1) < TS (T2) – if T1 is the older transaction and T2 has held some resource, then it allows T1 to wait until resource is available for execution. That means if a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available. If T1 is an older transaction and has held some resource with it and if T2 is waiting for it, then T2 is killed and restarted later with random delay but with the same timestamp. i.e. if the older transaction has held some resource and the younger transaction waits for the resource, then the younger transaction is killed and restarted with a very minute delay with the same timestamp.

- **Wound Wait Scheme:**

In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource. The younger transaction is restarted with a minute delay but with the same timestamp. If the younger transaction is requesting a resource that is held by an older one, then the younger transaction is asked to wait till the older one releases it.
The following table lists the differences between Wait – Die and Wound -Wait scheme prevention schemes:

Wait – Die	**Wound -Wait**
It is based on a non-pre-emptive technique.	It is based on a pre-emptive technique.
In this, older transactions must wait for the younger one to release its data items.	In this, older transactions never wait for younger transactions.
The number of aborts and rollbacks is higher in these techniques.	In this, the number of aborts and rollback is lesser.

Applications:

- **Delayed Transactions:** Deadlocks can cause transactions to be delayed, as the resources they need are being held by other transactions. This can lead to slower response times and longer wait times for users.
- **Lost Transactions:** In some cases, deadlocks can cause transactions to be lost or aborted, which can result in data inconsistencies or other issues.
- **Reduced Concurrency:** Deadlocks can reduce the level of concurrency in the system, as transactions are blocked waiting for resources to become available. This can lead to slower transaction processing and reduced overall throughput.
- **Increased Resource Usage:** Deadlocks can result in increased resource usage, as transactions that are blocked waiting for resources to become available continue to consume system resources. This can lead to performance degradation and increased resource contention.
- **Reduced User Satisfaction:** Deadlocks can lead to a perception of poor system performance and can reduce user satisfaction with the application. This can have a negative impact on user adoption and retention.

Features of deadlock in a DBMS:

- **Mutual Exclusion:** Each resource can be held by only one transaction at a time, and other transactions must wait for it to be released.
- **Hold and Wait:** Transactions can request resources while holding on to resources already allocated to them.
- **No Preemption:** Resources cannot be taken away from a transaction forcibly, and the transaction must release them voluntarily.
- **Circular Wait:** Transactions are waiting for resources in a circular chain, where each transaction is waiting for a resource held by the next transaction in the chain.
- **Indefinite Blocking:** Transactions are blocked indefinitely, waiting for resources to become available, and no transaction can proceed.
- **System Stagnation:** Deadlock leads to system stagnation, where no transaction can proceed, and the system is unable to make any progress.
- **Inconsistent Data:** Deadlock can lead to inconsistent data if transactions are unable to complete and leave the database in an intermediate state.
- **Difficult to Detect and Resolve:** Deadlock can be difficult to detect and resolve, as it may involve multiple transactions, resources, and dependencies.

Disadvantages:

- **System downtime:** Deadlock can cause system downtime, which can result in loss of productivity and revenue for businesses that rely on the DBMS.
- **Resource waste:** When transactions are waiting for resources, these resources are not being used, leading to wasted resources and decreased system efficiency.
- **Reduced concurrency:** Deadlock can lead to a decrease in system concurrency, which can result in slower transaction processing and reduced throughput.
- **Complex resolution:** Resolving deadlock can be a complex and time-consuming process, requiring system administrators to intervene and manually resolve the deadlock.
- **Increased system overhead:** The mechanisms used to detect and resolve deadlock, such as timeouts and rollbacks, can increase system overhead, leading to decreased performance.

STARVATION

Starvation or Livelock is the situation when a transaction has to wait for an indefinite period of time to acquire a lock.

Reasons for Starvation:

- If the waiting scheme for locked items is unfair. (priority queue)
- Victim selection (the same transaction is selected as a victim repeatedly)
- Resource leak.
- Via denial-of-service attack.

Starvation can be best explained with the help of an example –

Suppose there are 3 transactions namely T1, T2, and T3 in a database that is trying to acquire a lock on data item ‘ I ‘. Now, suppose the scheduler grants the lock to T1(maybe due to some priority), and the other two transactions are waiting for the lock. As soon as the execution of T1 is over, another transaction T4 also comes over and requests a lock on data item I. Now, this time the scheduler grants lock to T4, and T2, T3 has to wait again. In this way, if new transactions keep on requesting the lock, T2 and T3 may have to wait for an indefinite period of time, which leads to **Starvation**.

Solutions to 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 lock.
- **Wait-die and wound wait scheme:** These are the schemes that use the timestamp ordering mechanism of transactions.
- **Timeout Mechanism:** A timeout mechanism can be implemented in which a transaction is only allowed to wait for a certain amount of time before it is aborted or restarted. This ensures that no transaction waits indefinitely, and prevents the possibility of starvation.
- **Resource Reservation:** A resource reservation scheme can be used to allocate resources to a transaction before it starts execution. This ensures that the transaction has access to the necessary resources and reduces the chances of waiting for a resource indefinitely.
- **Preemption:** Preemption involves the forcible removal of a lock from a transaction that has been waiting for a long time, in favor of another transaction that has a higher priority or has been waiting for a shorter time. Preemption ensures that no transaction waits indefinitely, and prevents the possibility of starvation.
- **Dynamic Lock Allocation:** In this approach, locks are allocated dynamically based on the current state of the system. The system may analyze the current lock requests and allocate locks in such a way that prevents deadlocks and reduces the chances of starvation.
- **Parallelism:** By allowing multiple transactions to execute in parallel, the system can ensure that no transaction waits indefinitely, and reduces the chances of starvation. This approach requires careful consideration of the potential for conflicts and race conditions between transactions.

In a database management system (DBMS), starvation occurs when a transaction or process is not able to get the resources it needs to proceed and is continuously delayed or blocked. This can happen when other transactions or processes are given priority over the one that is experiencing starvation.

In DBMSs, resources such as locks, memory, and CPU time are typically shared among multiple transactions or processes. If some transactions or processes are given priority over others, it is possible for one or more transactions or processes to experience starvation.

For example, if a transaction is waiting for a lock that is held by another transaction, it may be blocked indefinitely if the other transaction never releases the lock. This can lead to the first transaction experiencing starvation if it is continuously blocked and unable to proceed.

DBMSs typically use various techniques to prevent or mitigate starvation, such as:

Resource allocation policies: DBMSs can use policies to allocate resources in a fair manner, ensuring that no transaction or process is consistently given priority over others.

Priority-based scheduling: DBMSs can use scheduling algorithms that take into account the priority of transactions or processes, ensuring that high-priority transactions or processes are executed before low-priority ones.

Timeout mechanisms: DBMSs can use timeout mechanisms to prevent transactions or processes from being blocked indefinitely, by releasing resources if a transaction or process waits for too long.

Resource management: DBMSs can also use techniques such as resource quotas and limits to prevent any single transaction or process from monopolizing resources, thus reducing the likelihood of starvation.

Disadvantages of Starvation:

- **Decreased performance:** Starvation can cause decreased performance in a DBMS by preventing transactions from making progress and causing a bottleneck.
- **Increased response time:** Starvation can increase response time for transactions that are waiting for resources, leading to poor user experience and decreased productivity.
- **Inconsistent data:** If a transaction is unable to complete due to starvation, it may leave the database in an inconsistent state, which can lead to data corruption and other problems.
- **Difficulty in troubleshooting:** Starvation can be difficult to troubleshoot because it may not be immediately apparent which transaction is causing the problem.
- **Potential for deadlock:** If multiple transactions are competing for the same resources, starvation can lead to deadlock, where none of the transactions can proceed, causing a complete system failure.

16. Lock Based Protocol

In a database management system (DBMS), lock-based concurrency control (BCC) is used to control the access of multiple transactions to the same data item. This protocol helps to maintain data consistency and integrity across multiple users. In the protocol, transactions gain locks on data items to control their access and prevent conflicts between concurrent transactions.
First things first, I hope you are familiar with some of the concepts relating to Transaction.

- What is a Recoverable Schedule?
- What are Cascading Rollbacks and Cascadeless schedules?
- Determining if a schedule is Conflict Serializable.

Now, we all know the four properties a transaction must follow. Yes, you got that right, I mean the **ACID** properties. Concurrency control techniques are used to ensure that the *Isolation* (or non-interference) property of concurrently executing transactions is maintained.

A trivial question I would like to pose in front of you, (I know you must know this but still) why you think that we should have interleaving execution of transactions if it may lead to problems such as Irrecoverable Schedule, Inconsistency, and many more threats. Why not just let it be Serial schedules and we may live peacefully, with no complications at all?

Yes, the performance affects the efficiency too much which is not acceptable. Hence a Database may provide a mechanism that ensures that the schedules are either conflict or view serializable and recoverable (also preferably cascadeless). Testing for a schedule for Serializability after it has been executed is *too late!* So, we need Concurrency Control Protocols that ensure Serializability.

Concurrency Control Protocols

allow concurrent schedules but ensure that the schedules are conflict/view serializable and are recoverable and maybe even cascadeless. These protocols do not examine the precedence graph as it is being created, instead a protocol imposes a discipline that avoids non-serializable schedules. Different concurrency control protocols provide different advantages between the amount of concurrency they allow and the amount of overhead that they impose.

Now, let's get going: Different categories of protocols:

- **Lock Based Protocol**
  - Basic 2-PL
  - Conservative 2-PL
  - Strict 2-PL
  - Rigorous 2-PL
- **Graph Based Protocol**
- **Time-Stamp Ordering Protocol**
- **Multiple Granularity Protocol**
- **Multi-version Protocol**

For GATE we'll be focusing on the First three protocols.

Lock Based Protocols

A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it. They synchronize the access by concurrent transactions to the database items. It is required in this protocol that all the data items must be accessed in a mutually exclusive manner. Let me introduce you to two common locks which are used and some terminology followed in this protocol.

- **Shared Lock (S):** also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
- **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Lock Compatibility Matrix:

	S	X
S	✔	✘
X	✘	✘

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

Upgrade / Downgrade locks

A transaction that holds a lock on an item **A**is allowed under certain condition to change the lock state from one state to another. Upgrade: A S(A) can be upgraded to X(A) if Ti is the only transaction holding the S-lock on element A. Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

So, by now we are introduced with the types of locks and how to apply them. But wait, just by applying locks if our problems could have been avoided then life would have been so simple! If you have done Process Synchronization under OS you must be familiar with one consistent problem, starvation, and Deadlock! We will be discussing them shortly, but just so you know we have to apply Locks, but they must follow a set of protocols to avoid such undesirable problems. Shortly we will use 2-Phase Locking (2-PL) which will use the

concept of Locks to avoid deadlock. So, applying simple locking, we may not always produce Serializable results, it may lead to Deadlock Inconsistency.

Problem With Simple Locking

Consider the Partial Schedule:

	T1	**T2**
1	lock-X(B)	
2	read(B)	
3	B:=B-50	
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	

Deadlock

In deadlock consider the above execution phase. Now, **T1** holds an Exclusive lock over B, and **T2** holds a Shared lock over A. Consider Statement 7, **T2** requests for lock on B, while in Statement 8 **T1** requests lock on A. This as you may notice imposes a **Deadlock** as none can proceed with their execution.

Starvation

It is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

17. Two Phase Locking (2-PL)

there are two types of Locks available Shared S(a) and Exclusive X(a). Implementing this lock system without any restrictions gives us the **Simple Lock-based protocol** (or ***Binary Locking***), but it has its own disadvantages, they do not guarantee **Serializability**. Schedules may follow the preceding rules but a non-serializable schedule may result.

To guarantee serializability, we must follow some additional protocols *concerning the positioning of locking and unlocking operations* in every transaction. This is where the concept of **Two-Phase Locking(2-PL)** comes into the picture, 2-PL ensures serializability. Now, let's dig deep!

Two Phase Locking

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

Note:

If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in the shrinking phase.

Let's see a transaction implementing 2-PL.

	T1	**T2**
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	…….	…….
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	…….

This is just a skeleton transaction that shows how unlocking and locking work with 2-PL. Note for:

Transaction T1

- The growing Phase is from steps 1-3
- The shrinking Phase is from steps 5-7
- Lock Point at 3

Transaction T2

- The growing Phase is from steps 2-6
- The shrinking Phase is from steps 8-9
- Lock Point at 6

Lock Point

The Point at which the growing phase ends, i.e., when a transaction takes the final lock, it needs to carry on its work. Now look at the schedule, you'll surely understand. I have said that 2-PL ensures serializability, but there are still some drawbacks of 2-PL. Let us glance at the drawbacks.

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

Categories of two-Phase Locking

Now that we are familiar with what is Two-Phase Locking (2-PL) and the basic rules which should be followed which ensures serializability. Moreover, we came across problems with 2-PL, Cascading Aborts, and Deadlocks. Now, we turn towards the enhancements made on 2-PL which try to make the protocol nearly error-free. Briefly, we allow some modifications to 2-PL to improve it. There are three categories:

- Strict 2-PL
- Rigorous 2-PL
- Conservative 2-PL

Now recall the rules followed in Basic 2-PL, over that we make some extra modifications. Let's now see what are the modifications and what drawbacks they solve.

Strict 2-PL –

This requires that in addition to the lock being 2-Phase **all Exclusive(X) locks** held by the transaction be released until *after* the Transaction Commits. Following Strict 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence, it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still, *Deadlocks are possible*!

Rigorous 2-PL –

This requires that in addition to the lock being 2-Phase **all Exclusive(X) and Shared(S) locks** held by the transaction be released until *after* the Transaction Commits. Following Rigorous 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence, it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still, *Deadlocks are possible*!

Note: *The difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.*

Conservative 2-PL –

A.K.A **Static 2-PL**, this protocol requires the transaction to lock all the items it accesses before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the

transaction does not lock any of the items, instead, it waits until all the items are available for locking.
However, it is difficult to use in practice because of the need to predeclare the read-set and the write-set which is not possible in many situations. In practice, the most popular variation of 2-PL is Strict 2-PL.
Venn Diagram below shows the classification of schedules that are rigorous and strict. The universe represents the schedules that can be serialized as 2-PL. Now as the diagram suggests, and it can also be logically concluded, if a schedule is Rigorous then it is Strict. We can also think in another way, say we put a restriction on a schedule which makes it strict, adding another to the list of restrictions make it Rigorous. Take a moment to again analyze the diagram and you'll definitely get it.

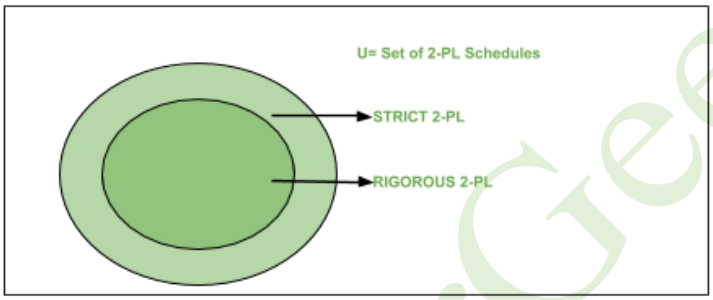


Figure 38: Conservative

Image – Venn Diagram showing categories of languages under 2-PL
Now, let's see the schedule below, tell me if this schedule can be locked using 2-PL, and if yes, show how and what class of 2-PL does your answer belongs to.

	T_1	**T_2**
1	Read(A)	
2		Read(A)
3	Read(B)	
4	Write(B)	
5	Commit	
6		Read(B)
7		Write(B)
6		Commit

Yes, the schedule is conflict serializable, so we can try implementing 2-PL. So, let's try…

Solution:

	T_1	T_2
1	Lock-S(A)	
2	Read(A)	
3		Lock-S(A)
4		Read(A)
5	Lock-X(B)	
6	Read(B)	
7	Write(B)	
8	Commit	
9	Unlock(A)	
10	Unlock(B)	
11		Lock-X(B)
12		Read(B)
13		Write(B)
14		Commit
15		Unlock(A)
16		Unlock(B)

Now, this is one way I choose to implement the locks on A and B. You may try a different sequence but remember to follow the 2-PL protocol. With that said, observe that our locks are released after Commit operation so this satisfies Strict 2-PL protocol.

By now, I guess you must've got the idea of how to differentiate between types of 2-PL. Remember the theory as problems come in the examination sometimes just based on theoretical knowledge. Next, we'll look at some examples of Conservative 2-PL and how does it differ from the above two types of 2-PL. What makes it Deadlock free and also so difficult to implement. Then we'll conclude the topic of 2-PL. Shortly we'll move on to another type of Lock-based Protocol- Graph-Based Protocols.

18.Timestamp Ordering Protocol

Concurrency Control can be implemented in different ways. One way to implement it is by using Locks. Now, let us discuss Time Stamp Ordering **Protocol.**

As earlier introduced, **Timestamp** is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Refer to the timestamp of a transaction *T* as ***TS(T)***.

Timestamp Ordering Protocol –

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. An algorithm must ensure that, for each item accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering –

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that $\mathbf{TS(T_i) < TS(T_j)}$. The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in the following two cases.

- Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
    - If ***R_TS(X) > TS(T)*** and if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
- Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:

- If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
- If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects two conflicting operations that occur in an incorrect order, it rejects the latter of the two operations by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp can ensure that our schedule will be *deadlock free*.

One drawback of the Basic TO protocol is that **Cascading Rollback** is still possible. Suppose we have a Transaction T_1 and T_2 has used a value written by T_1. If T_1 is aborted and resubmitted to the system then, T_2 must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializability since the precedence graph will be of the form:

Figure 39: Precedence Graph for TS ordering

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may *not be cascade free*, and may not even be recoverable.

Strict Timestamp Ordering –

A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.

Advantages:

- **High Concurrency:** Timestamp-based concurrency control allows for a high degree of concurrency by ensuring that transactions do not interfere with each other.
- **Efficient:** The technique is efficient and scalable, as it does not require locking and can handle a large number of transactions.
- **No Deadlocks:** Since there are no locks involved, there is no possibility of deadlocks occurring.
- **Improved Performance:** By allowing transactions to execute concurrently, the overall performance of the database system can be improved.

Disadvantages:

- **Limited Granularity:** The granularity of timestamp-based concurrency control is limited to the precision of the timestamp. This can lead to situations where transactions are unnecessarily blocked, even if they do not conflict with each other.
- **Timestamp Ordering:** In order to ensure that transactions are executed in the correct order, the timestamps need to be carefully managed. If not managed properly, it can lead to inconsistencies in the database.
- **Timestamp Synchronization:** Timestamp-based concurrency control requires that all transactions have synchronized clocks. If the clocks are not synchronized, it can lead to incorrect ordering of transactions.
- **Timestamp Allocation:** Allocating unique timestamps for each transaction can be challenging, especially in distributed systems where transactions may be initiated at different locations.

Timestamp and Deadlock Prevention schemes

Deadlock occurs when each transaction **T** in a schedule of *two or more* transactions waiting for some item locked by some other transaction **T'** in the set. Thus, both end up in a deadlock situation, waiting for the other to release the lock on the item. Deadlocks are a common problem and we have introduced the problem while solving the Concurrency Control by the introduction of Locks. Deadlock avoidance is a major issue and some protocols were suggested to avoid them, like Conservative 2-PL and Graph-Based protocols but some drawbacks are still there.

Here, we will discuss a new concept of **Transaction Timestamp TS(T_i)**. A timestamp is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system, so a timestamp may be thought of as the transaction start time.

There may be different ways of generating timestamps such as

- A simple counter that increments each time its value is assigned to a transaction. They may be numbered ***1, 2, 3….***. Though we'll have to reset the counter from time to time to avoid overflow.
- Using the current date/time from the system clock. Just ensuring that no two transactions are given the same value in the same clock tick, we will always get a unique timestamp. This method is widely used.

Deadlock Prevention Schemes based on Timestamp:
As discussed, Timestamps are unique identifiers assigned to each transaction. They are based on the order in which Transactions are started. Say if $\mathbf{T_1}$ starts before $\mathbf{T_2}$ then $\mathbf{TS(T_1)}$ will be less than (<) $\mathbf{TS(T_2)}$.
There are two schemes to prevent deadlock called *wound-wait* and *wait-die*. Say there are two transactions $\mathbf{T_i}$ and $\mathbf{T_j}$, now say $\mathbf{T_i}$ tries to lock an item *X* but item *X* is already locked by some $\mathbf{T_j}$, now in such a conflicting situation the two schemes which prevent deadlock. We'll use this context shortly.

- **Wait_Die:** An older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
  From the context above, if $\mathbf{TS(T_i) < TS(T_j)}$, then ($T_i$ older than T_j) T_i is allowed to wait; otherwise *abort* T_i (T_i younger than T_j) *and restart it later with the same timestamp.*
- **Wound_Wait:** It is just the opposite of the Wait_Die technique. Here, a younger transaction is allowed to wait for an older one, whereas if an older transaction requests an item held by the younger transaction, we preempt the younger transaction by aborting it.
  From the context above, if $\mathbf{TS(T_i) < TS(T_j)}$, then ($T_i$ older than T_j) T_j is aborted (i.e., T_i wounds T_j) and restarts it later with the same Timestamp; *otherwise (T_i younger than T_j) T_i is allowed to wait.*

Thus, both schemes end up aborting the younger of the two transactions that may be involved in a deadlock. It is done on the basis of the assumption that aborting the younger transaction will waste less processing which is logical. In such a case there cannot be a cycle since we are waiting linearly in both cases.

Another group of protocols prevents deadlock but *does not require Timestamps*. They are discussed below:

- **No-waiting Algorithm:** This follows a simple approach, if a Transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking if a deadlock will occur or not. Here, no Transaction ever waits so there is no possibility for deadlock.
  This method is somewhat not practical. It may cause the transaction to abort and restart unnecessarily.

- **Cautious Waiting:** If $\mathbf{T_i}$ tries to lock an item *X* but is not able to do because *X* is locked by some $\mathbf{T_j}$. In such a conflict, if $\mathbf{T_j}$ is not waiting for some other locked item, then $\mathbf{T_i}$ is allowed to wait, otherwise, *abort* T_i.

Another approach, to deal with deadlock is deadlock detection, we can use Wait-for-Graph. This uses a similar approach when we used to check for cycles while checking for serializability.

Starvation: One problem that may occur when we use locking is starvation which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. We may have some solutions for Starvation. One is using a **first come first serve** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. This is a widely used mechanism to reduce starvation. Our Concurrency Control Manager is responsible to schedule the transactions, so it employs different methods to overcome them.

Timestamp-based concurrency control and deadlock prevention schemes are two important techniques used in database management systems to ensure transaction correctness and concurrency.

In addition, timestamp-based schemes use a validation procedure to check whether a transaction has read data that has been modified by another transaction after the first transaction has read it. If such a conflict is detected, the transaction is rolled back to ensure consistency and correctness.

Deadlock prevention schemes are used to prevent situations where two or more transactions are waiting for each other to release locks, resulting in a deadlock. Deadlocks occur when two transactions hold exclusive locks on resources that the other transaction needs to proceed. To prevent deadlocks, DBMSs use several schemes, including:

Timeout-based schemes: Transactions are allowed to hold a lock for a limited time. If a transaction exceeds its allotted time, it is rolled back, allowing other transactions to proceed.

Wait-die schemes: If a transaction requests a lock held by another transaction, the requesting transaction waits if its timestamp is older than the timestamp of the transaction holding the lock. If the timestamp of the requesting transaction is newer, it rolls back and is restarted with a new timestamp.

Wound-wait schemes: If a transaction requests a lock held by another transaction, the requesting transaction is granted the lock if its timestamp is newer than the timestamp of the transaction holding the lock. If the timestamp of the requesting transaction is older, the transaction holding the lock is rolled back and restarted with a new timestamp.

19. File Organization in DBMS

A database consists of a huge amount of data. The data is grouped within a table in RDBMS, and each table has related records. A user can see that the data is stored in the form of tables, but this huge amount of data is stored in physical memory in the form of files.

What is a File?
A file is named a collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes, and optical disks.

What is File Organization?
File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in a certain order is called File Organization. **File Structure** refers to the format of the label and data blocks and of any logical control record.

The Objective of File Organization

- It helps in the faster selection of records i.e. it makes the process faster.
- Different Operations like inserting, deleting, and updating different records are faster and easier.
- It prevents us from inserting duplicate records via various operations.
- It helps in storing the records or the data very efficiently at a minimal cost.

19.1. Types of File Organization

Types of File Organizations
Various methods have been introduced to Organize files. These methods have advantages and disadvantages on the basis of access or selection. Thus it is all upon the programmer to decide the best-suited file Organization method according to his requirements.
Some types of File Organizations are:

- Sequential File Organization
- Heap File Organization
- Hash File Organization
- B+ Tree File Organization
- Clustered File Organization
- ISAM (Indexed Sequential Access Method)

We will be discussing each of the file Organizations in further sets of this article along with the differences and advantages/ disadvantages of each file Organization method.

1) Sequential File Organization

The easiest method for file Organization is the Sequential method. In this method, the file is stored one after another in a sequential manner. There are two ways to implement this method:

- **Pile File Method**

This method is quite simple, in which we store the records in a sequence i.e. one after the other in the order in which they are inserted into the tables.

Insertion of the new record: Let the R1, R3, and so on up to R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, then it is simply placed at the end of the file.

- **Sorted File Method**

In this method, As the name itself suggests whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. The sorting of records may be based on any primary key or any other key.

Insertion of the new record: Let us assume that there is a preexisting sorted sequence of four records R1, R3, and so on up to R7 and R8. Suppose a new record R2 must be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence.

Advantages of Sequential File Organization

- Fast and efficient method for huge amounts of data.
- Simple design.
- Files can be easily stored in magnetic tapes i.e. cheaper storage mechanism.

Disadvantages of Sequential File Organization

- Time wastage as we cannot jump on a particular record that is required, but we have to move in a sequential manner which takes our time.
- The sorted file method is inefficient as it takes time and space for sorting records.

2) Heap File Organization

Heap File Organization works with data blocks. In this method, records are inserted at the end of the file, into the data blocks. No Sorting or Ordering is required in this method. If a data block is full, the new record is stored in some other block,

Here the other data block need not be the very next data block, but it can be any block in the memory. It is the responsibility of DBMS to store and manage the new records.

Insertion of the new record: Suppose we have four records in the heap R1, R5, R6, R4, and R3, and suppose a new record R2 has to be inserted in the heap then, since the last data block i.e data block 3 is full it will be inserted in any of the data blocks selected by the DBMS, let's say data block 1.

If we want to search, delete or update data in the heap file Organization we will traverse the data from the beginning of the file till we get the requested record. Thus if the database is very huge, searching, deleting, or updating the record will take a lot of time.

Advantages of Heap File Organization

- Fetching and retrieving records is faster than sequential records but only in the case of small databases.
- When there is a huge number of data that needs to be loaded into the database at a time, then this method of file Organization is best suited.

Disadvantages of Heap File Organization

- The problem of unused memory blocks.
- Inefficient for larger databases.

19.2. Hashing in DBMS

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database. For larger databases containing thousands and millions of records, the indexing data structure technique becomes very inefficient because searching a specific record through indexing will consume more time. This doesn't align with the goals of DBMS, especially when performance and date retrieval time are minimized. So, to counter this problem hashing technique is used. In this article, we will learn about various hashing techniques.

What is Hashing?

The hashing technique utilizes an auxiliary hash table to store the data records using a hash function. There are 2 key components in hashing:

- **Hash Table:** A hash table is an array or data structure, and its size is determined by the total volume of data records present in the database. Each memory location in a hash table is called a '***bucket***' or hash indice and stores a data record's exact location and can be accessed through a hash function.
- **Bucket:** A bucket is a memory location (index) in the hash table that stores the data record. These buckets generally store a disk block which further stores multiple records. It is also known as the hash index.

- **Hash Function:** A hash function is a mathematical equation or algorithm that takes one data record's primary key as input and computes the hash index as output.

Hash Function

A hash function is a mathematical algorithm that computes the index or the location where the current data record is to be stored in the hash table so that it can be accessed efficiently later. This hash function is the most crucial component that determines the speed of fetching data.

Working of Hash Function

The hash function generates a hash index through the primary key of the data record.

Now, there are 2 possibilities:

1. The hash index generated isn't already occupied by any other value. So, the address of the data record will be stored here.
2. The hash index generated is already occupied by some other value. This is called collision so to counter this, a collision resolution technique will be applied.
3. Now whenever we query a specific record, the hash function will be applied and returns the data record comparatively faster than indexing because we can directly reach the exact location of the data record through the hash function rather than searching through indices one by one.

Example:

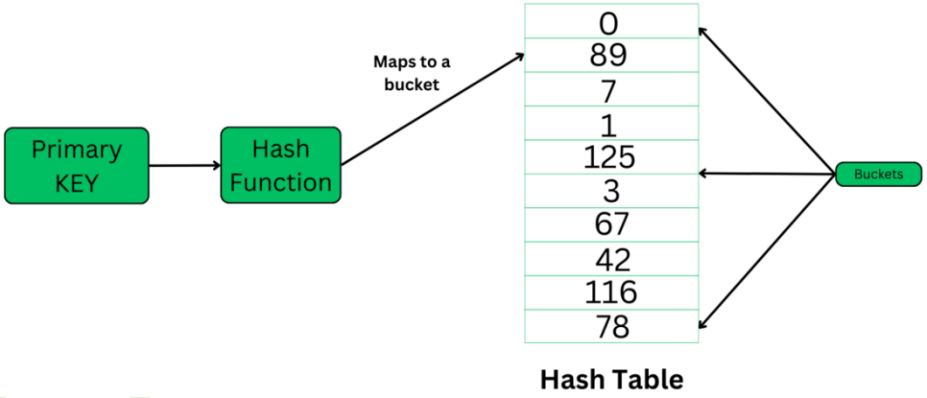


Figure 40: Hashing

Types of Hashing in DBMS

There are two primary hashing techniques in DBMS.

1) Static Hashing

In static hashing, the hash function always generates the same bucket's address. For example, if we have a data record for employee_id = 107, the hash function is mod-5 which is – H(x) % 5, where x = id. Then the operation will take place like this:

H(106) % 5 = 1.
This indicates that the data record should be placed or searched in the 1st bucket (or 1st hash index) in the hash table.
Example:

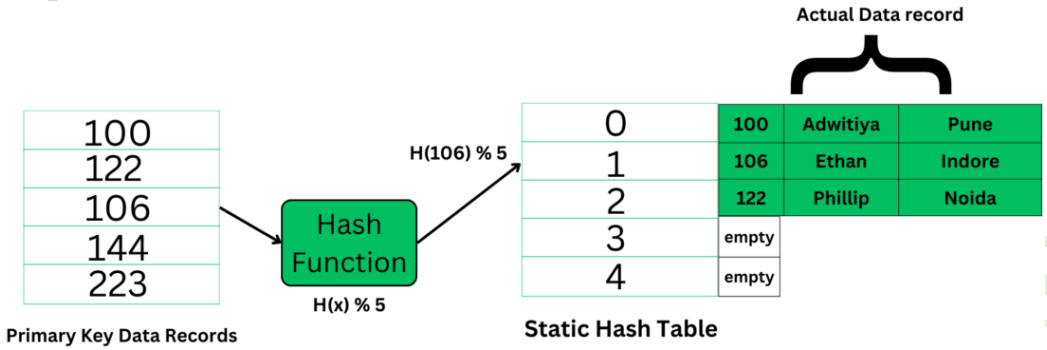


Figure 41: Static Hashing Technique

The primary key is used as the input to the hash function and the hash function generates the output as the hash index (bucket's address) which contains the address of the actual data record on the disk block.
Static Hashing has the following Properties.

- **Data Buckets:** The number of buckets in memory remains constant. The size of the hash table is decided initially and it may also implement chaining that will allow handling some collision issues though, it's only a slight optimization and may not prove worthy if the database size keeps fluctuating.
- **Hash function:** It uses the simplest hash function to map the data records to its appropriate bucket. It is generally modulo-hash function
- **Efficient for known data size:** It's very efficient in terms when we know the data size and its distribution in the database.
- It is inefficient and inaccurate when the data size dynamically varies because we have limited space and the hash function always generates the same value for every specific input. When the data size fluctuates very often it's not at all useful because collision will keep happening and it will result in problems like – bucket skew, insufficient buckets etc.

To resolve this problem of bucket overflow, techniques such as – chaining and open addressing are used. Here's a brief info on both:

- **Chaining**

Chaining is a mechanism in which the hash table is implemented using an array of type nodes, where each bucket is of node type and can contain a long chain of linked lists to store the data records. So, even if a hash function generates the same value for any data record it can still be stored in a bucket by adding a new node.

However, this will give rise to the problem bucket skew that is, if the hash function keeps generating the same value again and again then the hashing will become inefficient as the remaining data buckets will stay unoccupied or store minimal data.

- **Open Addressing/Closed Hashing**

This is also called closed hashing this aims to solve the problem of collision by looking out for the next empty slot available which can store data. It uses techniques like **linear probing**, **quadratic probing**, **double hashing,** etc.

2) Dynamic Hashing

Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

Properties of Dynamic Hashing

- The buckets will vary in size dynamically periodically as changes are made offering more flexibility in making any change.
- Dynamic Hashing aids in improving overall performance by minimizing or completely preventing collisions.
- **It has the following major components:** Data bucket, Flexible hash function, and directories
- A flexible hash function means that it will generate more dynamic values and will keep changing periodically asserting to the requirements of the database.
- Directories are containers that store the pointer to buckets. If bucket overflow or bucket skew-like problems happen to occur, then bucket splitting is done to maintain efficient retrieval time of data records. Each directory will have a directory id.
- **Global Depth:** It is defined as the number of bits in each directory id. The more the number of records, the more bits are there.

Working of Dynamic Hashing

Example: If global depth: k = 2, the keys will be mapped accordingly to the hash index. K bits starting from LSB will be taken to map a key to the buckets. That leave
s us with the following 4 possibilities: 00, 11, 10, 01.

19.3. All about B-Tree

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as "large key" trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

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

Note: "n" is the total number of elements in the B-tree

Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t – 1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

- Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of a B-Tree of minimum order 5
Note: *that in practical B-Trees, the value of the minimum order is much more than* 5.

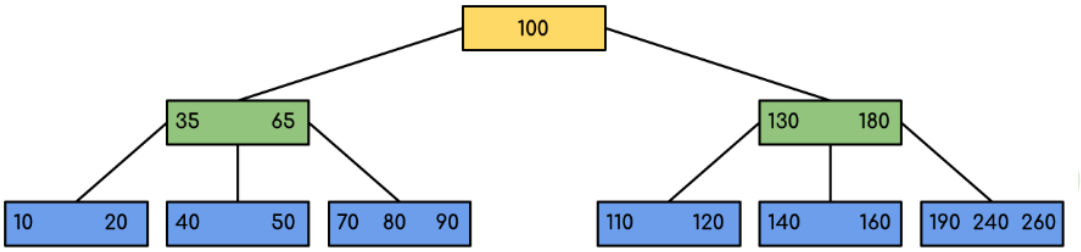


Figure 42: B-Tree

We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

Traversal in B-Tree:
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:
Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

- Start from the root and recursively traverse down.
- For every visited non-leaf node,
  - If the node has the key, we simply return the node.
  - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

Applications of B-Trees:

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures.

Insertion Operation in B-Tree

A new key is always inserted at the leaf node. Let the key to be inserted be k. Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.

How to make sure that a node has space available for a key before the key is inserted? We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up, unlike BSTs which grow down.

Insertion
1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.
Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if a new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from the root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because the parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

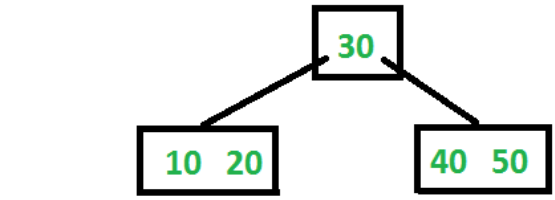
Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.
Initially root is NULL. Let us first insert 10.

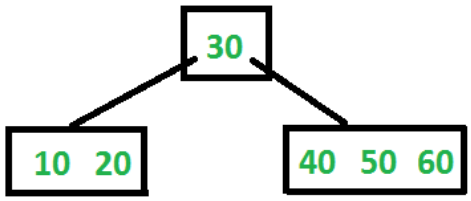
Insert 10

Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Insert 60

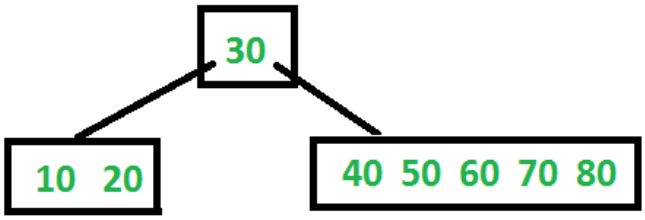




Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

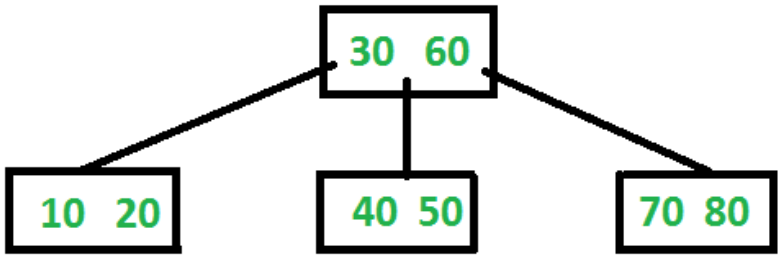
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



Delete Operation

B Trees is a type of data structure commonly known as a Balanced Tree that stores multiple data items very easily. B Trees are one of the most useful data structures that provide ordered access to the data in the database. In this article, we will see the delete operation in the B-Tree. B-Trees are self-balancing trees.

Deletion Process in B-Trees

Deletion from a B-tree is more complicated than insertion because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number t-1 of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

Various Cases of Deletion

Case 1: If the key k is in node x and x is a leaf, delete the key k from x.

Case 2: If the key k is in node x and x is an internal node, do the following.

- If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k with k0 in x. (We can find k0 and delete it in a single downward pass.)
- If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k with k0 in x. (We can find k0 and delete it in a single downward pass.)
- Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

Case 3: If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i)

has only t-1 keys, execute steps 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

- If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).
- If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

19.4. All about B+ Tree

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a **B+ tree structure** of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree is linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree is used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Features of B+ Trees

- **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
- **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
- **Ordered:** B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
- **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.

- **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.
- **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

Why Use B+ Tree?

- B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
- B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Difference Between B+ Tree and B Tree

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values
Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use
Performance	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

Implementation of B+ Tree

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, 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. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, to access them.
Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore, form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

Structure of B+ Tree
B+ Trees contain two types of nodes:

- **Internal Nodes:** Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- **Leaf Nodes:** Leaf Nodes are the nodes that have n pointers.

Advantages of B+Trees

- B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
- Data stored in a B+ tree can be accessed both sequentially and directly.
- It takes an equal number of disk accesses to fetch records.
- B+trees have redundant search keys, and storing search keys repeatedly is not possible.

Disadvantages of B+ Trees

- The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

Application of B+ Trees

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

Insertion in B+ Tree

During insertion following properties of **B+ Tree** must be followed:

- Each node except root can have a maximum of **M** children and at least **ceil(M/2)** children.
- Each node can contain a maximum of **M – 1** keys and a minimum of **ceil(M/2) – 1** keys.
- The root has at least two children and atleast one search key.
- While insertion overflow of the node occurs when it contains more than **M – 1** search key values.

Here **M** is the order of B+ tree.

Steps for insertion in B+ Tree

Step 1. Every element is inserted into a leaf node. So, go to the appropriate leaf node.

Step 2. Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow go ahead with the following steps mentioned below to deal with overflow while maintaining the B+ Tree properties.

Properties for insertion B+ Tree

Case 1: Overflow in leaf node

- Split the leaf node into two nodes.
- First node contains **ceil((m-1)/2)** values.
- Second node contains the remaining values.
- Copy the smallest search key value from second node to the parent node.(Right biased)

Below is the illustration of inserting 8 into B+ Tree of order of 5:

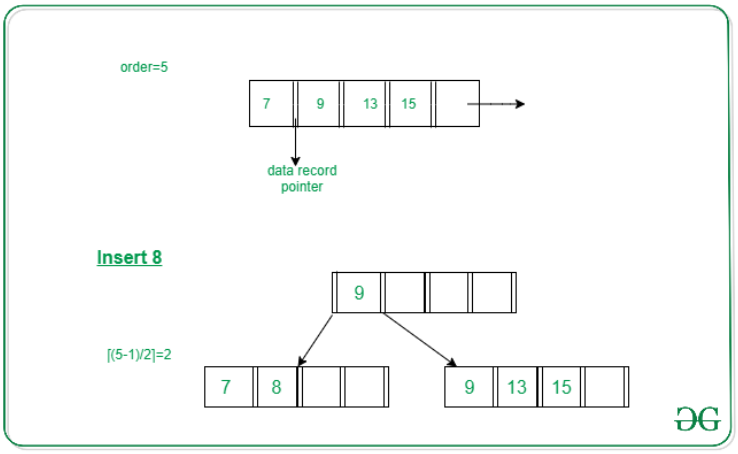


Figure 43: Insertion in B+ Tree

Case 2: Overflow in non-leaf node

- Split the non-leaf node into two nodes.
- First node contains ceil(m/2)-1 values.
- Move the smallest among remaining to the parent.
- Second node contains the remaining keys.

Below is the illustration of inserting 15 into B+ Tree of order of 5:

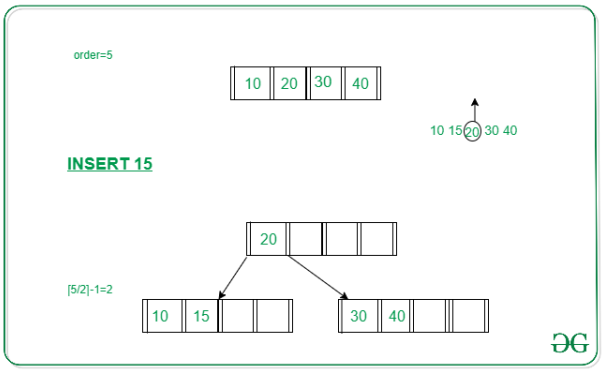


Figure 44: Insertion in B+ Tree

Difference between B Tree and B+ Tree

Basis of Comparison	B tree	B+ tree
Pointers	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
Search	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and more accurate.
Redundant Keys	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at the leaf.
Insertion	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
Deletion	Deletion of the internal node is very complex, and the tree has to undergo a lot of transformations.	Deletion of any node is easy because all nodes are found at leaf.
Leaf Nodes	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
Access	Sequential access to nodes is not possible	Sequential access is possible just like linked list
Height	For a particular number nodes height is larger	Height is lesser than B tree for the same number of nodes
Application	B-Trees used in Databases, Search engines	B+ Trees used in Multilevel Indexing, Database indexing
Number of Nodes	Number of nodes at any intermediary level ‘l’ is 2^l.	Each intermediary node can have n/2 to n children.