

UNIT-1

Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

What is a Database?

To find out what database is, we have to start from data, which is the basic building block of any DBMS.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).

Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	Name	Age
1	ABC	19

Table or Relation: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called **Fields, Attributes or Domains**. The rows are called **Tuples or Records**.

Database: Collection of related relations. Consider the following collection of tables:

T1

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3

Roll	Year
1	I
2	II
3	I

—

T4

Year	Hostel
I	H1
II	H2

We now have a collection of 4 tables. They can be called a “related collection” because we can clearly find out that there are some common attributes existing in a selected pair of tables. Because of these common attributes we may combine the data of two or more tables together to find out the complete details of a student. Questions like “Which hostel does the youngest student live in?” can be answered now, although

Age and **Hostel** attributes are in different tables.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

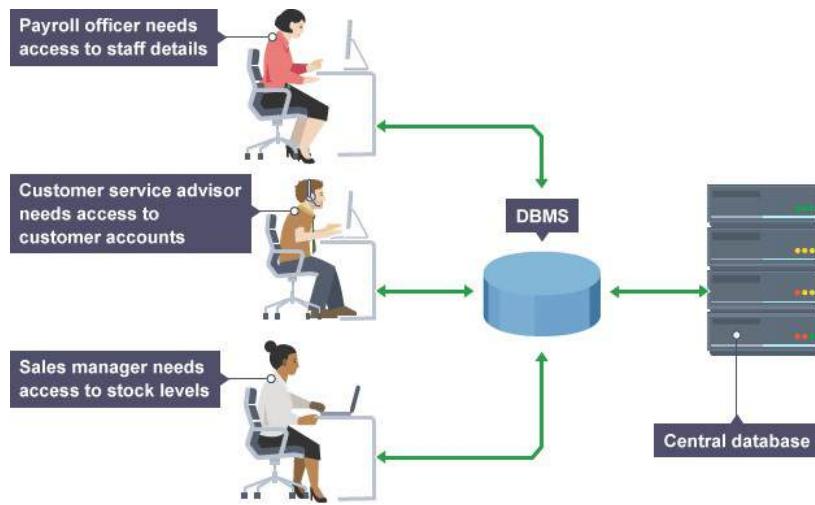


Figure 1.1: Employees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

In a database, data is organized strictly in row and column format. The rows are called **Tuple** or **Record**. The data items within one row may belong to different data types. On the other hand, the columns are often called **Domain** or **Attribute**. All the data items within a single attribute are of the same data type.

What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data**, we mean known facts that can be recorded and that have implicit meaning.

The management system is important because without the existence of some kind of rules and regulations it is not possible to maintain the database. We have to select the particular attributes which should be included in a particular table; the common attributes to create relationship between two tables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must be resolved by having some kind of rules to follow in order to maintain the integrity of the database.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This

chapter briefly introduces the principles of database systems.

Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow he users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources

Enterprise Information

- *Sales*: For customer, product, and purchase information.
- *Accounting*: For payments, receipts, account balances, assets and other accounting information.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
- *Online retailers*: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

Banking and Finance

- *Banking*: For customer information, accounts, loans, and banking transactions.
- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- *Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- ✓ Add new students, instructors, and courses
- ✓ Register students for courses and generate class rosters
- ✓ Assign grades to students, compute grade point averages (GPA), and generate transcripts

FILE PROCESSING SYSTEM

A file processing system is a traditional method of organizing and managing data where information is stored in individual files. Each file typically contains related data and may be stored in a hierarchical directory structure. File processing systems predate modern database management systems (DBMS) and were commonly used in early computer systems.

key characteristics and components of a file processing system:

1. **Files:** Data is stored in individual files, which may be organized hierarchically within directories. Each file contains a specific set of related data.
2. **Data Redundancy:** Data redundancy is common in file processing systems since the same data may be stored in multiple files. This redundancy can lead to inconsistencies and inefficiencies in data management.
3. **Data Independence:** In file processing systems, data is often tightly coupled with the applications that use it. Changes to the structure of the data require corresponding modifications to the application code, leading to a lack of data independence.
4. **Limited Data Sharing:** File processing systems typically lack mechanisms for sharing data between different applications or users. Data is often accessible only to the application that created it, making it challenging to integrate data across multiple systems.
5. **Sequential Access:** Access to data in file processing systems is often sequential, meaning that data must be read or written sequentially from the beginning of the file to the desired location. This can result in slow performance, especially for large datasets.
6. **Lack of Data Integrity:** Without built-in mechanisms for enforcing data integrity constraints, such as referential integrity or data validation rules, file processing systems are prone to data inconsistencies and errors.
7. **Limited Security:** File processing systems typically provide limited security features, such as basic file permissions. This can make it challenging to control access to sensitive data and protect against unauthorized access.

Limitations of file processing systems:

1. **Data Redundancy and Inconsistency:** In file processing systems, data redundancy is common because the same data may be stored in multiple files. This redundancy can lead to inconsistencies if updates are not properly propagated across all relevant files. For example, if a customer changes their address, this change would need to be made in every file containing their information, increasing the risk of errors and inconsistencies.
2. **Limited Data Sharing and Accessibility:** File processing systems typically lack mechanisms for sharing data between different applications or users. Each application may have its own set of files, making it difficult to integrate data across multiple systems. This lack of data sharing can result in data silos and hinder collaboration between departments or teams.
3. **Data Dependence:** In file processing systems, data is often tightly coupled with the applications that use it. Changes to the structure of the data require corresponding

modifications to the application code, leading to a lack of data independence. This can make it challenging to adapt to evolving business requirements or integrate new technologies.

4. **Difficulty in Data Retrieval:** Accessing data in file processing systems often requires navigating through hierarchical directory structures and reading files sequentially. This sequential access method can be inefficient, especially for large datasets, leading to slower performance compared to databases with indexed access.
5. **Limited Data Integrity and Security:** File processing systems typically lack built-in mechanisms for enforcing data integrity constraints, such as referential integrity or data validation rules. Without these safeguards, the risk of data corruption or unauthorized access is higher. Additionally, file permissions may be limited in their ability to control access to sensitive data.
6. **Scalability and Maintenance Challenges:** As data volume and complexity increase, maintaining and scaling file processing systems becomes more challenging. Adding new data or modifying existing data structures may require significant manual effort and can be error-prone. Additionally, file system maintenance tasks, such as data backup and recovery, can be cumbersome and time-consuming.
7. **Lack of Concurrency Control:** File processing systems typically lack built-in mechanisms for managing concurrent access to data by multiple users or applications. This can lead to data inconsistency and conflicts when multiple users attempt to modify the same data simultaneously.
8. **Limited Query Capabilities:** File processing systems do not typically offer advanced query capabilities or optimization features. Analyzing data or generating complex reports may require custom programming and manual data manipulation, leading to inefficiencies and increased development time.

Database Management System (DBMS)

A Database Management System (DBMS) is a software system that enables users to interact with a database. It provides an interface for creating, managing, and accessing databases while ensuring the integrity, security, and efficiency of data storage and retrieval.

components and functions of a DBMS:

1. **Data Definition:** The DBMS allows users to define the structure of the database, including specifying data types, relationships between data elements, and constraints to enforce data integrity.
2. **Data Manipulation:** Users can insert, update, delete, and retrieve data from the database using query languages such as SQL (Structured Query Language) or through graphical interfaces provided by the DBMS.
3. **Data Security:** DBMSs offer features for controlling access to the database, including user authentication, authorization, and encryption to protect sensitive data from unauthorized access or modification.
4. **Concurrency Control:** DBMSs manage access to data by multiple users or applications concurrently, ensuring that transactions are executed reliably and without interference.

5. **Data Integrity:** DBMSs enforce data integrity constraints, such as primary keys, foreign keys, and unique constraints, to maintain the consistency and accuracy of data stored in the database.
6. **Data Backup and Recovery:** DBMSs provide mechanisms for backing up and restoring data to prevent data loss in the event of hardware failure, human error, or other disasters.
7. **Query Optimization:** DBMSs optimize query execution by analyzing query plans, indexing data, and caching frequently accessed data to improve performance and efficiency.
8. **Transaction Management:** DBMSs support transactions, which are units of work that must be executed atomically (i.e., all or none), consistently, isolated from other transactions, and durably stored in the database.
9. **Replication and High Availability:** Some DBMSs offer features for replicating data across multiple servers and ensuring high availability to minimize downtime and improve fault tolerance.
10. **Scalability:** DBMSs are designed to scale to handle large volumes of data and high numbers of concurrent users or transactions by supporting distributed architectures and horizontal scaling.

Examples of popular DBMSs include:

- **Oracle Database:** A comprehensive relational database management system known for its scalability, security, and advanced features.
- **MySQL:** An open-source relational database management system that is widely used for web applications and small to medium-sized databases.
- **Microsoft SQL Server:** A relational database management system developed by Microsoft, commonly used in enterprise environments.
- **PostgreSQL:** An open-source relational database management system known for its extensibility, standards compliance, and advanced features.
- **MongoDB:** A NoSQL database management system that stores data in flexible, JSON-like documents and is popular for handling unstructured or semi-structured data.

Advantages of DBMS:

Database Management Systems (DBMS) offer numerous advantages over traditional file processing systems.

1. **Data Integration and Centralization:** DBMSs allow for the centralization of data storage, enabling organizations to store all their data in one location. This facilitates data integration and ensures that all users and applications have access to a consistent and up-to-date view of the data.

2. Data Consistency and Integrity: DBMSs enforce data integrity constraints, such as primary keys, foreign keys, and unique constraints, to maintain the consistency and accuracy of data stored in the database. This helps prevent data duplication, inconsistencies, and errors.
3. Data Security: DBMSs provide robust security features, including user authentication, authorization, and encryption, to protect sensitive data from unauthorized access or modification. Access control mechanisms ensure that only authorized users can view or modify specific data.
4. Concurrent Access and Transaction Management: DBMSs support concurrent access to data by multiple users or applications while ensuring data consistency and integrity. Transaction management features ensure that transactions are executed reliably and that data changes are either committed or rolled back atomically.
5. Data Sharing and Accessibility: DBMSs enable easy sharing of data between different users, departments, or applications. This promotes collaboration and allows organizations to leverage their data assets more effectively.
6. Query Capabilities and Performance Optimization: DBMSs offer powerful query languages, such as SQL (Structured Query Language), and optimization techniques to retrieve and manipulate data efficiently. Indexing, caching, and query optimization algorithms improve query performance and responsiveness.
7. Scalability: DBMSs are designed to scale to handle large volumes of data and high numbers of concurrent users or transactions. They support distributed architectures and horizontal scaling to accommodate growing data storage and processing requirements.
8. Data Backup and Recovery: DBMSs provide mechanisms for backing up and restoring data to prevent data loss in the event of hardware failure, human error, or other disasters. Automated backup schedules and recovery procedures ensure data durability and availability.
9. Data Independence: DBMSs separate the physical storage of data from the applications that use it, providing a layer of abstraction known as data independence. This allows for easier application development, maintenance, and evolution, as changes to the database schema do not require corresponding modifications to application code.
10. Support for Data Analysis and Reporting: DBMSs offer features for data analysis, reporting, and business intelligence, such as OLAP (Online Analytical Processing) and data mining

capabilities. These features enable organizations to gain insights from their data and make informed business decisions.

Difference between File System and DBMS

Basics	File System	DBMS
Structure	The file system is a way of arranging the files in a storage medium within a computer.	DBMS is software for managing the database.
Data Redundancy	Redundant data can be present in a file system.	In DBMS there is no redundant data.
Backup and Recovery	It doesn't provide Inbuilt mechanism for backup and recovery of data if it is lost.	It provides in house tools for backup and recovery of data even if it is lost.
Query processing	There is no efficient query processing in the file system.	Efficient query processing is there in DBMS.
Consistency	There is less data consistency in the file system.	There is more data consistency because of the process of normalization.
Complexity	It is less complex as compared to DBMS.	It has more complexity in handling as compared to the file system.
Security Constraints	File systems provide less security in comparison to DBMS.	DBMS has more security mechanisms as compared to file systems.
Cost	It is less expensive than DBMS.	It has a comparatively higher cost than a file system.
Data Independence	There is no data independence.	In DBMS data independence exists, mainly of two types:

Basics	File System	DBMS
		1) Logical Data Independence. 2) Physical Data Independence.
User Access	Only one user can access data at a time.	Multiple users can access data at a time.
Meaning	The users are not required to write procedures.	The user has to write procedures for managing databases
Sharing	Data is distributed in many files. So, it is not easy to share data.	Due to centralized nature data sharing is easy
Data Abstraction	It give details of storage and representation of data	It hides the internal details of Database
Integrity Constraints	Integrity Constraints are difficult to implement	Integrity constraints are easy to implement
Attributes	To access data in a file , user requires attributes such as file name, file location.	No such attributes are required.
Example	Cobol, C++	Oracle, SQL Server

The main difference between a file system and a DBMS (Database Management System) is the way they organize and manage data.

1. File systems are used to manage files and directories, and provide basic operations for creating, deleting, renaming, and accessing files. They typically store data in a hierarchical structure, where

files are organized in directories and subdirectories. File systems are simple and efficient, but they lack the ability to manage complex data relationships and ensure data consistency.

2. On the other hand, DBMS is a software system designed to manage large amounts of structured data, and provide advanced operations for storing, retrieving, and manipulating data. DBMS provides a centralized and organized way of storing data, which can be accessed and modified by multiple users or applications. DBMS offers advanced features like data validation, indexing, transactions, concurrency control, and backup and recovery mechanisms. DBMS ensures data consistency, accuracy, and integrity by enforcing data constraints, such as primary keys, foreign keys, and data types.

Characteristics of Database Management Systems (DBMS):

1. **Data Integration:** DBMS centralizes data storage, allowing for easy integration and access across the organization.
2. **Data Independence:** DBMS separates the physical storage of data from the applications, providing flexibility and ease of maintenance.
3. **Data Integrity:** DBMS enforces data integrity constraints to ensure the consistency and accuracy of stored data.
4. **Concurrency Control:** DBMS manages concurrent access to data, preventing data corruption and maintaining consistency.
5. **Query Optimization:** DBMS optimizes query execution for improved performance and efficiency.

Advantages of DBMS:

1. **Data Consistency:** DBMS ensures data consistency by minimizing redundancy and enforcing data integrity constraints.
2. **Data Security:** DBMS provides robust security features to protect sensitive data from unauthorized access or modification.
3. **Data Sharing:** DBMS facilitates data sharing and collaboration across departments, improving organizational efficiency.
4. **Scalability:** DBMS scales to handle large volumes of data and concurrent users, supporting the growth of the organization.
5. **Data Independence:** DBMS separates data from applications, allowing for easier maintenance and evolution of the database schema.
6. **Query Capabilities:** DBMS offers powerful query languages and optimization techniques for efficient data retrieval and manipulation.
7. **Data Analysis:** DBMS supports data analysis and reporting through features such as OLAP and data mining.

Data models in Database Management Systems (DBMS)

Data models in Database Management Systems (DBMS) represent the structure of the database and the relationships between its elements. They provide a conceptual framework for organizing and understanding the data stored in the database.

Data models used in DBMS:

1. Relational Model:

Features:

- Organizes data into tables with rows (tuples) and columns (attributes).
- Defines relationships between tables using keys (primary, foreign).
- Supports ACID properties (Atomicity, Consistency, Isolation, Durability) for transaction management.
- Uses Structured Query Language (SQL) for data manipulation and querying.

Advantages:

- Well-defined structure facilitates data integrity and consistency.
- Allows for flexible querying and data retrieval using SQL.
- Supports normalization techniques to minimize redundancy and improve efficiency.
- Widely used and supported by various relational database management systems (RDBMS).

Disadvantages:

- Performance may degrade with complex queries or large datasets.
- Not suitable for handling unstructured or semi-structured data efficiently.
- Schema modifications can be complex and may require downtime.
- May lack support for certain advanced features (e.g., hierarchical data).

2. Object-Oriented Model:

Features:

- Represents data as objects with attributes and methods.
- Supports inheritance, encapsulation, and polymorphism.
- Allows for complex data structures and relationships between objects.

Advantages:

- Mimics real-world entities and relationships more closely than relational models.
- Supports complex data types and behaviors, facilitating modeling of complex domains.

- Promotes code reuse and modularity through object-oriented principles.
- Suitable for applications with complex business logic and domain-specific requirements.

Disadvantages:

- Lack of standardized query language, leading to vendor-specific implementations.
- Object-relational impedance mismatch, where the object-oriented model does not align perfectly with relational databases.
- May require specialized skills and tools for development and maintenance.
- Performance overhead associated with object-relational mapping (ORM) frameworks.

3. Document Model (NoSQL):

Features:

- Stores data as flexible, self-contained documents (e.g., JSON, BSON).
- Supports nested structures and arrays within documents.
- Schema flexibility allows for dynamic addition and modification of fields.
- Scales horizontally to handle large volumes of data and high concurrency.

Advantages:

- Well-suited for handling semi-structured or unstructured data, such as documents, JSON, or XML.
- Facilitates rapid development and iteration, as schema changes do not require downtime.
- Supports horizontal scaling and distributed architectures, providing high availability and fault tolerance.
- Provides flexibility to adapt to evolving data requirements and application needs.

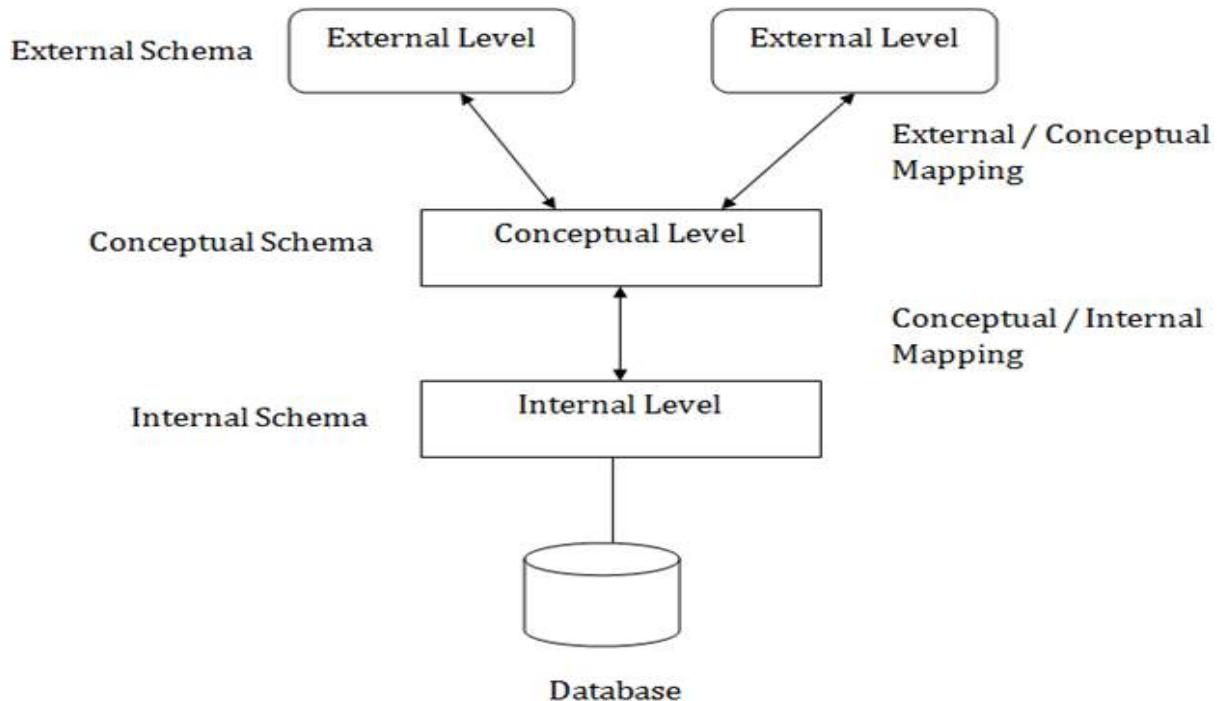
Disadvantages:

- Limited support for complex transactions and ACID properties compared to relational databases.
- May lack standardized query languages, requiring vendor-specific APIs or tools.
- Not ideal for applications requiring complex relationships or joins between documents.
- Data consistency may be more challenging to maintain in distributed environments.

Three schema Architecture

- The three schema architecture is also called ANSI/SPARC architecture or three-level architecture.
- This framework is used to describe the structure of a specific database system.
- The three schema architecture is also used to separate the user applications and physical database.
- The three schema architecture contains three-levels. It breaks the database down into three different categories.

The three-schema architecture is as follows:



In the above diagram:

- It shows the DBMS architecture.
- Mapping is used to transform the request and response between various database levels of architecture.
- Mapping is not good for small DBMS because it takes more time.
- In External / Conceptual mapping, it is necessary to transform the request from external level to conceptual schema.
- In Conceptual / Internal mapping, DBMS transform the request from the conceptual to internal level.

Objectives of Three schema Architecture

The main objective of three level architecture is to enable multiple users to access the same data with a personalized view while storing the underlying data only once. Thus it separates the user's view from the physical structure of the database.

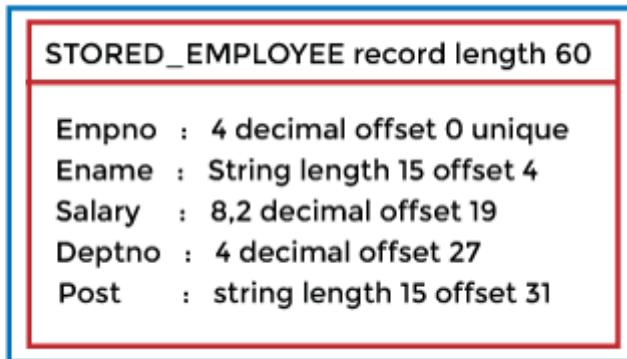
This separation is desirable for the following reasons:

- Different users need different views of the same data.
- The approach in which a particular user needs to see the data may change over time.

- The users of the database should not worry about the physical implementation and internal workings of the database such as data compression and encryption techniques, hashing, optimization of the internal structures etc.
- All users should be able to access the same data according to their requirements.
- DBA should be able to change the conceptual structure of the database without affecting the user's
- Internal structure of the database should be unaffected by changes to physical aspects of the storage.

1. Internal Level

Internal view



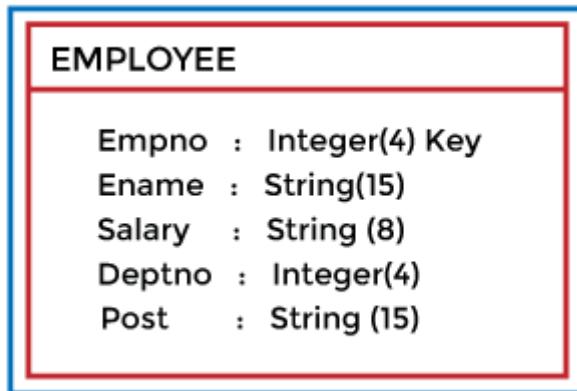
- The internal level has an internal schema which describes the physical storage structure of the database.
- The internal schema is also known as a physical schema.
- It uses the physical data model. It is used to define that how the data will be stored in a block.
- The physical level is used to describe complex low-level data structures in detail.

The internal level is generally concerned with the following activities:

- Storage space allocations.
For Example: B-Trees, Hashing etc.
- Access paths.
For Example: Specification of primary and secondary keys, indexes, pointers and sequencing.
- Data compression and encryption techniques.
- Optimization of internal structures.
- Representation of stored fields.

2. Conceptual Level

Global view



- The conceptual schema describes the design of a database at the conceptual level. Conceptual level is also known as logical level.
- The conceptual schema describes the structure of the whole database.
- The conceptual level describes what data are to be stored in the database and also describes what relationship exists among those data.
- In the conceptual level, internal details such as an implementation of the data structure are hidden.
- Programmers and database administrators work at this level.

3. External Level



- At the external level, a database contains several schemas that sometimes called as subschema. The subschema is used to describe the different view of the database.
- An external schema is also known as view schema.
- Each view schema describes the database part that a particular user group is interested and hides the remaining database from that user group.
- The view schema describes the end user interaction with database systems.

Mapping between Views

The three levels of DBMS architecture don't exist independently of each other. There must be correspondence between the three levels i.e. how they actually correspond with each other. DBMS is responsible for correspondence between the three types of schema. This correspondence is called Mapping.

There are basically two types of mapping in the database architecture:

- Conceptual/ Internal Mapping
- External / Conceptual Mapping

Conceptual/ Internal Mapping

The Conceptual/ Internal Mapping lies between the conceptual level and the internal level. Its role is to define the correspondence between the records and fields of the conceptual level and files and data structures of the internal level.

External/ Conceptual Mapping

The external/Conceptual Mapping lies between the external level and the Conceptual level. Its role is to define the correspondence between a particular external and the conceptual view.

Benefits of 3-Tier Architecture

The 3-tier architecture in DBMS provides several benefits, including:

- **Scalability:** The architecture separates the application processing and data management layers, which allows for easy scalability of each layer independently.
- **Flexibility:** The architecture allows for the replacement or upgrade of one layer without affecting the other layers.
- **Security:** The architecture provides an additional layer of security, as the data management tier can be isolated from the application and presentation tiers, reducing the risk of unauthorized access.

Overall, the 3-tier architecture in DBMS is a flexible, scalable, and secure approach to building modern web applications and enterprise systems. It separates the user interface, application processing, and data management into distinct layers, providing clear boundaries between each layer and improving system performance, reliability, and maintainability.

Advantages and Disadvantages of using Three Schema Architecture:

There are various **Advantages** to employing a three-schema architecture in database management systems. Some of the benefits include:

- One of the primary advantages of a DBMS's three schemas is data independence. All three levels are separate from one another. As a result, we can update one layer without affecting the others.
- Each schema may scale separately, allowing the database to perform better while also handling more traffic.
- Because the layers are separated in a three-schema design, it is easier to maintain and alter each one individually.

Disadvantages:

Despite its many advantages, the three schema architecture has a few drawbacks:

- This strategy can be challenging and costly for large corporations because it requires a lot of effort to set up and maintain.
- It can also cause delays and errors if the data is not properly translated between the various components.
- It can also be difficult to ensure that only authorised individuals have access to sensitive information.

Data Independence

- Data independence can be explained using the three-schema architecture.
- Data independence refers characteristic of being able to modify the schema at one level of the database system without altering the schema at the next higher level.

There are two types of data independence:

1. Logical Data Independence

- Logical data independence refers to the characteristic of being able to change the conceptual schema without having to change the external schema.
- Logical data independence is used to separate the external level from the conceptual view.
- If we do any changes in the conceptual view of the data, then the user view of the data would not be affected.
- Logical data independence occurs at the user interface level.

2. Physical Data Independence

- Physical data independence can be defined as the capacity to change the internal schema without having to change the conceptual schema.
- If we do any changes in the storage size of the database system server, then the conceptual structure of the database will not be affected.
- Physical data independence is used to separate conceptual levels from the internal levels.
- Physical data independence occurs at the logical interface level.

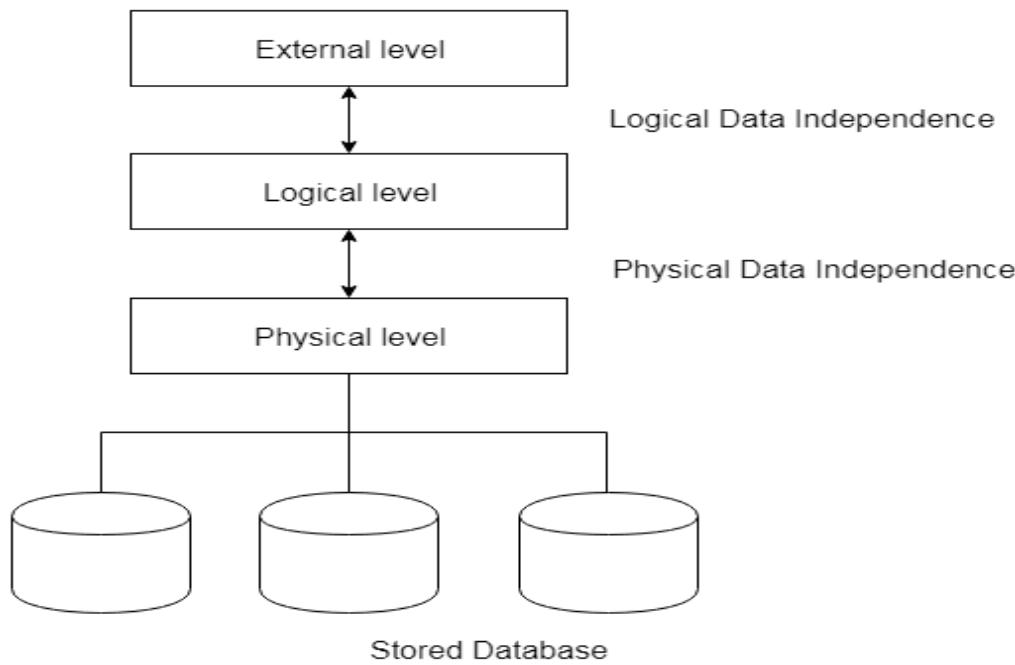


Fig: Data Independence

Data Abstraction in DBMS

Data abstraction is the procedure of concealing irrelevant or unwanted data from the end user.

Let us understand this with an example, if you go to a shop to buy a pair of shoes, you ask the shopkeeper to show you the shoes of a certain company, and you also tell the shopkeeper about the size, color, and material you want. Then, you will only see the specified things in the shoes, or will you be asking the shopkeeper questions such as, where are these shoes made? From where does the material come? What is the cost of the material?

The answer to these questions is **NO**. You will not ask these questions because these questions are of no use. You do not care about these questions. You are only concerned about a few things, such as the company, size, color, material, and how the shoes look. That is why these unimportant details are kept hidden from the end user. This is the process we call data abstraction.

What is Data abstraction in Database Management System?

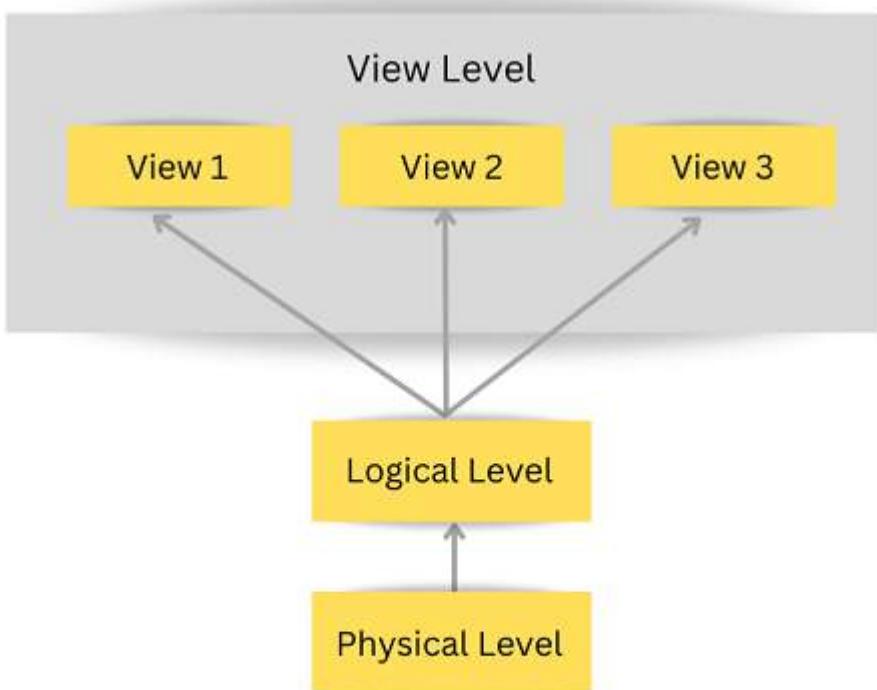
The database system contains intricate data structures and relations. The developers keep away the complex data from the user and remove the complications so that the user can comfortably access data in the database and can only access the data they want, which is done with the help of data abstraction.

The main purpose of data abstraction is to hide irrelevant data and provide an abstract view of the data. With the help of data abstraction, developers hide irrelevant data from the user and provide them the relevant data. By doing this, users can access the data without any hassle, and the system will also work efficiently.

In DBMS, data abstraction is performed in layers which means there are levels of data abstraction in. Based on these levels, the database management system is designed.

Levels of Data Abstractions in DBMS

In DBMS, there are three levels of data abstraction, which are as follows:



Levels of Data Abstraction in DBMS

1. Physical or Internal Level:

The physical or internal layer is the lowest level of data abstraction in the database management system. It is the layer that defines how data is actually stored in the database. It defines methods to access the data in the database. It defines complex data structures in detail, so it is very complex to understand, which is why it is kept hidden from the end user.

Data Administrators (DBA) decide how to arrange data and where to store data. The Data Administrator (DBA) is the person whose role is to manage the data in the database at the physical or internal level. There is a data center that securely stores the raw data in detail on hard drives at this level.

2. Logical or Conceptual Level:

The logical or conceptual level is the intermediate or next level of data abstraction. It explains what data is going to be stored in the database and what the relationship is between them.

It describes the structure of the entire data in the form of tables. The logical level or conceptual level is less complex than the physical level. With the help of the logical level, Data Administrators (DBA) abstract data from raw data present at the physical level.

3. View or External Level:

View or External Level is the highest level of data abstraction. There are different views at this level that define the parts of the overall data of the database. This level is for the end-user interaction; at this level, end users can access the data based on their queries.

Advantages of data abstraction in DBMS

- Users can easily access the data based on their queries.
- It provides security to the data stored in the database.
- Database systems work efficiently because of data abstraction.

DBMS Architecture

Database management systems architecture will help us understand the components of database system and the relation among them.

The architecture of DBMS depends on the computer system on which it runs. For example, in a client-server DBMS architecture, the database systems at server machine can run several requests made by client machine. We will understand this communication with the help of diagrams.

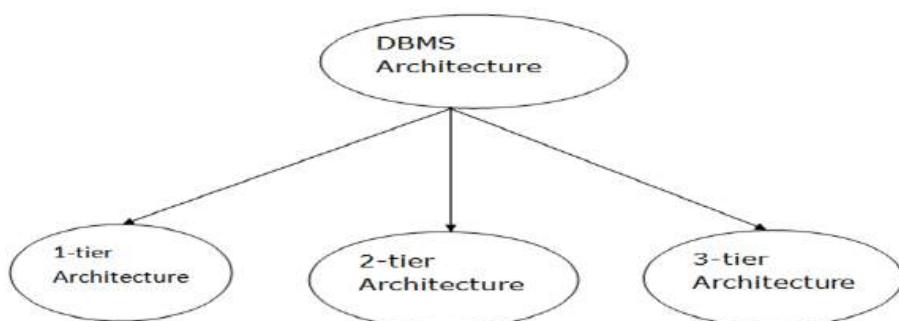
The DBMS design depends upon its architecture. The basic client/server architecture is used to deal with a large number of PCs, web servers, database servers and other components that are connected with networks.

- The client/server architecture consists of many PCs and a workstation which are connected via the network.
- DBMS architecture depends upon how users are connected to the database to get their request done.

Types of DBMS Architecture:

There are three types of DBMS architecture:

1. Single tier architecture
2. Two tier architecture
3. Three tier architecture



Database architecture can be seen as a single tier or multi-tier. But logically, database architecture is of two types like: **2-tier architecture** and **3-tier architecture**.

1-Tier Architecture /Single Tier Architecture

- In this architecture, the database is directly available to the user. It means the user can directly sit on the DBMS and uses it.
- Any changes done here will directly be done on the database itself. It doesn't provide a handy tool for end users.
- The 1-Tier architecture is used for development of the local application, where programmers can directly communicate with the database for the quick response.

For example, lets say you want to fetch the records of employee from the database and the database is available on your computer system, so the request to fetch employee details will be done by your computer and the records will be fetched from the database by your computer as well. This type of system is generally referred as local database system.

2-Tier Architecture

- The 2-Tier architecture is same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. For this interaction, API's like: **ODBC**, **JDBC** are used.
- The user interfaces and application programs are run on the client-side.
- The server side is responsible to provide the functionalities like: query processing and transaction management.
- To communicate with the DBMS, client-side application establishes a connection with the server side.
- To communicate with the DBMS, client-side application establishes a connection with the server side.

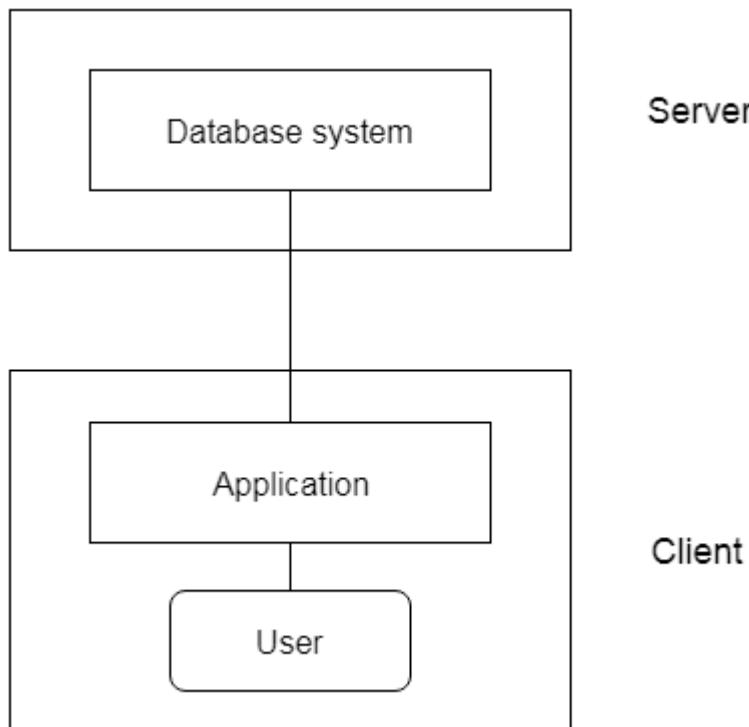
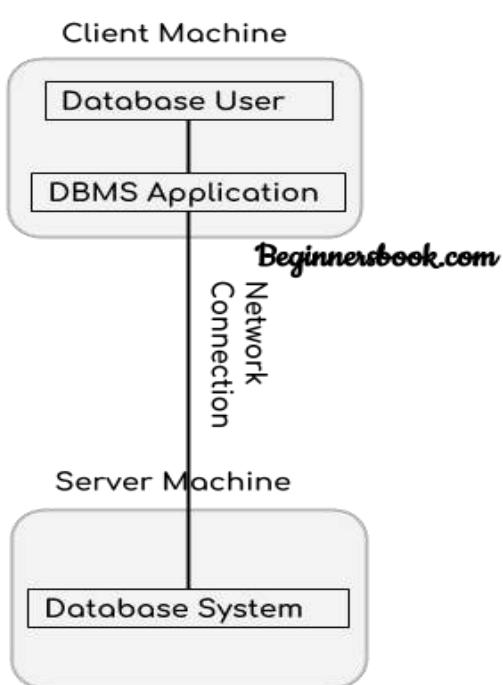


Fig: 2-tier Architecture



Two-Tier architecture

Fig: 2-tier Architecture

In two-tier architecture, the Database system is present at the server machine and the DBMS application is present at the client machine, these two machines are connected with each other through a reliable network as shown in the above diagram.

Whenever client machine makes a request to access the database present at server using a query language like sql, the server perform the request on the database and returns the result back to the client. The application connection interface such as JDBC, ODBC are used for the interaction between server and client.

3-Tier Architecture

- The 3-Tier architecture contains another layer between the client and server. In this architecture, client can't directly communicate with the server.
- The application on the client-end interacts with an application server which further communicates with the database system.
- End user has no idea about the existence of the database beyond the application server. The database also has no idea about any other user beyond the application.
- The 3-Tier architecture is used in case of large web application.

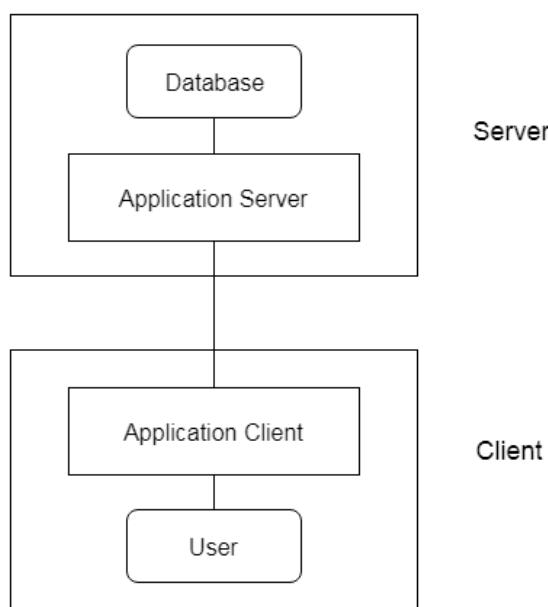
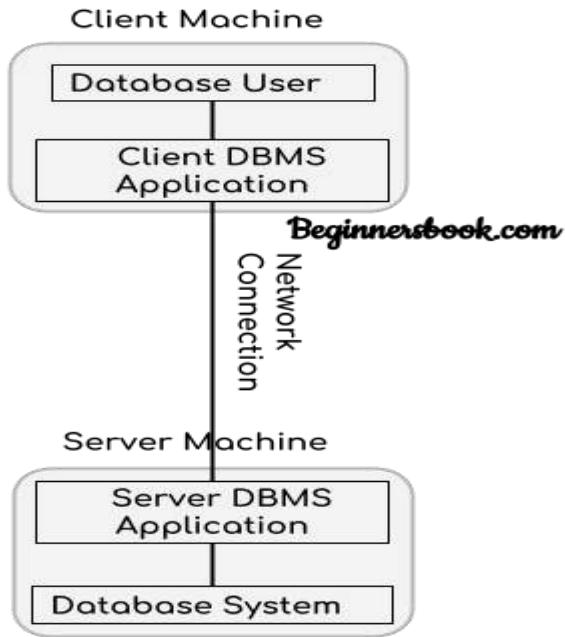


Fig: 3-tier Architecture



Three-Tier architecture

In three-tier architecture, another layer is present between the client machine and server machine. In this architecture, the client application doesn't communicate directly with the database systems present at the server machine, rather the client application communicates with server application and the server application internally communicates with the database system present at the server.

Structure of DBMS

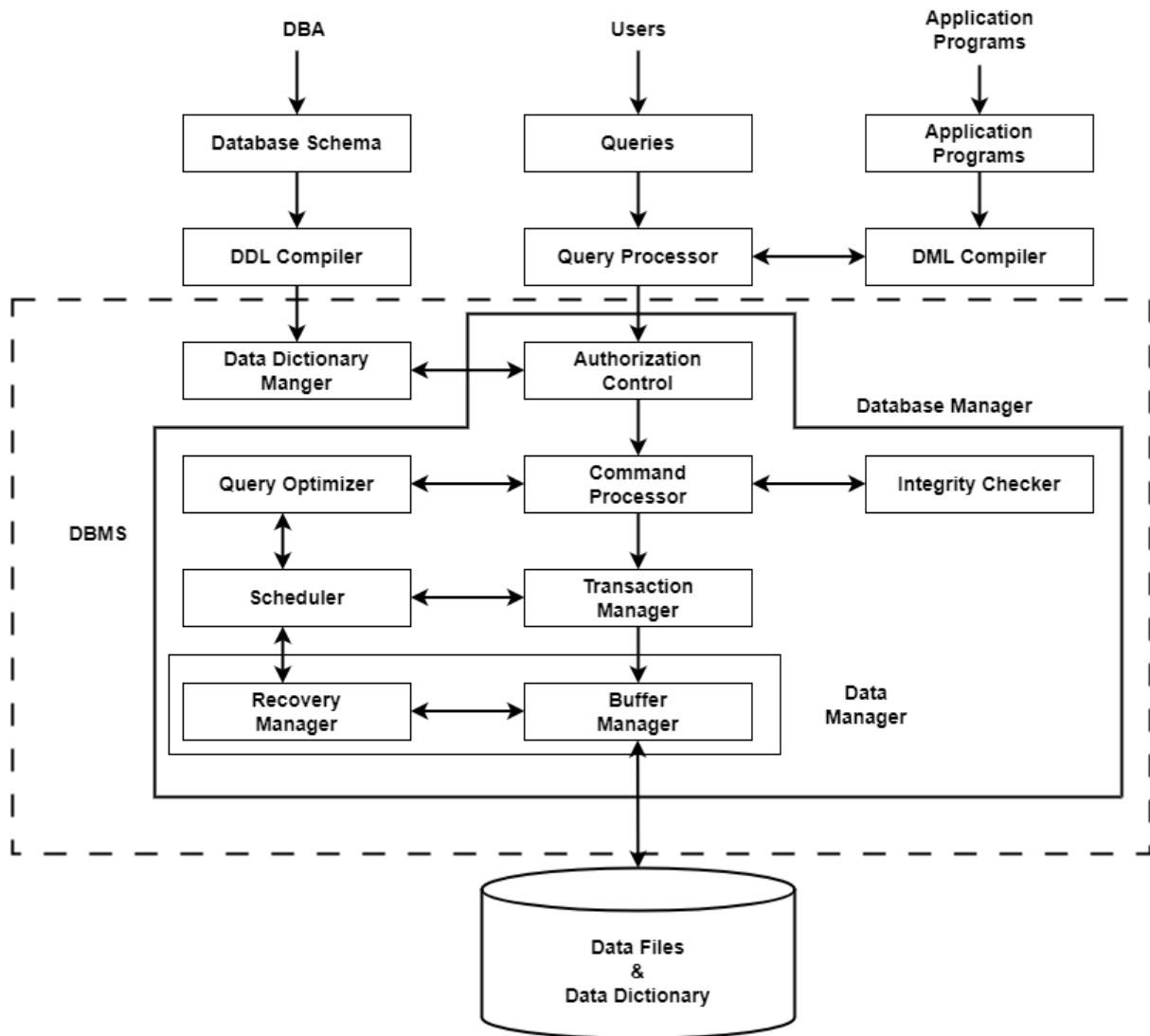
DBMS: DBMS means Database Management System, which is a tool or software used to create the database or delete or manipulate the database. A software programme created to store, retrieve, query, and manage data is known as a Database Management System (DBMS). Data can be generated, read, updated, and destroyed by authorized entities thanks to user interfaces (UIs).

Because they give programmers, Database Managers, and end users a consolidated view of the data, Database Management Systems are crucial because they relieve applications and end users of the need to comprehend the physical location of the data. Application Programme Interfaces (APIs) manage internet requests and responses for particular sorts of data.

In marketing materials, the phrase "database as a service" (DBaaS) may be used to refer to both relational and non-relational DBMS components that are given via the internet.

Users of DBMSs include application programmers, Database Administrators (DBAs), and end users.

Database Administrators are typically the only people who work directly with a DBMS. Today, end users read and write to databases using front-end interfaces made by programmers, while programmers use cloud APIs to connect with DBMSs.



Three Parts that make up the Database System are:

- Query Processor
- Storage Manager
- Disk Storage

The explanations for these are provided below:

1. Query Processor

The query processing is handled by the query processor, as the name implies. It executes the user's query, to put it simply. In this way, the query processor aids the database system in making data access simple and easy. The query processor's primary duty is to successfully execute the query. The Query Processor transforms (or interprets) the user's application program-provided requests into instructions that a computer can understand.

Components of the Query Processor

- **DDL Interpreter:**

Data Definition Language is what DDL stands for. As implied by the name, the DDL Interpreter interprets DDL statements like those used in schema definitions (such as create, remove, etc.). This interpretation yields a set of tables that include the metadata (data of data) that is kept in the data dictionary. Metadata may be stored in a data dictionary. In essence, it is a part of the disc storage that will be covered in a later section of this article.

- **DML Compiler:**

Compiler for DML Data Manipulation Language is what DML stands for. In keeping with its name, the DML Compiler converts DML statements like select, update, and delete into low-level instructions or simply machine-readable object code, to enable execution. The optimization of queries is another function of the DML compiler. Since a single question can typically be translated into a number of evaluation plans. As a result, some optimization is needed to select the evaluation plan with the lowest cost out of all the options. This process, known as query optimization, is exclusively carried out by the DML compiler. Simply put, query optimization determines the most effective technique to carry out a query.

- **Embedded DML Pre-compiler:**

Before the query evaluation, the embedded DML commands in the application program (such as SELECT, FROM, etc., in SQL) must be pre-compiled into standard procedural calls (program instructions that the host language can understand). Therefore, the DML statements which are embedded in an application program must be converted into routine calls by the Embedded DML Pre-compiler.

- **Query Optimizer:**

It starts by taking the evaluation plan for the question, runs it, and then returns the result. Simply said, the query evaluation engine evaluates the SQL commands used to access the database's contents before returning the result of the query. In a nutshell, it is in charge of analyzing the queries and running the object code that the DML Compiler produces. Apache Drill, Presto, and other Query Evaluation Engines are a few examples.

2. Storage Manager:

An application called Storage Manager acts as a conduit between the queries made and the data kept in the database. Another name for it is Database Control System. By applying the restrictions and running the DCL instructions, it keeps the database's consistency and integrity. It is in charge of retrieving, storing, updating, and removing data from the database.

Components of Storage Manager

Following are the components of Storage Manager:

- **Integrity Manager:**

Whenever there is any change in the database, the Integrity manager will manage the integrity constraints.

- **Authorization Manager:**

Authorization manager verifies the user that he is valid and authenticated for the specific query or request.

- **File Manager:**

All the files and data structure of the database are managed by this component.

- **Transaction Manager:**

It is responsible for making the database consistent before and after the transactions. Concurrent processes are generally controlled by this component.

- **Buffer Manager:**

The transfer of data between primary and main memory and managing the cache memory is done by the buffer manager.

3. Disk Storage

A DBMS can use various kinds of Data Structures as a part of physical system implementation in the form of disk storage.

Components of Disk Storage

Following are the components of Disk Manager:

- **Data Dictionary:**

It contains the metadata (data of data), which means each object of the database has some information about its structure. So, it creates a repository which contains the details about the structure of the database object.

- **Data Files:**

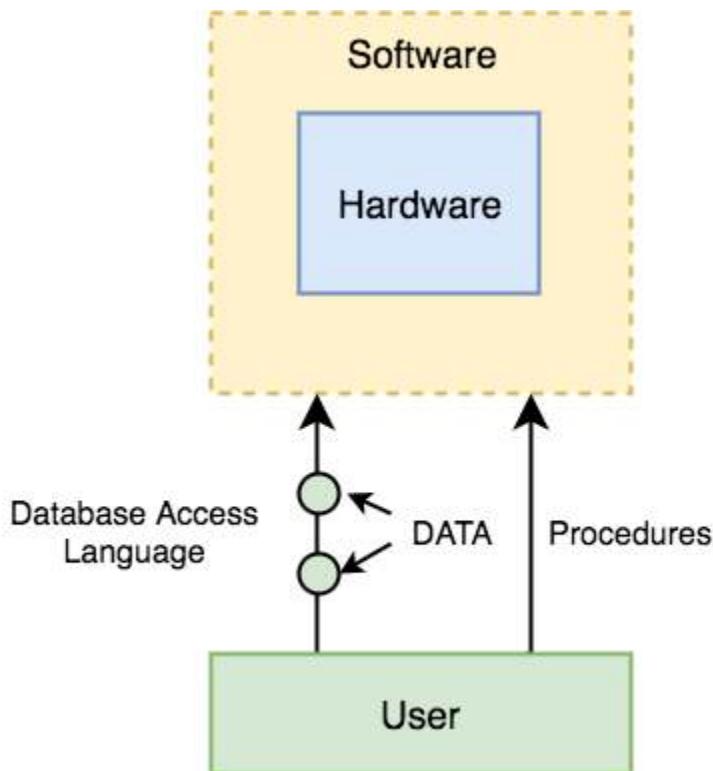
This component stores the data in the files.

- **Indices:**

These indices are used to access and retrieve the data in a very fast and efficient way.

Database system Environment:

The database system consists of many components. Each component performing very significant tasks in the database management system environment. A database environment is a collective system of components that comprise and regulates the group of data, management, and use of data, which consist of software, hardware, people, techniques of handling database, and the data also.



The hardware in a database environment means the computers and computer peripherals that are being used to manage a database, and the software means the whole thing right from the operating system (OS) to the application programs that include database management software like M.S. Access or SQL Server. Again the people in a database environment include those people who administrate and use the system. The techniques are the rules, concepts, and instructions given to both the people and the software along with the data with the group of facts and information positioned within the database environment.

Components of DBMS

There are many components available in the DBMS. Each component has a significant task in the DBMS. The database management system can be divided into five major components, they are:

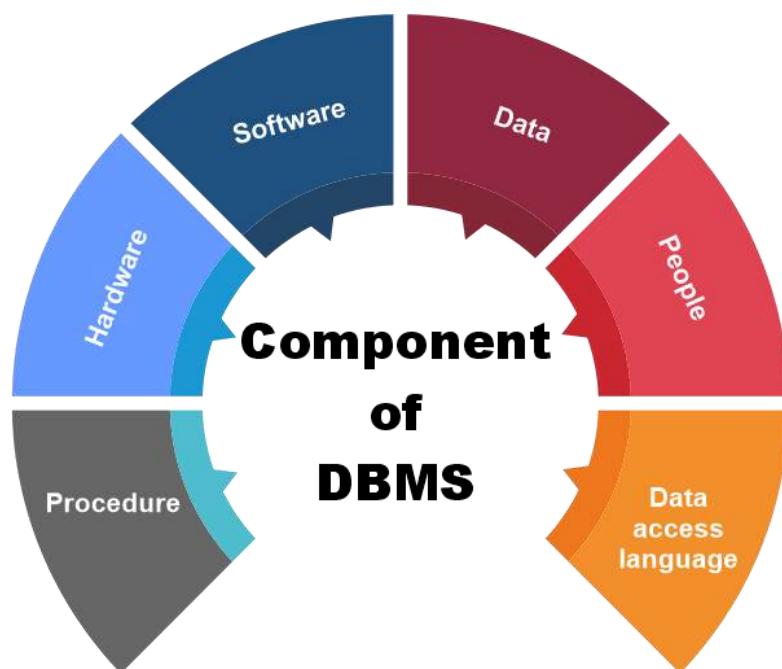
1.Hardware

2.Software

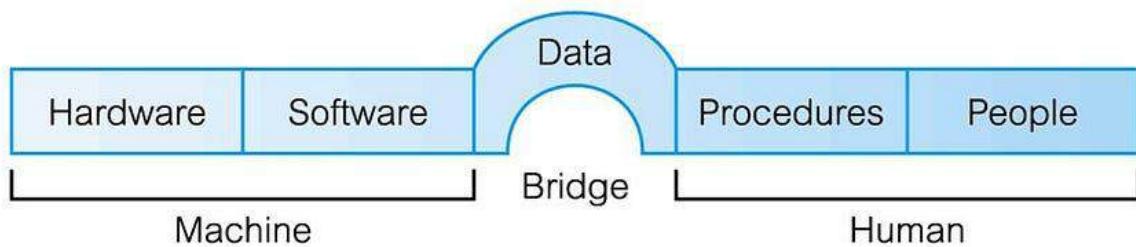
3.Data

4.Procedures

5.People



Components of DBMS Environment



1. Hardware

- Here the hardware means the physical part of the DBMS. Here the hardware includes output devices like a printer, monitor, etc., and storage devices like a hard disk.
- In DBMS, information hardware is the most important visible part. The equipment which is used for the visibility of the data is the printer, computer, scanner, etc. This equipment is used to capture the data and present the output to the user.
- With the help of hardware, the DBMS can access and update the database.
- The server can store a large amount of data, which can be shared with the help of the user's own system.
- The database can be run in any system that ranges from microcomputers to mainframe computers. And this database also provides an interface between the real worlds to the database.
- When we try to run any database software like MySQL, we can type any commands with the help of our keyboards, and RAM, ROM, and processor are part of our computer system.

2. Software

- Software is the main component of the DBMS.

- Software is defined as the collection of programs that are used to instruct the computer about its work. The software consists of a set of procedures, programs, and routines associated with the computer system's operation and performance. Also, we can say that computer software is a set of instructions that is used to instruct the computer hardware for the operation of the computers.
- The software includes so many software like network software and operating software. The database software is used to access the database, and the database application performs the task.
- This software has the ability to understand the database accessing language and then convert these languages to real database commands and then execute the database.
- This is the main component as the total database operation works on a software or application. We can also be called as database software the wrapper of the whole physical database, which provides an easy interface for the user to store, update and delete the data from the database.
- Some examples of DBMS software include MySQL, Oracle, SQL Server, dBase, FileMaker, Clipper, Foxpro, Microsoft Access, etc.

3. Data

- The term data means the collection of any raw fact stored in the database. Here the data are any type of raw material from which meaningful information is generated.
- The database can store any form of data, such as structural data, non-structural data, and logical data.
- The structured data are highly specific in the database and have a structured format. But in the case of non-structural data, it is a collection of different types of data, and these data are stored in their native format.
- We also call the database the structure of the DBMS. With the help of the database, we can create and construct the DBMS. After the creation of the database, we can create, access, and update that database.
- The main reason behind discovering the database is to create and manage the data within the database.
- Data is the most important part of the DBMS. Here the database contains the actual data and metadata. Here metadata means data about data.
- For example, when the user stores the data in a database, some data, such as the size of the data, the name of the data, and some data related to the user, are stored within the database. These data are called metadata.

4. Procedures

- The procedure is a type of general instruction or guidelines for the use of DBMS. This instruction includes how to set up the database, how to install the database, how to log in and log out of the database, how to manage the database, how to take a backup of the database, and how to generate the report of the database.

- In DBMS, with the help of procedure, we can validate the data, control the access and reduce the traffic between the server and the clients. The DBMS can offer better performance to extensive or complex business logic when the user follows all the procedures correctly.
 - The main purpose of the procedure is to guide the user during the management and operation of the database.
 - The procedure of the databases is so similar to the function of the database. The major difference between the database procedure and database function is that the database function acts the same as the SQL statement. In contrast, the database procedure is invoked using the CALL statement of the DBMS.
 - Database procedures can be created in two ways in enterprise architecture. These two ways are as below.
 - The individual object or the default object.
 - The operations in a container.
1. CREATE [OR REPLACE] PROCEDURE procedure_name (<Argument> {IN, OUT, IN OUT})
 2. <Datatype>,...)
 3. IS
 4. Declaration section<variable, constant> ;
 5. BEGIN
 6. Execution section
 7. EXCEPTION
 8. Exception section
 9. END

5. Database Access Language

- Database Access Language is a simple language that allows users to write commands to perform the desired operations on the data that is stored in the database.
- Database Access Language is a language used to write commands to access, upsert, and delete data stored in a database.
- Users can write commands or query the database using Database Access Language before submitting them to the database for execution.
- Through utilizing the language, users can create new databases and tables, insert data and delete data.
- Examples of database languages are SQL (structured query language), My Access, Oracle, etc. A database language is comprised of two languages.

1. Data Definition Language(DDL):It is used to construct a database. DDL implements database schema at the physical, logical, and external levels.

ADVERTISEMENT

The following commands serve as the base for all DDL commands:

- ALTER<object>
- COMMENT
- CREATE<object>
- DESCRIBE<object>
- DROP<object>
- SHOW<object>
- USE<object>

2. Data Manipulation Language(DML): It is used to access a database. The DML provides the statements to retrieve, modify, insert and delete the data from the database.

The following commands serve as the base for all DML commands:

- INSERT
- UPDATE
- DELETE
- LOCK
- CALL

6. People

- The people who control and manage the databases and perform different types of operations on the database in the DBMS.
- The people include database administrator, software developer, and End-user.
- Database administrator-database administrator is the one who manages the complete database management system. DBA takes care of the security of the DBMS, its availability, managing the license keys, managing user accounts and access, etc.
- Software developer- This user group is involved in developing and designing the parts of DBMS. They can handle massive quantities of data, modify and edit databases, design and develop new databases, and troubleshoot database issues.
- End user - These days, all modern web or mobile applications store user data. How do you think they do it? Yes, applications are programmed in such a way that they collect user data and store the data on a DBMS system running on their server. End users are the ones who store, retrieve, update and delete data.
- The users of the database can be classified into different groups.
 - i. Native Users
 - ii. Online Users
 - iii. Sophisticated Users

- iv. Specialized Users
- v. Application Users
- vi. DBA - Database Administrator

Schema and Instances in DBMS

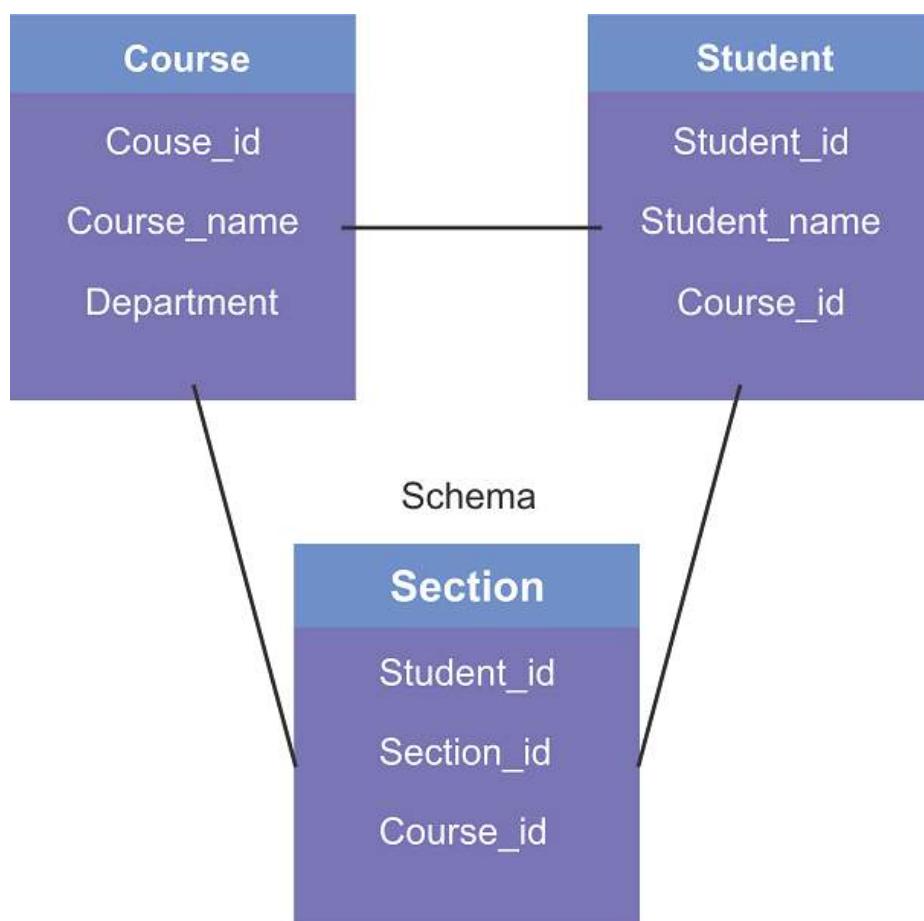
Schema in DBMS

The schema defines the logical view of the database. It provides some knowledge about the database and what data needs to go where.

In DBMS, the schema is shown in diagram format.

We can understand the relationship between the data present in the database. With the help of this schema, we can implement the DBMS function such as delete, insert, search, update, etc.

Let us understand this by the below diagram. There are three diagrams, i.e., section, course, and student. This diagram shows the relationship between the section and the course diagram. Schema is the only type of structural view of the database that is shown below.



- The data which is stored in the database at a particular moment of time is called an instance of the database.
- The overall design of a database is called schema.
- A database schema is the skeleton structure of the database. It represents the logical view of the entire database.
- A schema contains schema objects like table, foreign key, primary key, views, columns, data types, stored procedure, etc.
- A database schema can be represented by using the visual diagram. That diagram shows the database objects and relationship with each other.
- A database schema is designed by the database designers to help programmers whose software will interact with the database. The process of database creation is called data modeling.

A schema diagram can display only some aspects of a schema like the name of record type, data type, and constraints. Other aspects can't be specified through the schema diagram. For example, the given figure neither show the data type of each data item nor the relationship among various files.

In the database, actual data changes quite frequently. For example, in the given figure, the database changes whenever we add a new grade or add a student. The data at a particular moment of time is called the instance of the database.

STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

GRADE_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

Schema is further divided into three types. These three are as follows.

1. Logical schema.
2. View schema.
3. Physical schema.

1. Physical schema:

In the physical schema, the database is designed at the physical level. At this level, the schema describes how the data block is stored and how the storage is managed.

2. Logical schema:

In the logical schema, the database is designed at a logical level. At this level, the programmer and data administrator perform their work. Also, at this level, a certain amount of data is stored in a structured way. But the internal implementation data are hidden in the physical layer for the security proposed.

3. View schema:

In view schema, the database is designed at the view level. This schema describes the user interaction with the database system.

Moreover, Data Definition Language (DDL) statements help to denote the schema of a database. The schema represents the name of the table, the name of attributes, and their types; constraints of the tables are related to the schema. Therefore, if users want to modify the schema, they can write DDL statements.

Instance in DBMS

In DBMS, the data is stored for a particular amount of time and is called an instance of the database. The database schema defines the attributes of the database in the particular DBMS. The value of the particular attribute at a particular moment in time is known as an instance of the DBMS.

Let's take an example: we have a single table student in the database; today, the table has 100 records, so today, the instance of the database has 100 records. We are going to add another 100 records to this table by tomorrow, so the instance of the database tomorrow will have 200 records in the table. In short, at a particular moment, the data stored in the database is called the instance; this change over time as and when we add, delete or update data in the database.

Example:

A database instance for the Person database can be (UserName,email,contact) So the person construct will contain their individual entities in the attributes called as instance. This is shown below –

Person

Name	Email	Phone no
BOB	kksd@yasd.com	2343435
JANU	werwr@sadas.in	5345464
PRIYA	wefrwer@sdf.com	2342342

Differences between Database Schema and Instance

Both of these help in describing the data available in a database, but there is a fundamental difference between Schema and Instance in DBMS. Schema refers to the overall description of any given database. Instance basically refers to a collection of data and information that the database stores at any particular moment.

The major differences between schema and instance are as follows:

Database Schema	Database Instance
It is the definition of the database, or it is defined as the description of the database.	It is a snapshot of a database at a specific moment.
This corresponds to the variable declaration of a programming language.	The value of the variable in a program at a point in time corresponds to an instance of the database schema.
Defines the basic structure of the database, i.e., how the data will be stored in the database.	It is the set of information stored at a particular time.
Schema is same for whole database.	Data in instances can be changed using addition, deletion, updation.
It does not change very frequently.	It changes very frequently

RELATIONAL ALGEBRA

Relational Algebra is a procedural query language. Relational algebra mainly provides a theoretical foundation for relational databases and [SQL](#). The main purpose of using Relational Algebra is to define operators that transform one or more input relations into an output relation.

Given that these operators accept relations as input and produce relations as output, they can be combined and used to express potentially complex queries that transform potentially many input relations (whose data are stored in the database) into a single output relation (the query results). As it is pure mathematics, there is no use of English Keywords in Relational Algebra and operators are represented using symbols.

Fundamental Operators

These are the basic/fundamental operators used in Relational Algebra.

1. Selection(σ)
2. Projection(π)
3. Union(U)
4. Set Difference (-)
5. Set Intersection (\cap)
6. Rename(ρ)
7. Cartesian Product(X)

Advantages of relational algebra

The relational algebra has solid mathematical background. The mathematical background of relational algebra is the basis of many interesting developments and theorems. If we have two expressions for the same operation and if the expressions are proved to be equivalent, then a query optimizer can automatically substitute the more efficient form. Moreover, the relational algebra is a high level language which talks in terms of properties of sets of tuples and not in terms of for-loops.

Limitations of relational algebra

1. The relational algebra can't do arithmetic.

For example, if we want to know the price of 101 of petrol, by assuming a 10% increase in the price of the petrol, which can't be done using relational algebra.

2. The relational algebra cannot sort or print results in various formats.

For example we want to arrange the product name in the increasing order of their price. It can't be done using relational algebra.

3. Relational algebra can't perform aggregates.

4. The relational algebra cannot modify the database.

5. The relational algebra cannot compute “transitive closure”

Operators in Relational Algebra

1. Selection(σ): It is used to select required tuples of the relations.

Example:

A	B	C
1	2	4
2	2	3
3	2	3
4	3	4

For the above relation, $\sigma(c>3)R$ will select the tuples which have c more than 3.

A	B	C
1	2	4
4	3	4

Note: The selection operator only selects the required tuples but does not display them. For display, the data projection operator is used.

2. Projection(π): It is used to project required column data from a relation.

Example: Consider Table 1. Suppose we want columns B and C from Relation R.

$\pi(B,C)R$ will show following columns.

B	C
2	4
2	3
3	4

Note: By Default, projection removes duplicate data.

3. Union(U): Union operation in relational algebra is the same as union operation in [set theory](#).

Example:

FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN

Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21
Rohan	25

Consider the following table of Students having different optional subjects in their course.

$\pi(\text{Student_Name})\text{FRENCH} \cup \pi(\text{Student_Name})\text{GERMAN}$

Student_Name
Ram

Student_Name
Mohan
Vivek
Geeta
Shyam
Rohan

Note: The only constraint in the union of two relations is that both relations must have the same set of Attributes.

4. Set Difference(-): Set Difference in relational algebra is the same set difference operation as in set theory.

Example: From the above table of FRENCH and GERMAN, Set Difference is used as follows

$$\pi(\text{Student_Name})\text{FRENCH} - \pi(\text{Student_Name})\text{GERMAN}$$

Student_Name
Ram
Mohan

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

5. Set Intersection(\cap): Set Intersection in relational algebra is the same set intersection operation in set theory.

Example: From the above table of FRENCH and GERMAN, the Set Intersection is used as follows

$$\pi(\text{Student_Name})\text{FRENCH} \cap \pi(\text{Student_Name})\text{GERMAN}$$

Student_Name
Vivek
Geeta

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

6. Rename(p): Rename is a unary operation used for renaming attributes of a relation.

$\rho(a/b)R$ will rename the attribute 'b' of the relation by 'a'.

7. Cross Product(X): Cross-product between two relations. Let's say A and B, so the cross product between A X B will result in all the attributes of A followed by each attribute of B. Each record of A will pair with every record of B.

Example:

A

Name	Age	Sex
Ram	14	M
Sona	15	F
Kim	20	M

B

ID	Course
1	DS
2	DBMS

A X B

Name	Age	Sex	ID	Course
Ram	14	M	1	DS
Ram	14	M	2	DBMS
Sona	15	F	1	DS
Sona	15	F	2	DBMS
Kim	20	M	1	DS
Kim	20	M	2	DBMS

Note: If A has 'n' tuples and B has 'm' tuples then A X B will have ' n*m ' tuples.

Derived Operators

These are some of the [derived operators](#), which are derived from the fundamental operators.

1. [Natural Join\(\$\bowtie\$ \)](#)
2. [Conditional Join](#)

1. Natural Join(\bowtie): Natural join is a binary operator. Natural join between two or more relations will result in a set of all combinations of tuples where they have an equal common attribute.

Example:

EMP

Name	ID	Dept_Name
A	120	IT
B	125	HR
C	110	Sales

Name	ID	Dept_Name
D	111	IT

DEPT

Dept_Name	Manager
Sales	Y
Production	Z
IT	A

Natural join between EMP and DEPT with condition :

EMP.Dept_Name = DEPT.Dept_Name

EMP ⚡ DEPT

Name	ID	Dept_Name	Manager
A	120	IT	A
C	110	Sales	Y
D	111	IT	A

2. Conditional Join: Conditional join works similarly to natural join. In natural join, by default condition is equal between common attributes while in conditional join we can specify any condition such as greater than, less than, or not equal.

Example:

R

ID	Sex	Marks
1	F	45
2	F	55
3	F	60

S

ID	Sex	Marks
10	M	20
11	M	22
12	M	59

Join between R and S with condition **R.marks >= S.marks**

R.ID	R.Sex	R.Marks	S.ID	S.Sex	S.Marks
1	F	45	10	M	20
1	F	45	11	M	22
2	F	55	10	M	20
2	F	55	11	M	22
3	F	60	10	M	20

R.ID	R.Sex	R.Marks	S.ID	S.Sex	S.Marks
3	F	60	11	M	22
3	F	60	12	M	59

Advantages of relational algebra

The relational algebra has solid mathematical background. The mathematical background of relational algebra is the basis of many interesting developments and theorems. If we have two expressions for the same operation and if the expressions are proved to be equivalent, then a query optimizer can automatically substitute the more efficient form. Moreover, the relational algebra is a high level language which talks in terms of properties of sets of tuples and not in terms of for-loops.

Limitations of relational algebra

1. The relational algebra can't do arithmetic.

For example, if we want to know the price of 101 of petrol, by assuming a 10% increase in the price of the petrol, which can't be done using relational algebra.

2. The relational algebra cannot sort or print results in various formats.

For example we want to arrange the product name in the increasing order of their price. It can't be done using relational algebra.

3. Relational algebra can't perform aggregates.

4. The relational algebra cannot modify the database.

5. The relational algebra cannot compute “transitive closure”

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 in actuality, this huge amount of data is stored in physical memory in the form of files.

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](#).

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

Types of File Organizations

Various methods have been introduced to Organize files. These particular 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:

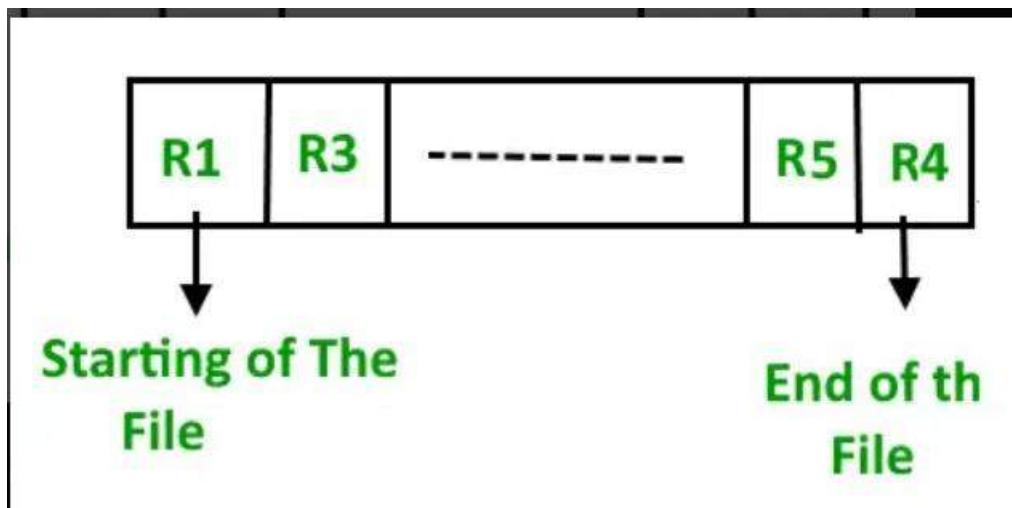
- Sequential File Organization
- Heap File Organization
- Hash File Organization
- B+ Tree File Organization
- Clustered File Organization
- ISAM (Indexed Sequential Access Method)

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:

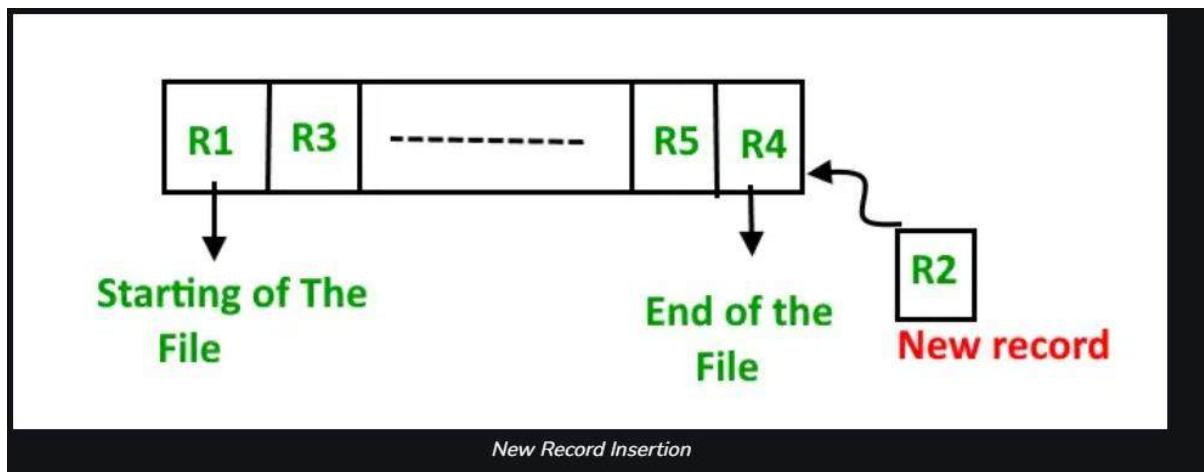
1. Pile File Method

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.



Pile File Method

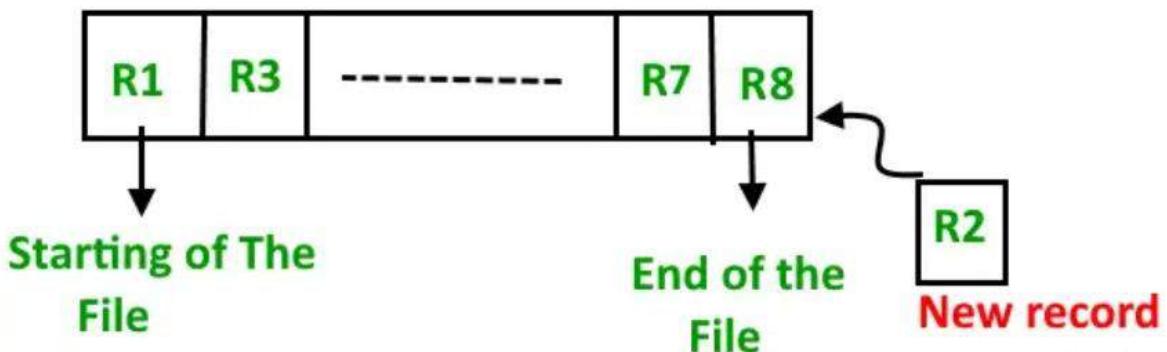
Insertion of the new record: Let the R1, R3, and so on up to R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, then it is simply placed at the end of the file.



New Record Insertion

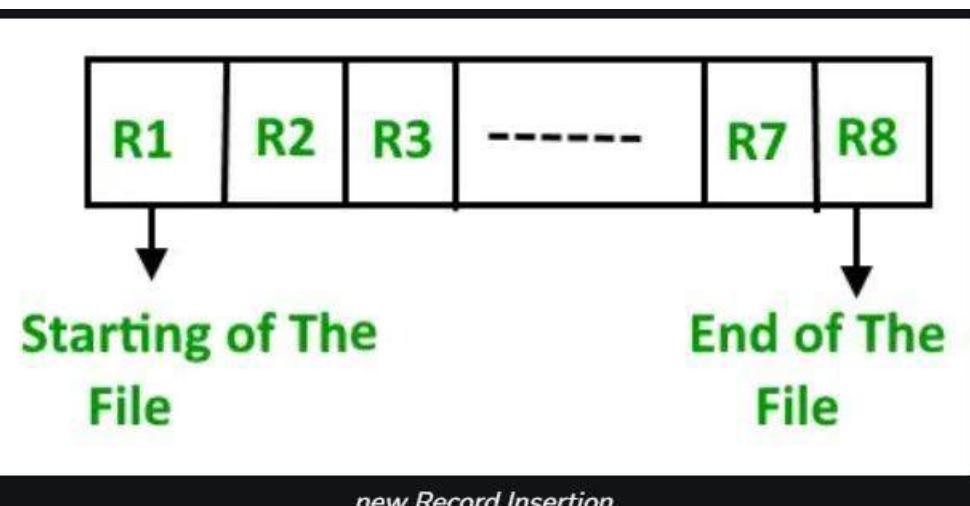
2. Sorted File Method

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.



Sorted File Method

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 has to 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.

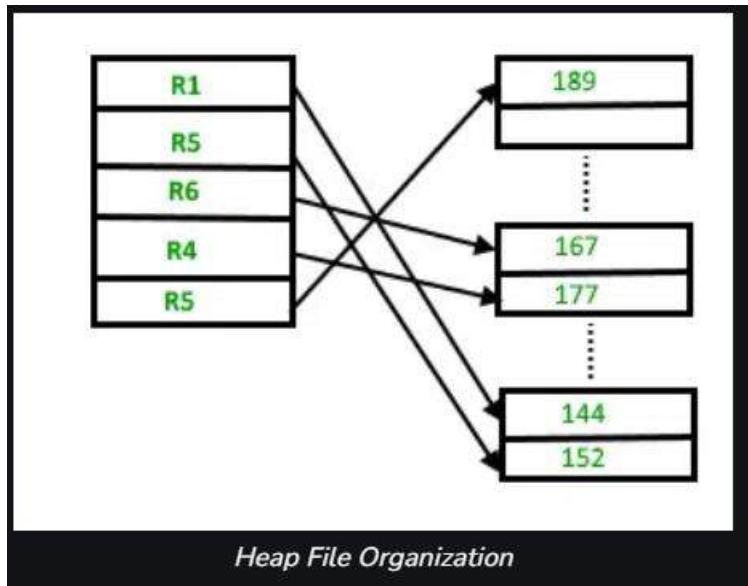
Disadvantages of Sequential File Organization

- 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.

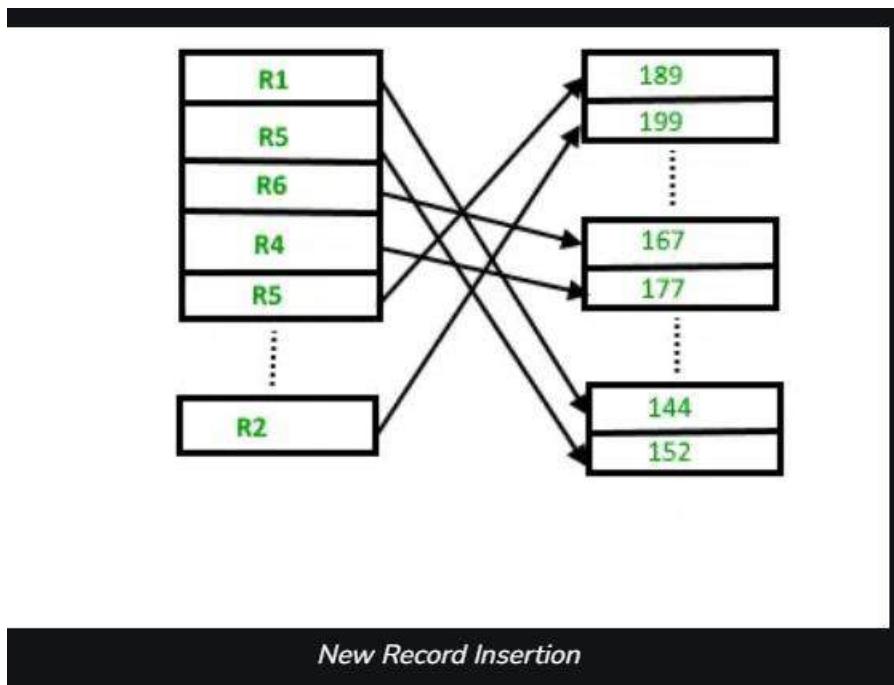
Heap File Organization

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.

In a [database management system](#), When we want to retrieve a particular data, It becomes very inefficient to search all the index values and reach the desired data. In this situation, Hashing technique comes into the picture.

Hashing is an efficient technique to directly search the location of desired data on the disk without using an index structure. Data is stored at the data blocks whose address is generated by using a hash function. The memory location where these records are stored is called a data block or data bucket.

Hash File Organization

- **Data bucket** – Data buckets are the memory locations where the records are stored. These buckets are also considered Units of Storage.
- **Hash Function** – The hash function is a mapping function that maps all the sets of search keys to the actual record address. Generally, the hash function uses the primary key to generate the hash index – the address of the data block. The hash function can be a simple mathematical function to any complex mathematical function.
- **Hash Index**-The prefix of an entire hash value is taken as a hash index. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2^n buckets. When all these bits are consumed? then the depth value is increased linearly and twice the buckets are allocated.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if we want to generate an address for STUDENT_ID = 104 using a mod (5) [hash function](#), it always results in the same bucket address 4. There will not be any changes to the bucket address here. Hence a number of data buckets in the memory for this static hashing remain constant throughout.

Operations:

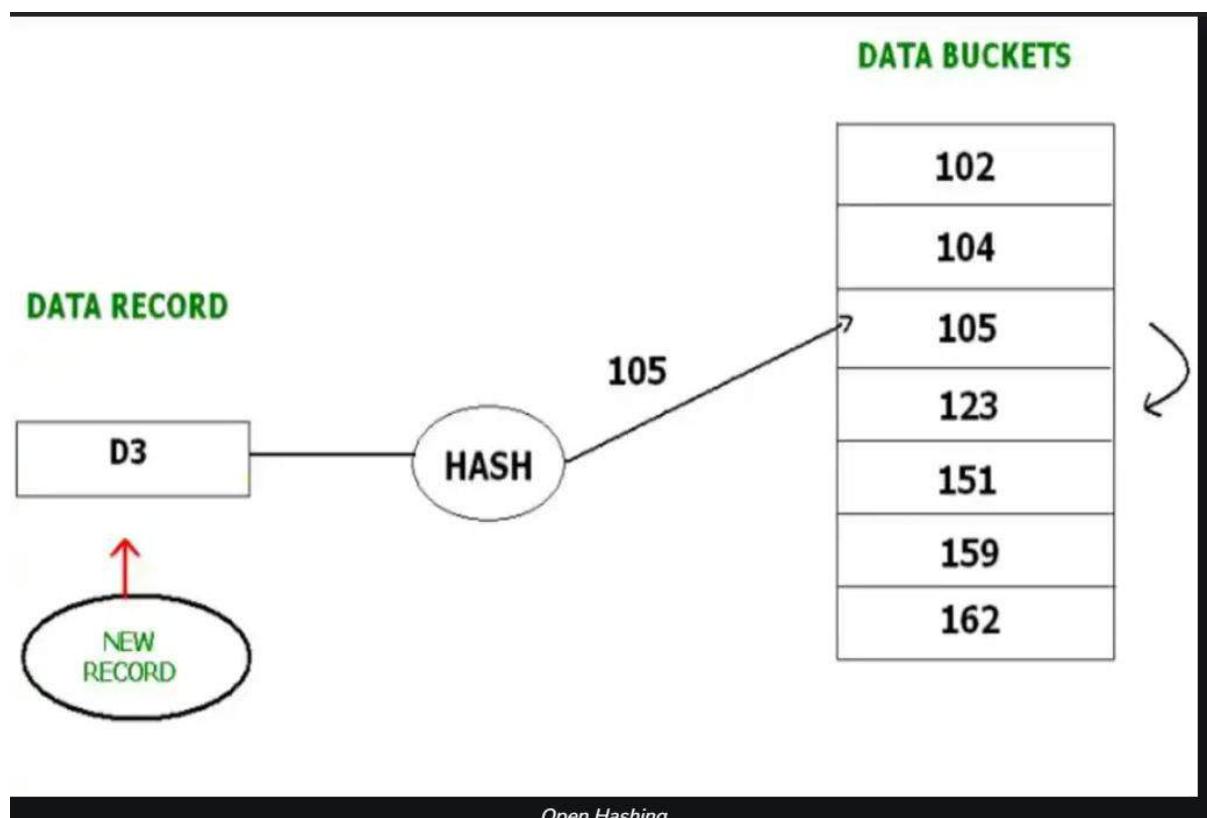
- **Insertion** – When a new record is inserted into the table, The hash function h generates a bucket address for the new record based on its hash key K . Bucket address = $h(K)$
- **Searching** – When a record needs to be searched, The same hash function is used to retrieve the bucket address for the record. For Example, if we want to retrieve the whole record for ID 104, and if the hash function is mod (5) on that ID, the bucket address generated would be 4. Then we will directly go to address 4 and retrieve the whole record for ID 104. Here ID acts as a hash key.

- **Deletion** – If we want to delete a record, Using the hash function we will first fetch the record which is supposed to be deleted. Then we will remove the records for that address in memory.
- **Updation** – The data record that needs to be updated is first searched using the hash function, and then the data record is updated.

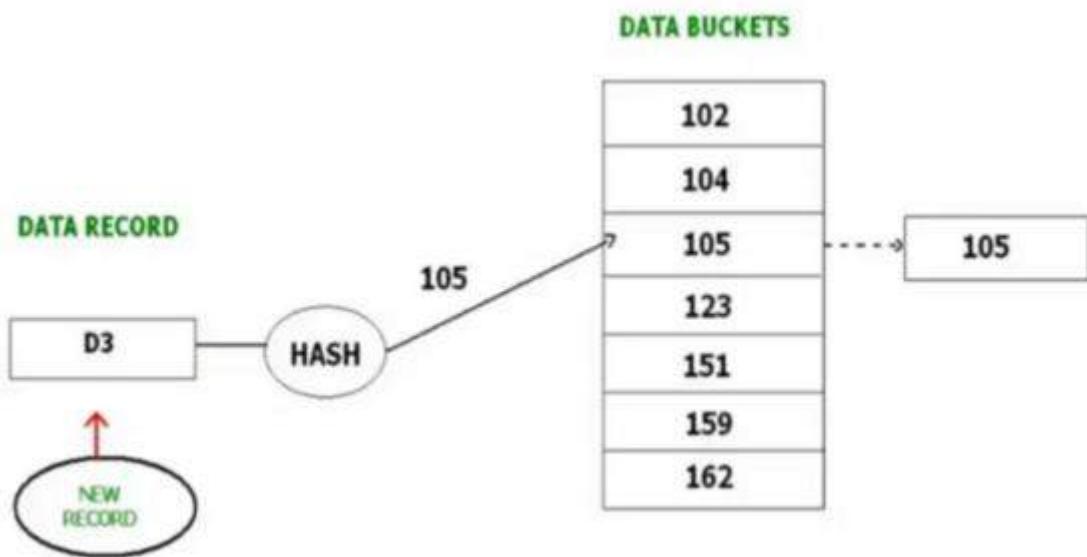
Now, If we want to insert some new records into the file But the data bucket address generated by the hash function is not empty or the data already exists in that address. This becomes a critical situation to handle. This situation is static hashing is called **bucket overflow**. How will we insert data in this case? There are several methods provided to overcome this situation.

Some commonly used methods are discussed below:

- **Open Hashing** – In the [Open hashing](#) method, the next available data block is used to enter the new record, instead of overwriting the older one. This method is also called linear probing. For example, D3 is a new record that needs to be inserted, the hash function generates the address as 105. But it is already full. So the system searches the next available data bucket, 123, and assigns D3 to it.



Closed hashing – In the Closed hashing method, a new data bucket is allocated with the same address and is linked to it after the full data bucket. This method is also known as overflow chaining. For example, we have to insert a new record D3 into the tables. The static hash function generates the data bucket address as 105. But this bucket is full to store the new data. In this case, a new data bucket is added at the end of the 105 data bucket and is linked to it. The new record D3 is inserted into the new bucket.



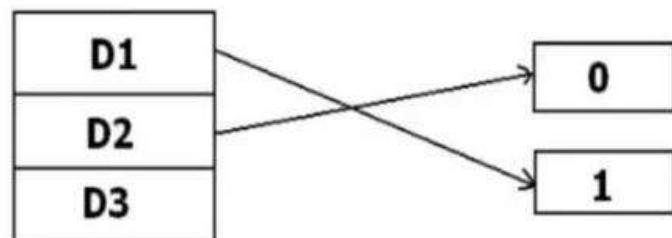
Closed Hashing

- **Quadratic probing:** [Quadratic probing](#) is very much similar to open hashing or linear probing. Here, The only difference between old and new buckets is linear. The quadratic function is used to determine the new bucket address.
- **Double Hashing:** [Double Hashing](#) is another method similar to linear probing. Here the difference is fixed as in linear probing, but this fixed difference is calculated by using another hash function. That's why the name is double hashing.

Dynamic Hashing

The drawback of static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. In [Dynamic hashing](#), data buckets grow or shrink (added or removed dynamically) as the records increase or decrease. Dynamic hashing is also known as extended hashing. In dynamic hashing, the hash function is made to produce a large number of values. For Example, there are three data records D1, D2, and D3. The hash function generates three addresses 1001, 0101, and 1010 respectively. This method of storing considers only part of this address – especially only the first bit to store the data. So it tries to load three of them at addresses 0 and 1.

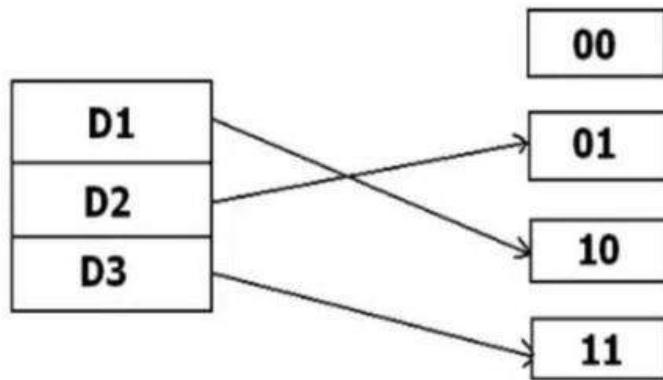
$$\begin{aligned}
 h(D1) &\rightarrow 1001 \\
 h(D2) &\rightarrow 0101 \\
 h(D3) &\rightarrow 1010
 \end{aligned}$$



dynamic hashing

But the problem is that No bucket address is remaining for D3. The bucket has to grow dynamically to accommodate D3. So it changes the address to have 2 bits rather than 1 bit, and then it updates the existing data to have a 2-bit address. Then it tries to accommodate D3.

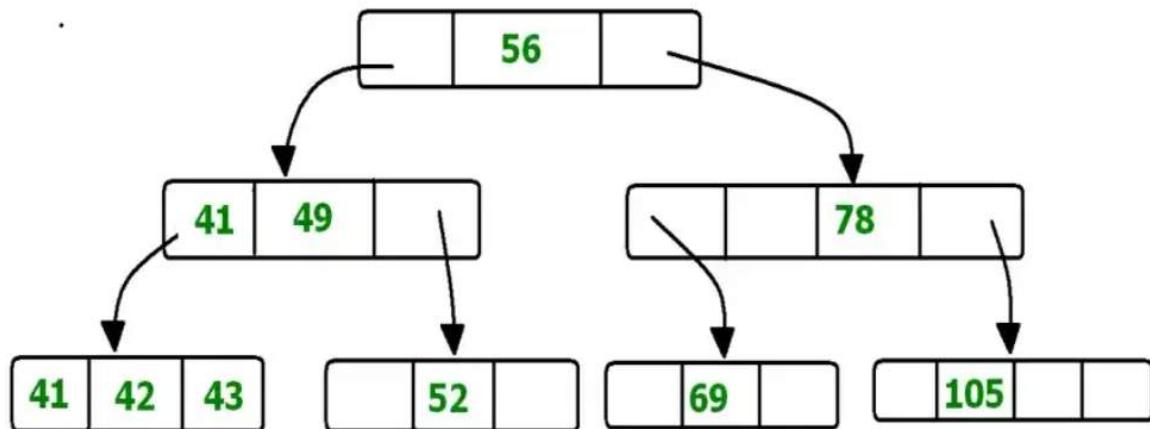
$h(D1) \rightarrow 1001$
 $h(D2) \rightarrow 0101$
 $h(D3) \rightarrow 1010$



dynamic hashing

B+ Tree, as the name suggests, uses a tree-like structure to store records in a File. It uses the concept of Key indexing where the primary key is used to sort the records. For each primary key, an index value is generated and mapped with the record. An index of a record is the address of the record in the file.

B+ Tree is very similar to a binary search tree, with the only difference being that instead of just two children, it can have more than two. All the information is stored in a leaf node and the intermediate nodes act as a pointer to the leaf nodes. The information in leaf nodes always remains a sorted sequential linked list.



B+ Tree File Organization

In the above diagram, 56 is the root node which is also called the main node of the tree. The intermediate nodes here, just consist of the address of leaf nodes. They do not contain any actual records. Leaf nodes consist of the actual record. All leaf nodes are balanced.

Advantages of B+ Tree File Organization

- [Tree traversal](#) is easier and faster.
- Searching becomes easy as all records are stored only in leaf nodes and are sorted in sequentially linked lists.
- There is no restriction on B+ tree size. It may grow/shrink as the size of the data increases/decreases.

Disadvantages of B+ Tree File Organization

- Inefficient for static tables.

Cluster File Organization

In **Cluster file organization**, two or more related tables/records are stored within the same file known as clusters. These files will have two or more tables in the same data block and the key attributes which are used to map these tables together are stored only once.

Thus it lowers the cost of searching and retrieving various records in different files as they are now combined and kept in a single cluster. For example, we have two tables or relation Employee and Department. These tables are related to each other.

EMPLOYEE				DEPARTMENT	
EMP_ID	EMP_NAME	EMP_ADD	DEP_ID	DEP_ID	DEP_NAME
01	JOE	CAPE TOWN	D_101	D_101	ECO
02	ANNIE	FRANSISCO	D_103	D_102	CS
03	PETER	CROY CITY	D_101	D_103	JAVA
04	JOHN	FRANSISCO	D_102	D_104	MATHS
05	LUNA	TOKYO	D_106	D_105	BIO
06	SONI	W.LAND	D_105	D_106	CIVIL
07	SAKACHI	TOKYO	D_104		
08	MARY	NOVI	D_101		

Cluster File Organization

Therefore this table is allowed to combine using a [join operation](#) and can be seen in a cluster file.

CLUSTER KEY



DEP_ID	DEP_NAME	EMP_ID	EMP_NAME	EMP_ADD
D_101	ECO	01	JOE	CAPE TOWN
		02	PETER	CROY CITY
		03	MARY	NOVI
D_102	CS	04	JOHN	FRANSISCO
D_103	JAVA	05	ANNIE	FRANSISCO
D_104	MATHS	06	SAKACHI	TOKYO
D_105	BIO	07	SONI	W.LAND
D_106	CIVIL	08	LUNA	TOKYO

DEPARTMENT + EMPLOYEE

Cluster File Organization

If we have to insert, update or delete any record we can directly do so. Data is sorted based on the primary key or the key with which searching is done. The **cluster key** is the key with which the joining of the table is performed.

Types of Cluster File Organization

There are two ways to implement this method.

- **Indexed Clusters:** In Indexed clustering, the records are grouped based on the cluster key and stored together. The above-mentioned example of the Employee and Department relationship is an example of an Indexed Cluster where the records are based on the Department ID.
- **Hash Clusters:** This is very much similar to an indexed cluster with the only difference that instead of storing the records based on cluster key, we generate a hash key value and store the records with the same hash key value.

Advantages of Cluster File Organization

- It is basically used when multiple tables have to be joined with the same joining condition.
- It gives the best output when the cardinality is 1:m.

Disadvantages of Cluster File Organization

- It gives a low performance in the case of a large database.
- In the case of a 1:1 cardinality, it becomes ineffective.

ISAM (Indexed Sequential Access Method):

A combination of sequential and indexed methods. Data is stored sequentially, but an index is maintained for faster access. Think of it like having a bookmark in a book that guides you to specific pages.

Advantages of ISAM :

- Faster retrieval compared to pure sequential methods.
- Suitable for applications with a mix of sequential and random access.

Disadvantages of ISAM :

- Index maintenance can add overhead in terms of storage and update operations.
- Not as efficient as fully indexed methods for random access.

LOGICAL DATABASE DESIGN

Logical database design in database management systems (DBMS) is the process of defining the structure of the database without considering the physical implementation details. It focuses on creating a conceptual model of the database that accurately represents the data and its relationships, ensuring efficient storage, retrieval, and manipulation of data.

Step-by-step approach to logical database design:

1. Requirement Analysis: Understand the requirements of the users and the organization. This involves gathering information about the data to be stored, its relationships, constraints, and any other relevant aspects.
2. Entity-Relationship (ER) Modeling: Create an Entity-Relationship Diagram (ERD) to represent the entities (objects or concepts about which data is stored), their attributes, and the relationships between them. Entities are represented as rectangles, attributes as ovals, and relationships as lines connecting entities.
3. Normalization: Apply normalization techniques to ensure that the database schema is free from anomalies such as redundancy, update anomalies, and insertion anomalies. Normalization typically involves decomposing the database schema into multiple tables to minimize redundancy and dependency.
4. Data Integrity Constraints: Define integrity constraints to enforce data integrity rules, such as primary key constraints, foreign key constraints, unique constraints, and domain constraints. These constraints ensure that the data remains accurate and consistent over time.

5. Data Types and Constraints: Specify the data types for each attribute based on the nature of the data (e.g., integer, string, date) and define any additional constraints (e.g., not null, default values) to enforce data consistency and validity.
6. Indexing: Identify the attributes that will be frequently used for searching and retrieval operations and create appropriate indexes to optimize query performance.
7. Views and Security: Define views to present subsets of data to users based on their requirements and access privileges. Implement security mechanisms to control access to the database objects and ensure data confidentiality and integrity.
8. Transaction Management: Design transaction management mechanisms to ensure the atomicity, consistency, isolation, and durability (ACID properties) of database transactions.
9. Performance Tuning: Fine-tune the database design by analyzing and optimizing the schema, queries, indexes, and other database objects to improve performance and scalability.
10. Documentation and Maintenance: Document the database design including data dictionaries, ER diagrams, schema diagrams, and other relevant documentation to facilitate maintenance and future enhancements.

By following these steps, you can create a well-structured and efficient logical database design that meets the requirements of the users and ensures the integrity and performance of the database system.

Let's create a simplified example of logical database design for a university system.

1. Requirement Analysis:

- The university system needs to store information about students, courses, and enrollments.
- Each student has a unique student ID, name, and date of birth.
- Each course has a unique course code, title, and credit hours.
- Students can enroll in multiple courses, and each course can have multiple students.

2. Entity-Relationship (ER) Modeling:

3. Normalization:

- We identify functional dependencies and normalize the schema to eliminate anomalies.
- For example, we might decompose a Student-Course table into separate Student and Course tables to eliminate redundancy.

4. Data Integrity Constraints:

- Define primary key constraints for Student ID and Course Code to ensure uniqueness.
- Define foreign key constraints to enforce referential integrity between the Student and Enrollment tables, and between the Course and Enrollment tables.

5. Data Types and Constraints:

- Student ID: integer, primary key
- Name: string
- Date of Birth: date
- Course Code: string, primary key
- Title: string
- Credit Hours: integer
- Enrollment Date: date

- Grade: string (nullable)

6. Indexing:

- Create indexes on Student ID and Course Code columns for efficient searching and retrieval.
- Optionally, index the Enrollment Date column for queries involving date-based filtering.

Primary Key

7. Views and Security:

- Define views to present information such as student transcripts or course schedules.
- Implement role-based access control to restrict access to sensitive data.

8. Transaction Management:

- Implement transaction management mechanisms to ensure ACID properties for database transactions.
- Use transaction control statements (e.g., BEGIN TRANSACTION, COMMIT, ROLLBACK) to manage transactions.

9. Performance Tuning:

- Analyze query performance using query execution plans and optimize queries using appropriate indexing and query optimization techniques.
- Monitor database performance metrics and tune database configuration parameters as needed.

10. Documentation and Maintenance:

- Document the database schema, constraints, indexes, and other relevant information.
- Maintain the documentation and update it as the database schema evolves over time.

This example illustrates the logical database design process for a university system, starting from requirements analysis to documentation and maintenance.

E.F.Codd's Rules in DBMS:

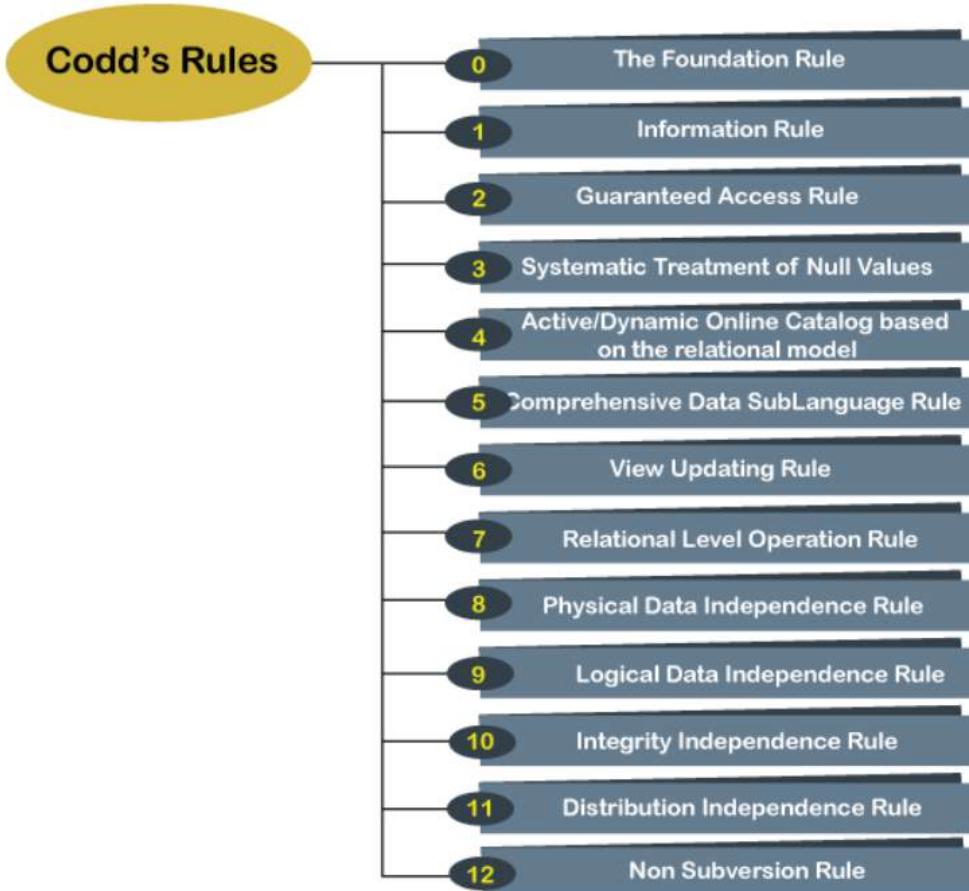
EF Codd is a computer scientist named Dr. Edgar F. Codd who first outlined the relational model which now became the most popular and one and only database model

Codd Proposed 13 rules (listed from 0 to 12) popularly known as codd's 12 rules which are used as a yardstick to test the quality of relational database management system(RDBMS)

These rules are made to ensure data integrity, consistency, and usability. This set of rules basically signifies the characteristics and requirements of a relational database management system (RDBMS).

Till now none of the commercial product has followed all the 13 rules even Oracle has followed 8.5 out of 13

Codd's Rules in DBMS:



Rule 0 – Foundation rule

Any relational database management system that is propounded to be RDBMS or advocated to be a RDBMS should be able to manage the stored data in its entirety through its relational capabilities.

If a system is said to be an RDBMS then the database should be managed using only relational capabilities

Rule 1: The Information Rule

All information, whether it is user information or metadata, that is stored in a database must be entered as a value in a cell of a table. It is said that everything within the database is organized in a table layout.

Rule 2: The Guaranteed Access Rule

Each data element is guaranteed to be accessible logically with a combination of the table name, primary key (row value), and attribute name (column value).

Example : emp+empid+ename,sal

Strictly the data must not be accessed via a pointer .

Rule 3: Systematic Treatment of NULL Values

Every Null value in a database must be given a systematic and uniform treatment.

Null values represent different situations it may be missing data or not applicable or no value situation.

Null values must be handled consistently and also primary key must not be null and any expression on null must give null.

Rule 4: Active Online Catalog Rule

The database catalog, which contains metadata about the database, must be stored and accessed using the same relational database management system.

Database dictionary is a catalog which shows structural description of the complete database and it must be stored online

This rule states that a database dictionary must be governed by the same rules and same query language as used for general database.

Rule 5: The Comprehensive Data Sublanguage Rule

A crucial component of any efficient database system is its ability to offer an easily understandable data manipulation language (DML) that facilitates defining, querying, and modifying information within the database.

The database should be accessible through a language which supports definition, manipulation and all transaction management activities, such a language is called structured language

For example, SQL, if database uses a different language for data access and manipulation then it is a violation of the rule.

Rule 6: The View Updating Rule

All views that are theoretically updatable must also be updatable by the system.

Difference views created for different purposes should be automatically updated by the system itself.

Rule 7: High-level Insert, Update, and Delete

A successful database system must possess the feature of facilitating high-level insertions, updates, and deletions that can grant users the ability to conduct these operations with ease through a single query.

Operations like insert, delete and update operations must be supported at each level of relation even though it might be a nested relation or a complex relation

Set operations like union, intersection, minus must be supported.

Rule 8: Physical Data Independence

Application programs and activities should remain unaffected when changes are made to the physical storage structures or methods.

Any change in the physical location of the table should not reflect the change at the application level

Example: If you rename or move a file from one disk to another then it should not affect the application.

Rule 9: Logical Data Independence or Integrity Independence

Application programs and activities should remain unaffected when changes are made to the logical structure of the data, such as adding or modifying tables.

If there are any changes done to the logical structure of the database table, then users view of data should not be changed

If the table is split into two tables, then a new view should give result as the join of these two tables but this rule is very difficult to satisfy.

Rule 10: Integrity Independence

Integrity constraints should be specified separately from application programs and stored in the catalog. They should be automatically enforced by the database system.

Database table should design itself on integrity rather than using external programs.

It should use primary keys, check constants triggers, etc which makes our DBMS independent of the front end application.

Rule 11: Distribution Independence

The distribution of data across multiple locations should be invisible to users, and the database system should handle the distribution transparently.

Data distribution over various geographical locations over a network should not reflect the end-user i.e you should feel that all the data is stored in a single place

This rule laid the foundation for the distributed database.

Rule 12: Non-Subversion Rule

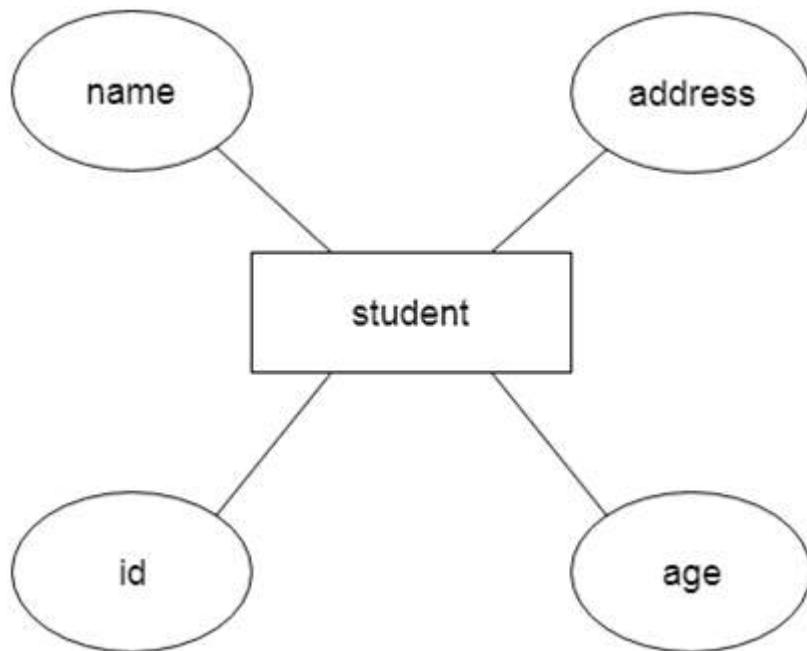
If the interface of the system is providing access to low-level records, then the interface must not be able to damage the system and bypass security and integrity constraints.

Any access given to the data that is present in the lowest level must not give a chance to authenticate constraints and change data ,this can be achieved through some kind of encryption techniques.

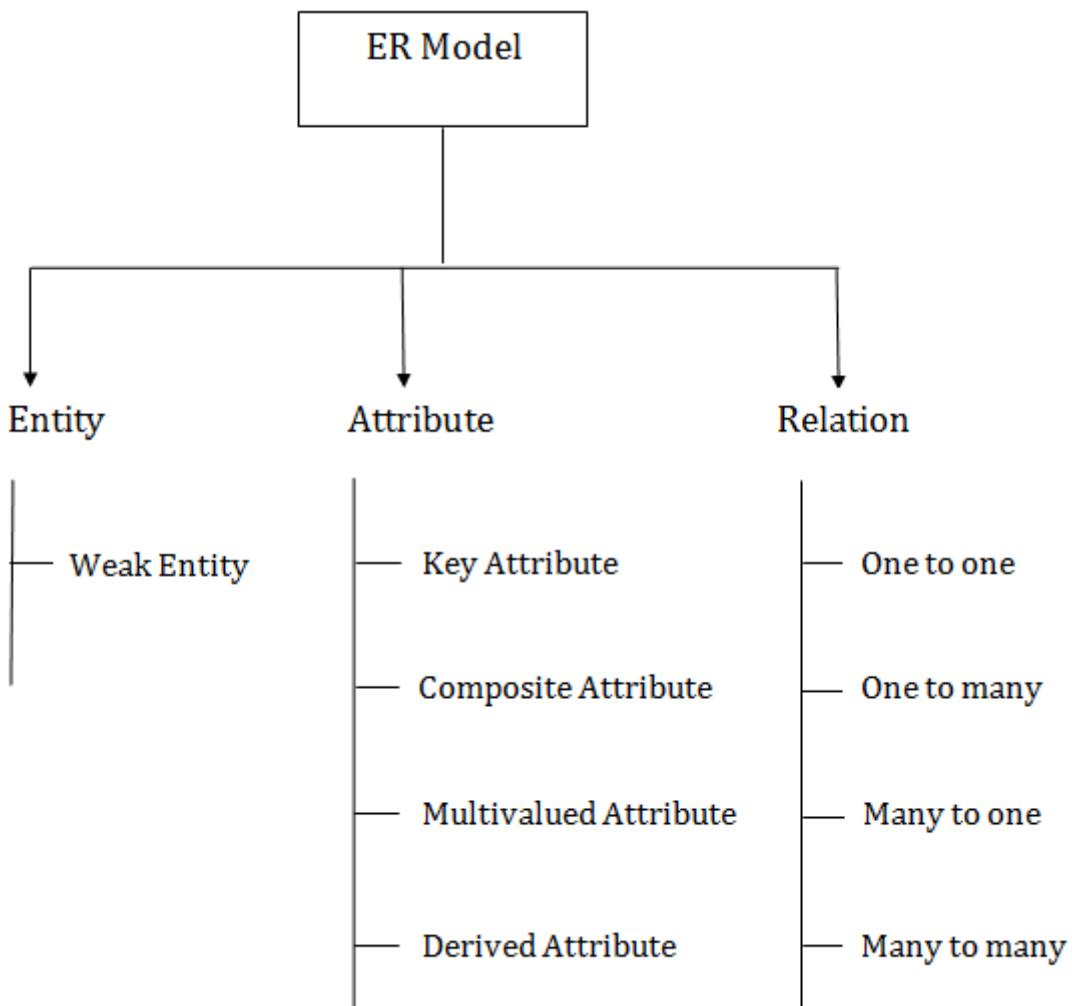
ER (Entity Relationship) Diagram in DBMS

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.



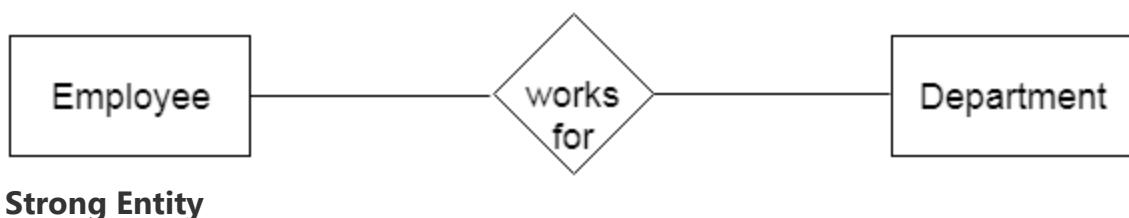
Component of ER Diagram



1 / Entity:

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.



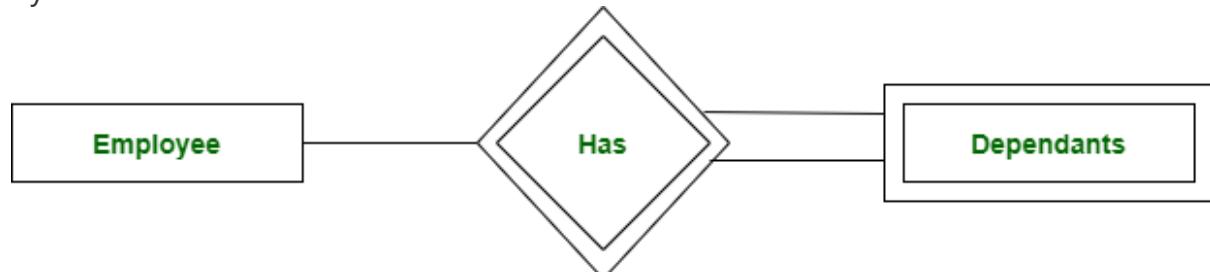
A **Strong Entity** is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

2. 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 can't 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 can't exist 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.



Strong Entity and Weak Entity

a. Weak Entity

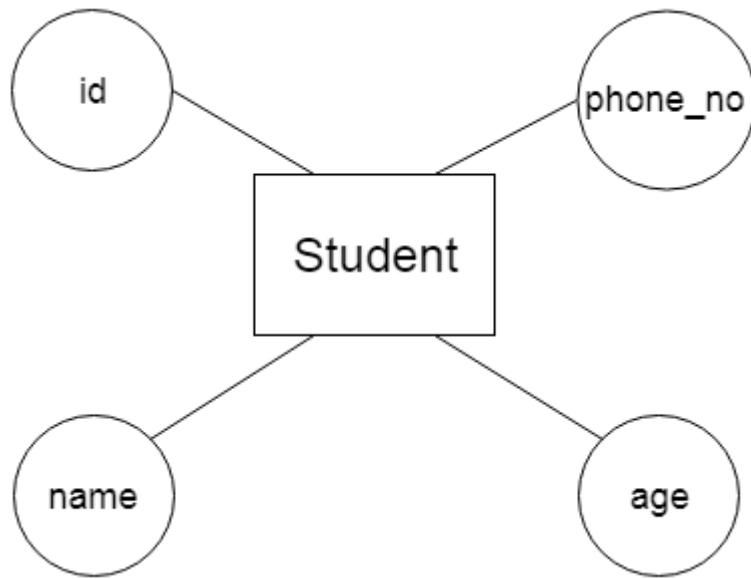
An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



2. Attribute

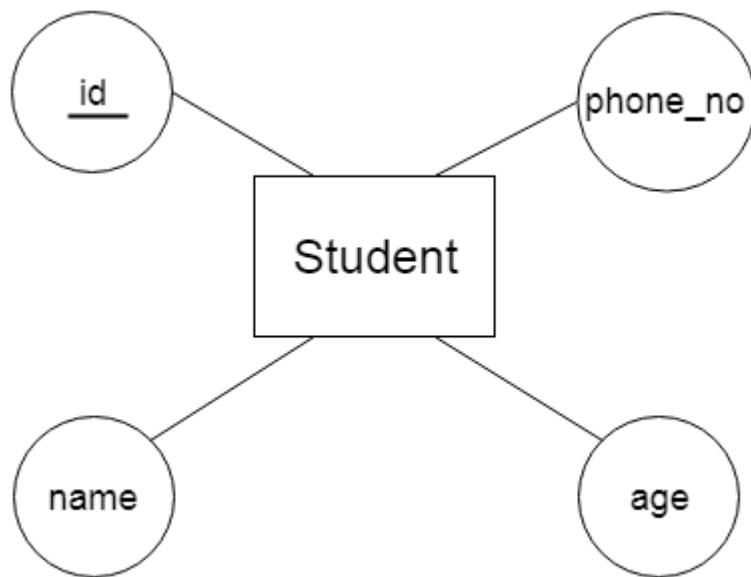
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.



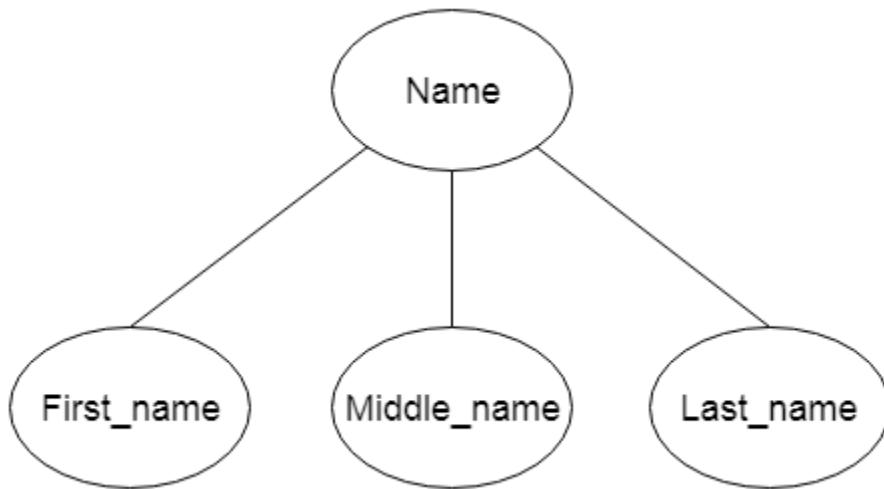
a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



b. Composite Attribute

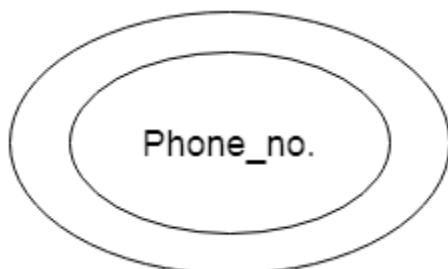
An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

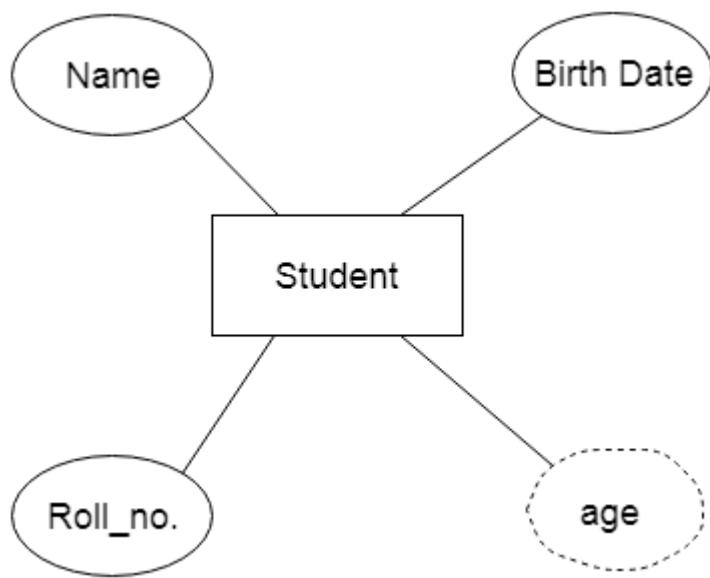
For example, a student can have more than one phone number.



d. Derived Attribute

An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.



~~3. Relationship~~

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.



Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

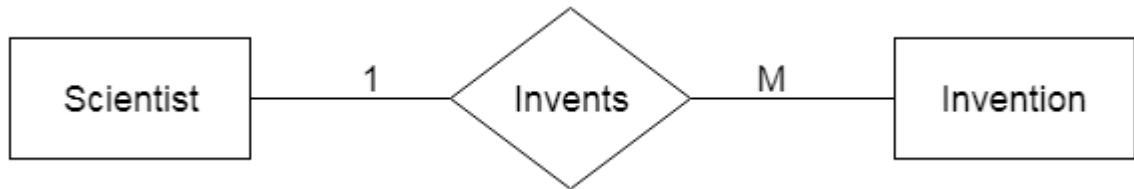
For example, A female can marry to one male, and a male can marry to one female.



b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

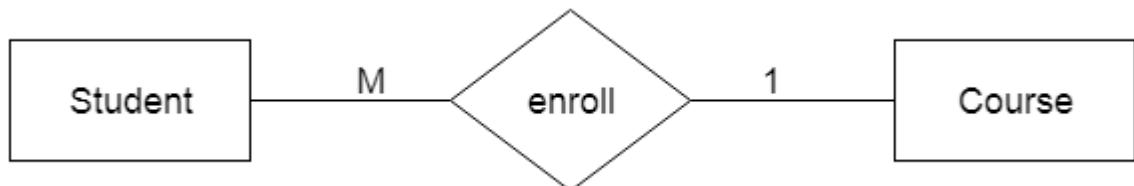
For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.



c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.



Notation of ER diagram

Database can be represented using the notations. In ER diagram, many notations are used to express the cardinality. These notations are as follows:

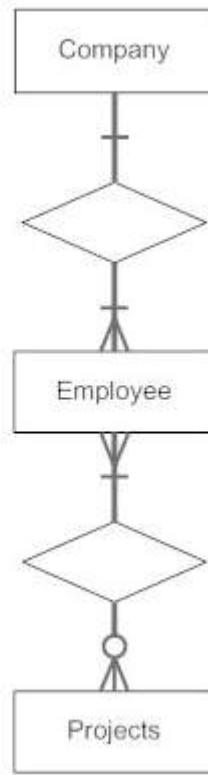
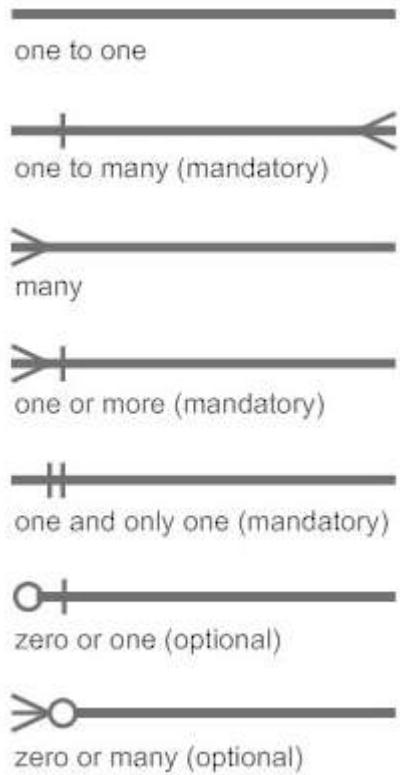


Fig: Notations of ER diagram

CREATING ENTITY RELATIONSHIP DIAGRAM

Entity Relationship Diagram Example:

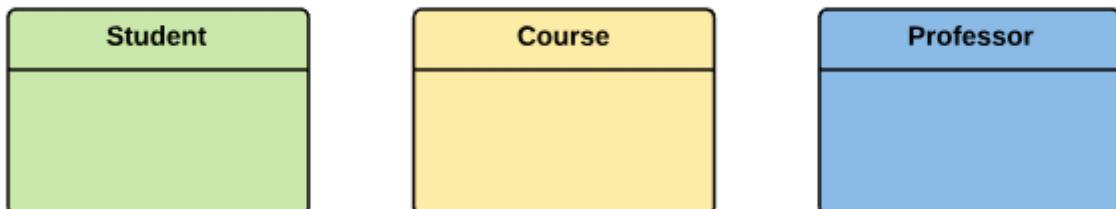
In a university, a Student enrolls in Courses. A student must be assigned to at least one or more Courses. Each course is taught by a single Professor. To maintain instruction quality, a Professor can deliver only one course.



Step 1) Entity Identification

We have three entities

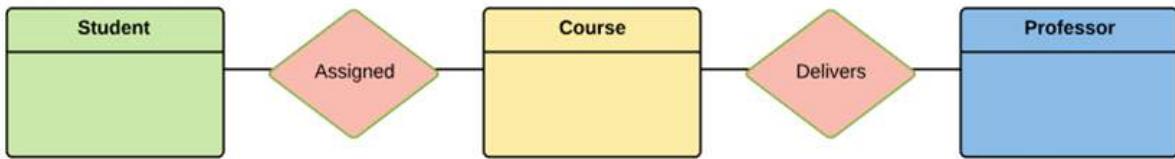
- Student
- Course
- Professor



Step 2) Relationship Identification

We have the following two relationships

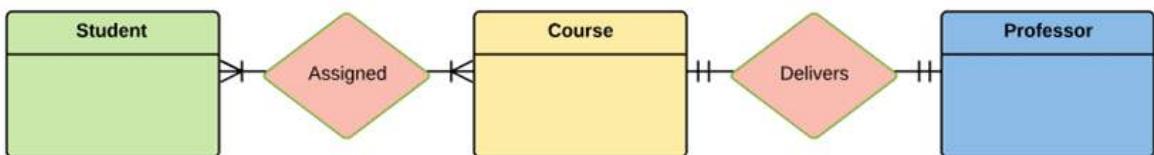
- The student is **assigned** a course
- Professor **delivers** a course



Step 3) Cardinality Identification

From the problem statement we know that,

- A student can be assigned **multiple** courses
- A Professor can deliver only **one** course



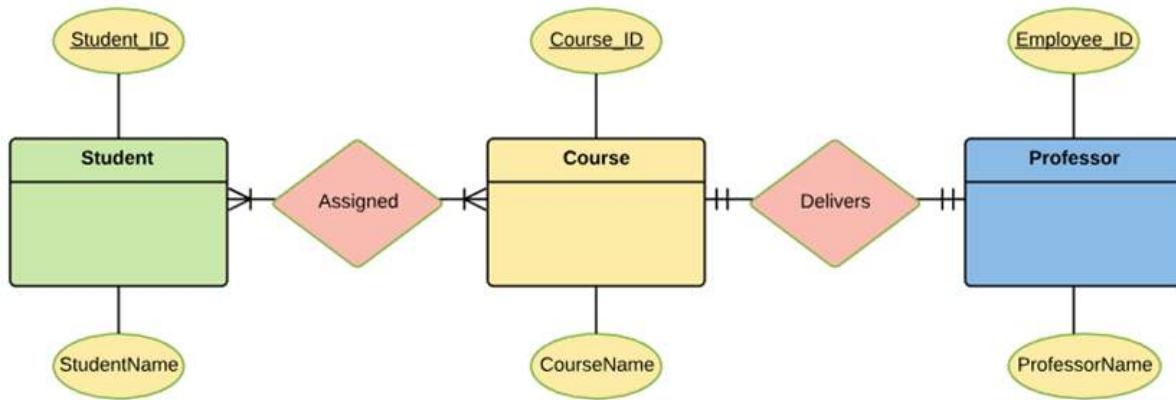
Step 4) Identify Attributes

You need to study the files, forms, reports, data currently maintained by the organization to identify attributes. You can also conduct interviews with various stakeholders to identify entities. Initially, it's important to identify the attributes without mapping them to a particular entity.

Once, you have a list of Attributes, you need to map them to the identified entities. Ensure an attribute is to be paired with exactly one entity. If you think an attribute should belong to more than one entity, use a modifier to make it unique.

Once the mapping is done, identify the primary Keys. If a unique key is not readily available, create one.

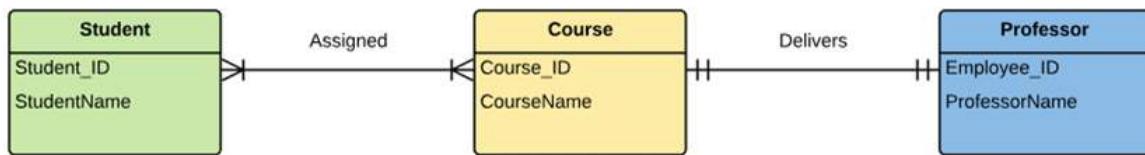
Entity	Primary Key	Attribute
Student	Student_ID	StudentName
Professor	Employee_ID	ProfessorName
Course	Course_ID	CourseName



For Course Entity, attributes could be Duration, Credits, Assignments, etc. For the sake of ease we have considered just one attribute.

Step 5) Create the ERD Diagram

A more modern representation of Entity Relationship Diagram Example



Best Practices for Developing Effective ER Diagrams

Here are some best practice or example for Developing Effective ER Diagrams.

- Eliminate any redundant entities or relationships
- You need to make sure that all your entities and relationships are properly labeled
- There may be various valid approaches to an ER diagram. You need to make sure that the ER diagram supports all the data you need to store
- You should assure that each entity only appears a single time in the ER diagram
- Name every relationship, entity, and attribute are represented on your diagram

- Never connect relationships to each other
- You should use colors to highlight important portions of the ER diagram

Converting ER model into Relations

ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.

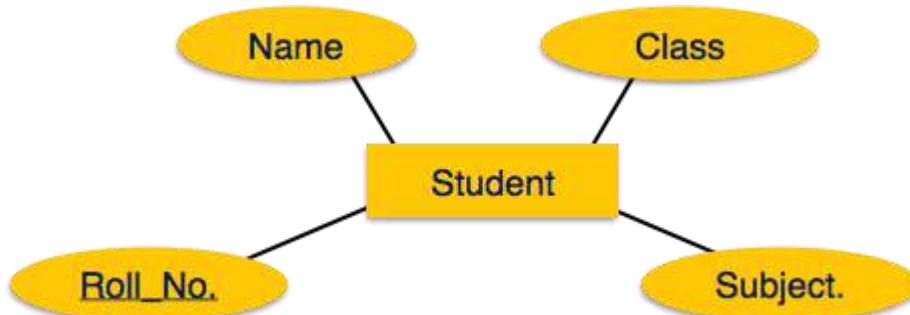
There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual. We may focus here on the mapping diagram contents to relational basics.

ER diagrams mainly comprise of –

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real-world object with some attributes.

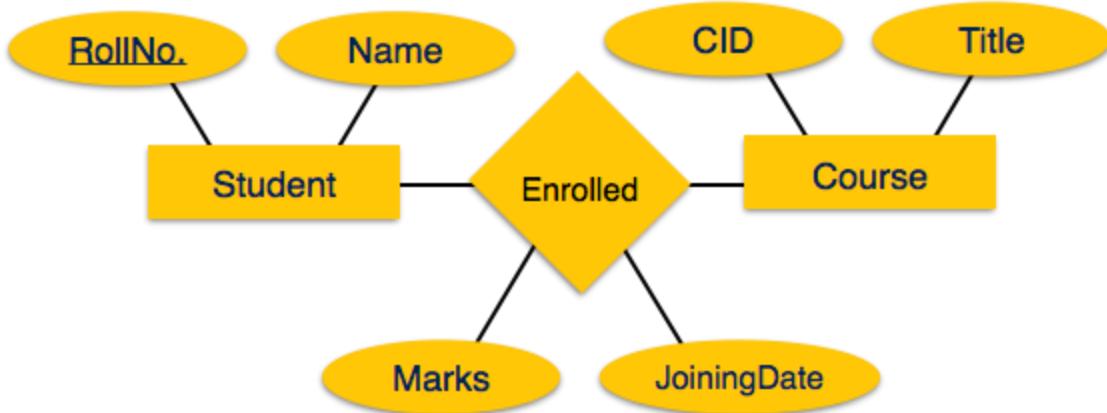


Mapping steps

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

Mapping Relationship

A relationship is an association among entities.

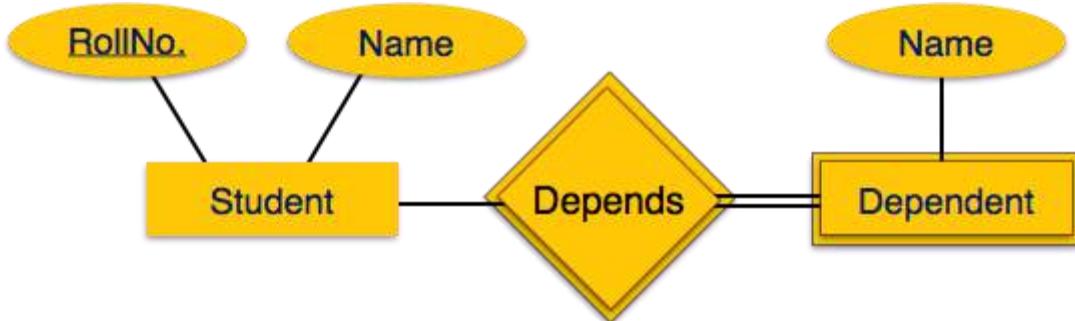


Mapping Process

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity set is one which does not have any primary key associated with it.

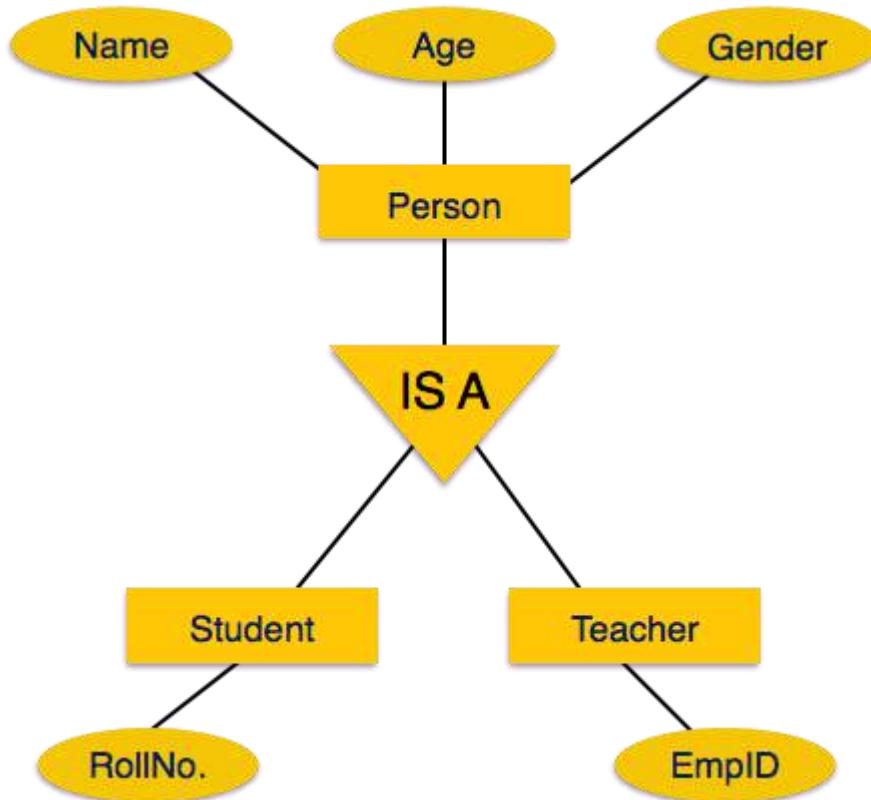


Mapping Process

- Create table for weak entity set.
- Add all its attributes to table as field.
- Add the primary key of identifying entity set.
- Declare all foreign key constraints.

Mapping Hierarchical Entities

ER specialization or generalization comes in the form of hierarchical entity sets.



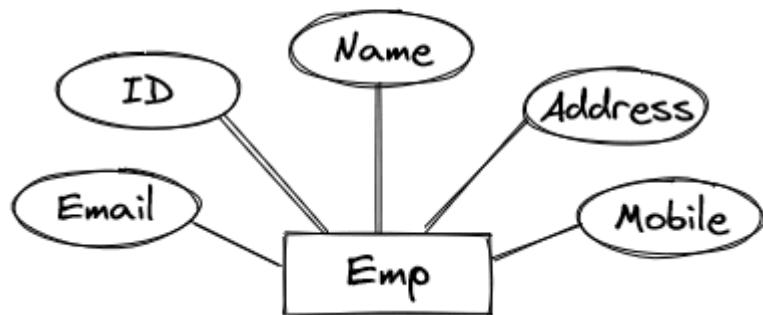
Mapping Process

- Create tables for all higher-level entities.
- Create tables for lower-level entities.
- Add primary keys of higher-level entities in the table of lower-level entities.
- In lower-level tables, add all other attributes of lower-level entities.
- Declare primary key of higher-level table and the primary key for lower-level table.
- Declare foreign key constraints.

Rule 1: Conversion of an entity set into a table

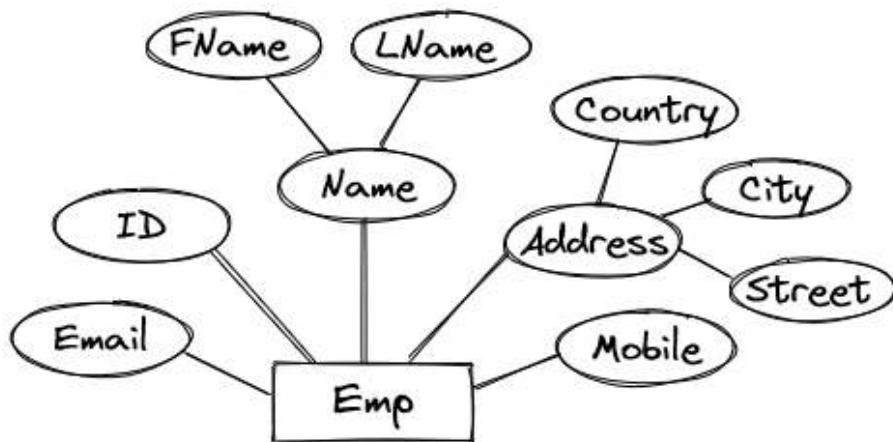
a) Representation of Strong Entity set with simple attributes.

$\text{EMP} (\text{ID}, \text{Name}, \text{Address}, \text{Mobile}, \text{Email})$



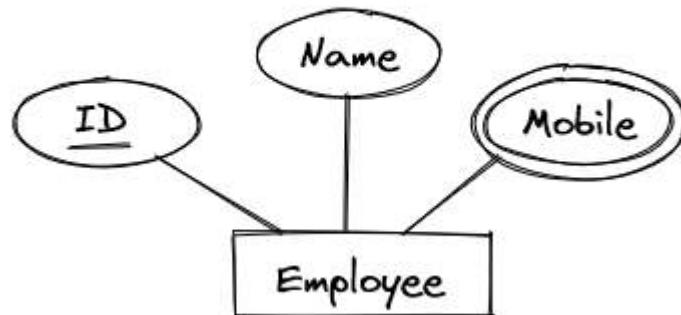
b) Representation of Strong entity set with Composite Attributes.

$\text{EMP} (\text{ID}, \text{FName}, \text{LName}, \text{Country}, \text{city}, \text{Street}, \text{Mobile}, \text{Email})$



c) Representation of Strong entity set with a multi-valued attribute.

Employee (ID, Name, Mobile)

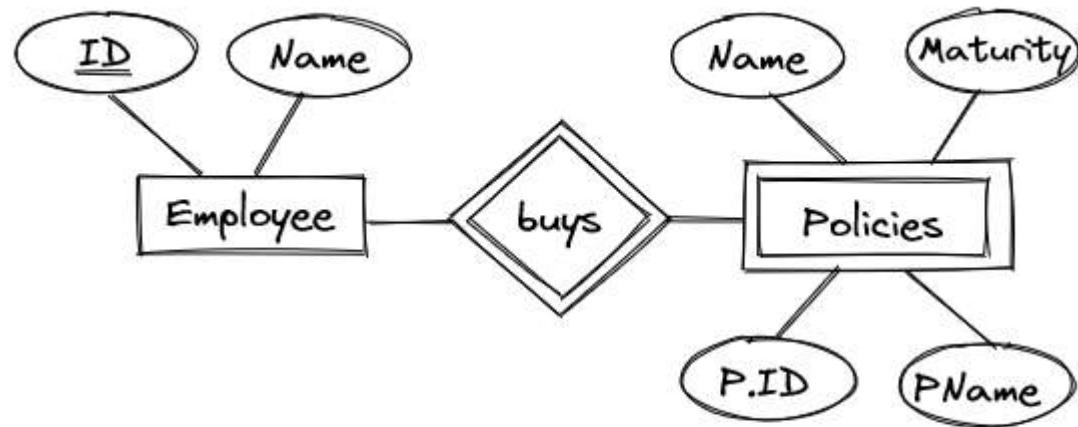


Employee table is further divided as below in order to eliminate redundancy due to multi-valued attribute.

ID	Name
001	Teena
002	Alina
003	Maryam

ID	Mobile1	Mobile2	Mobile3
001	1234	5678	
002	3578		
003	0214	9514	3574

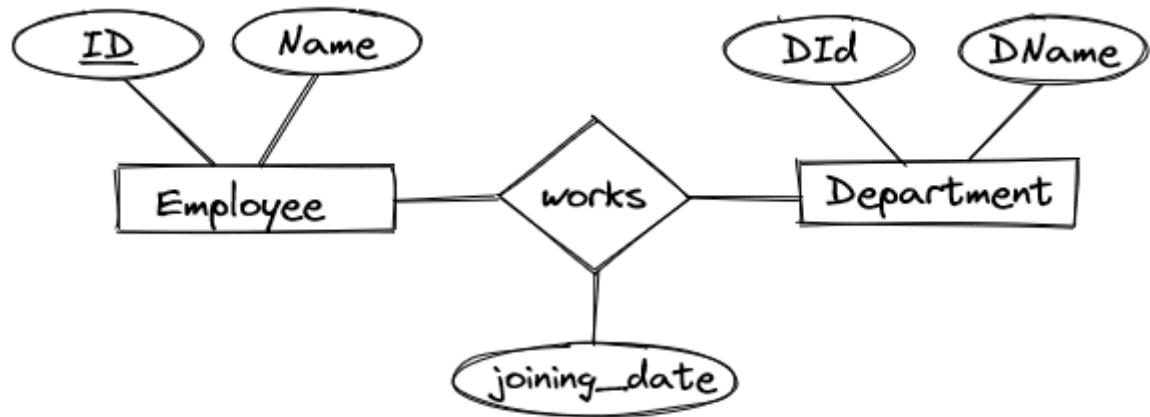
d) Representation of Weak entity set.



1) Employee(ID, Name)

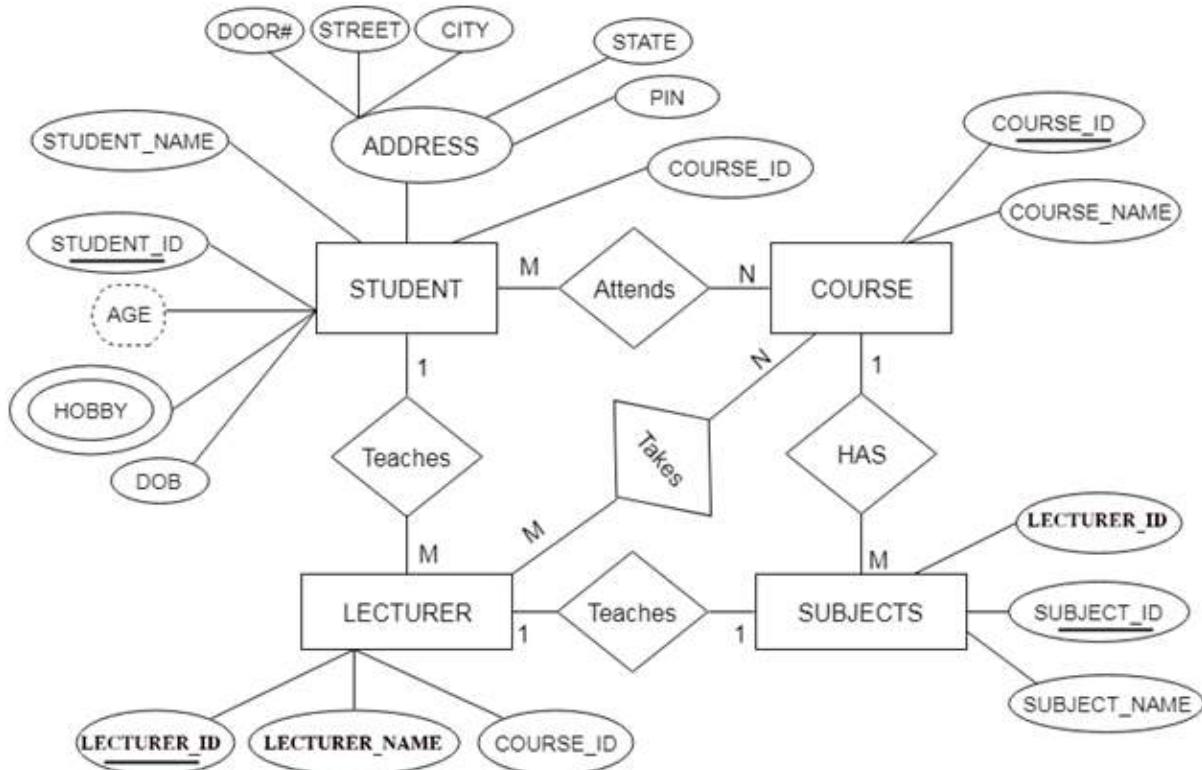
2) Policies(P.ID, ID, Name, PName, Maturity, PName)

Rule 2: Conversion of Relationship into Relation.



1. Employee (Name, ID)
2. Department (DId, DName)
3. works (ID, DId, joining_date)

Example 2: Student management system



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

- **Entity type becomes a table.**

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

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

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

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

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

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

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

- **Composite attribute represented by components.**

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

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

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

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

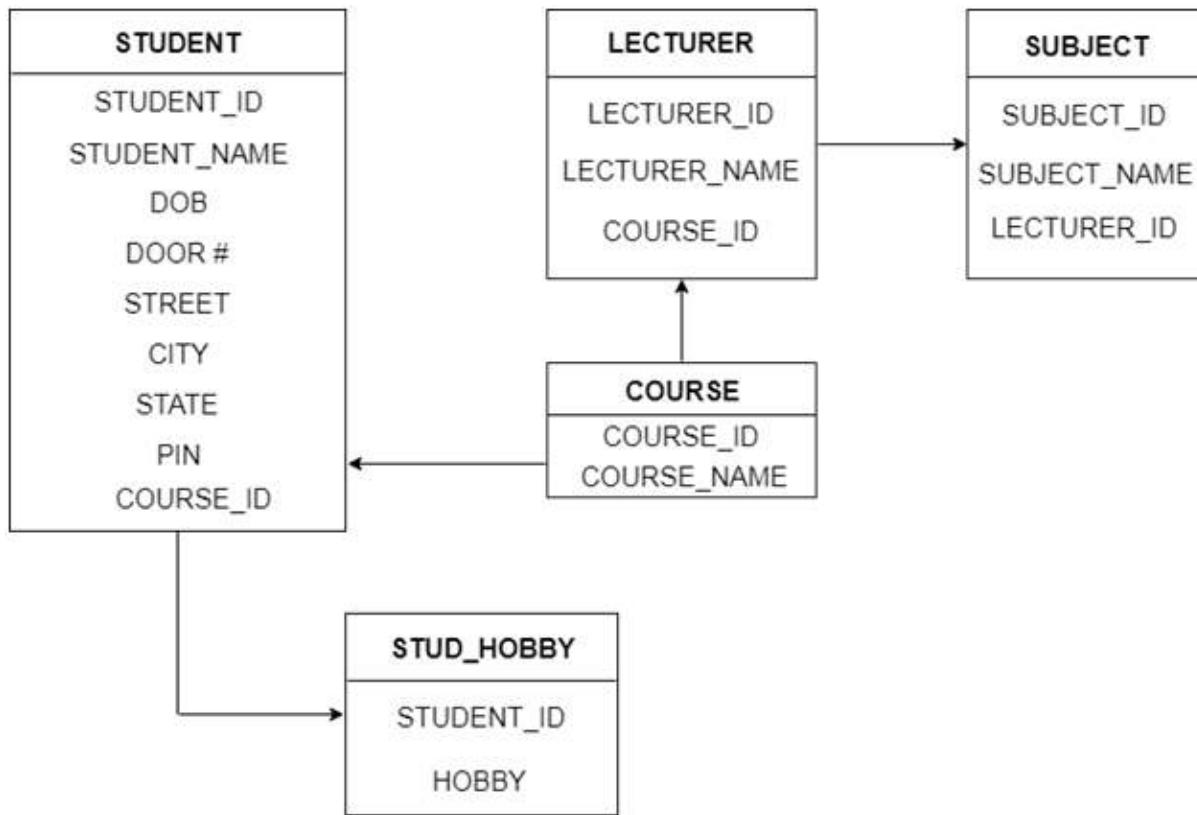
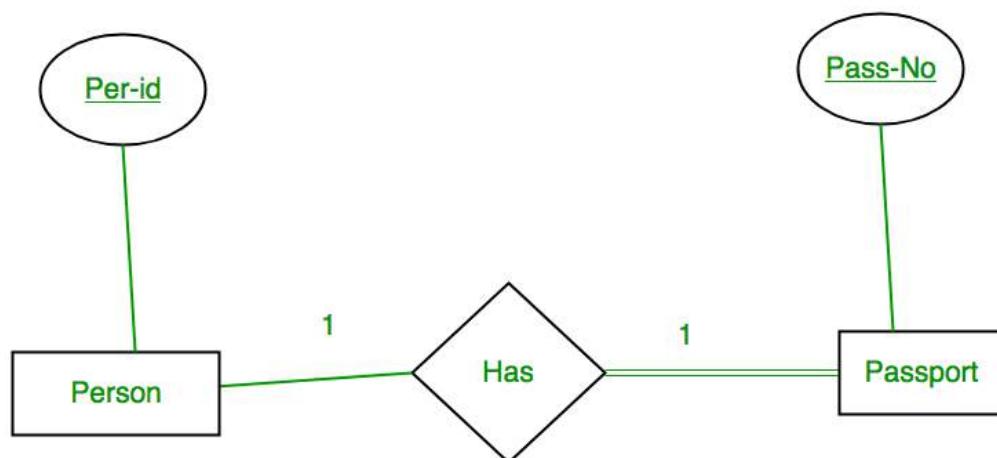


Figure: Table structure

Mapping ER model to Relation with cardinalities:

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



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	–				

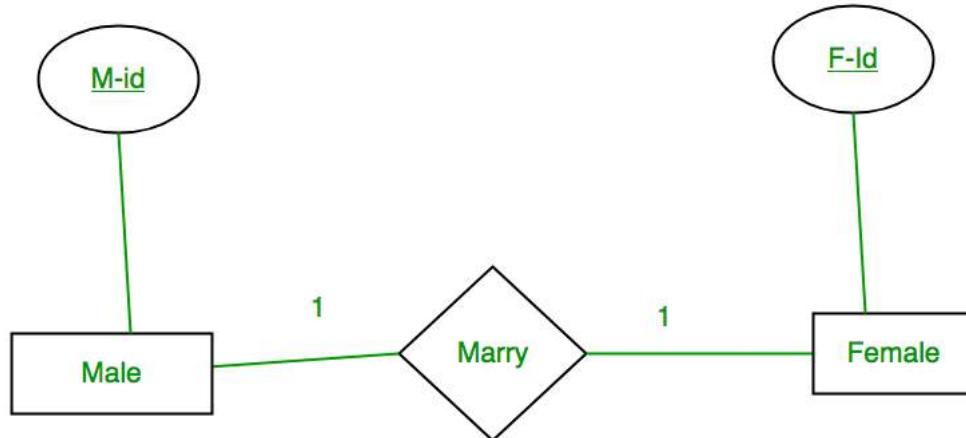
Table 1

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 PassportAttribute

Table 2

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



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	
<u>M-Id</u>	Other Male Attribute	<u>M-Id</u>	F-Id	<u>F-Id</u>	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 can't 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.

<u>M-Id</u>	Other Male Attribute	F-Id
-------------	----------------------	------

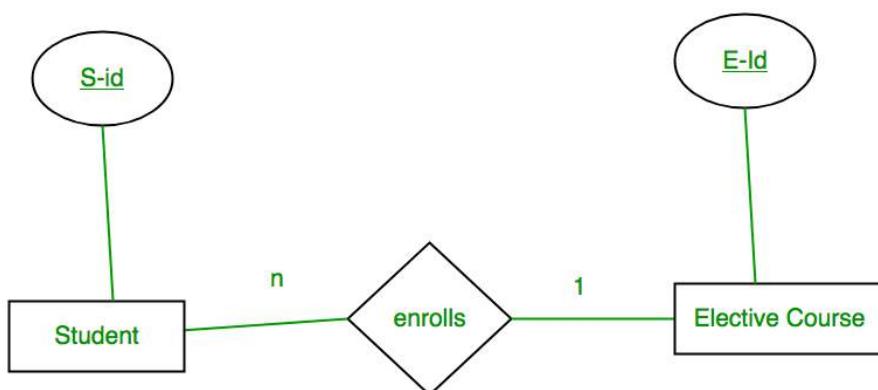
Table 4

<u>F-Id</u>	Other FemaleAttribute
-------------	-----------------------

Table 5

Note: Binary relationship with 1:1 cardinality will have 2 table if partial participation of both entities in the relationship. If atleast 1 entity has total participation, number of tables required will be 1.

Case 3: 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
----------------	--	----------------	--	------------------------

<u>S- Id</u>	Other Student Attribute	<u>S- Id</u>	E- Id	<u>E- Id</u>	Other CourseAttribute	Elective
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.

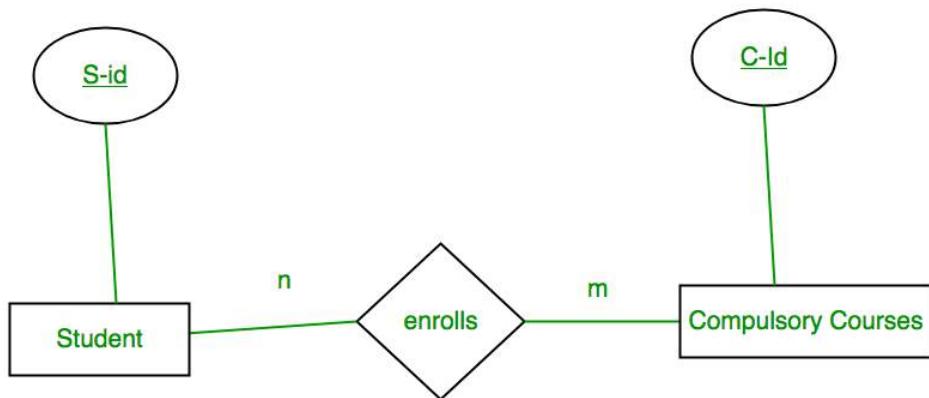
<u>S-Id</u>	Other Student Attribute	E-Id
-------------	-------------------------	------

Table 7

<u>E-Id</u>	Other Elective CourseAttribute
-------------	--------------------------------

Table 8

Case 4: Binary Relationship with m: n cardinality



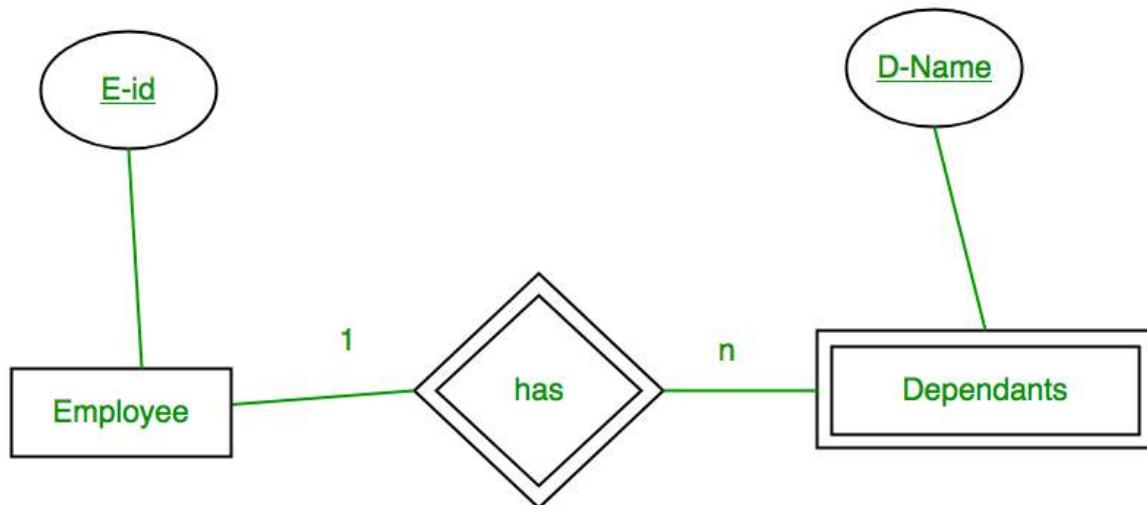
In this scenario, every student can enroll in more than 1 compulsory course and for a compulsory course there can be more than 1 student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Compulsory_Courses table corresponds 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		
<u>S- Id</u>	Other Attribute	Student	<u>S- Id</u>	<u>C- Id</u>	<u>C- Id</u>	Other CourseAttribute	Compulsory Courses
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



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
----------	-----	------------

<u>E- Id</u>	Other Employee Attribute	<u>E- Id</u>	<u>D- Name</u>	<u>D- Name</u>	<u>E- Id</u>	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.

<u>E-Id</u>	Other Employee Attribute
-------------	--------------------------

Table 11

<u>D-Name</u>	<u>E-Id</u>	Other DependentsAttribute
---------------	-------------	---------------------------

Table 12

Extended Entity-Relationship (EE-R) Model

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

Extended Entity-Relationship (EER) model is an enhancement of the original Entity-Relationship (ER) model, which includes additional concepts to represent more complex relationships and constraints in the database schema. The Extended ER model introduces new constructs such as specialization/generalization, aggregation, and attributes with multi-valued and derived values.

In addition to ER model concepts EE-R includes –

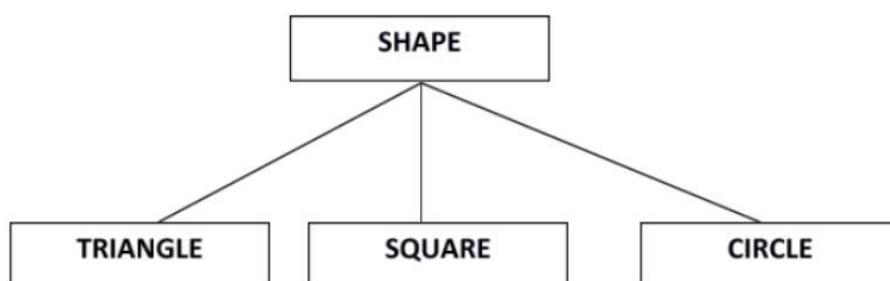
- Subclasses and Super classes.
- Specialization and Generalization.
- Category or union type.
- Aggregation.

These concepts are used to create EE-R diagrams.

Subclasses and Super class

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

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

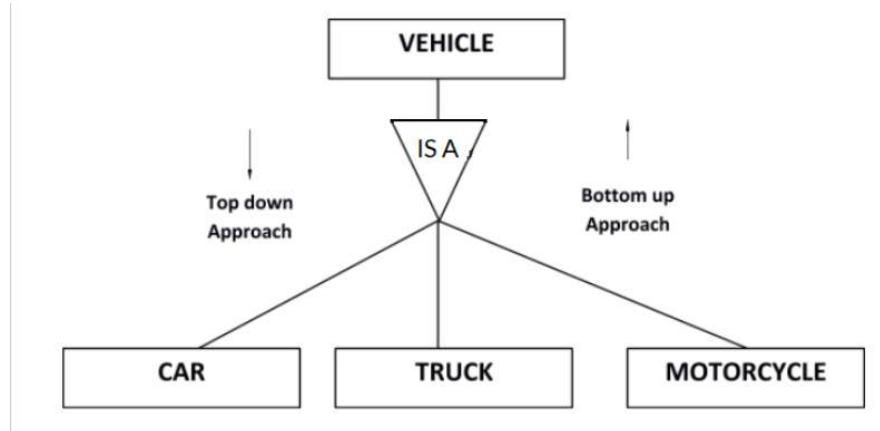


Super class shape has sub groups: Triangle, Square and Circle.

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

~~Specialization and Generalization~~

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



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

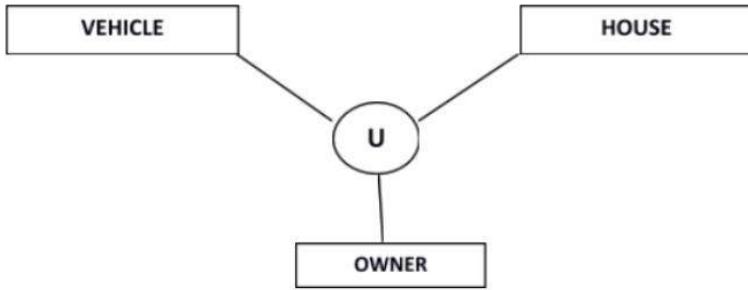
Specialization is a process of identifying subsets of an entity that share some different characteristic. It is a top down approach in which one entity is broken down into low level entity.

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

Example: In a university database, we can specialize the entity 'Person' into subtypes 'Student' and 'Faculty'. Each subtype inherits attributes from 'Person' but may have additional attributes specific to students or faculty members.

Category or Union

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

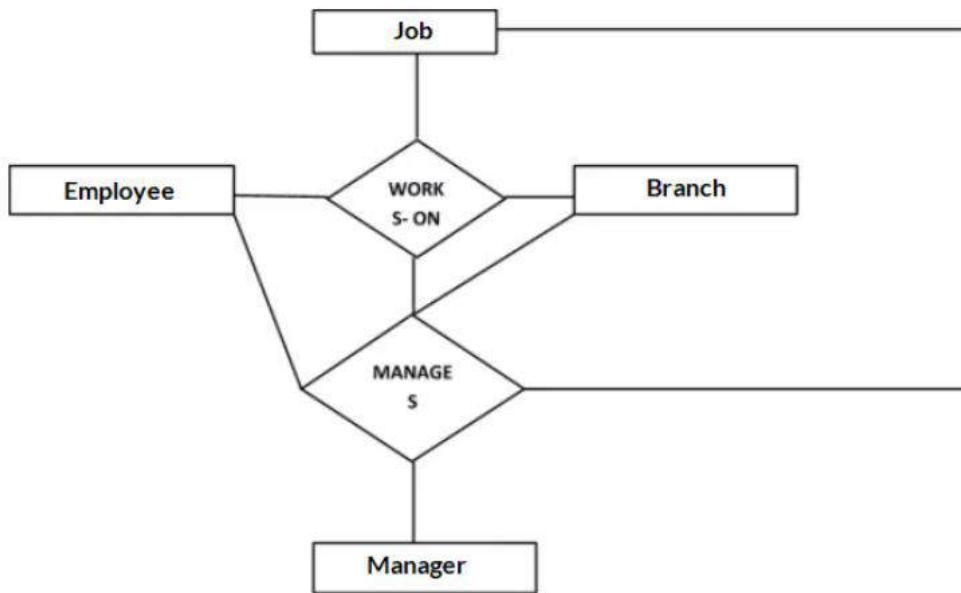


Owner is the subset of two super class: Vehicle and House.

Aggregation

Aggregation represents a relationship where an entity is composed of other entities. It allows modeling of complex relationships where one entity is a 'whole' and another entity is a 'part'.

Represents relationship between a whole object and its component.



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

Example: In a library database, we can aggregate the entities 'Book' and 'Author' into a higher-level entity 'Book_Author_Relationship', representing the relationship between books and their authors.

1. **Attributes with Multi-valued and Derived Values:**

- **Multi-valued attributes:** Attributes that can have multiple values for a single instance of an entity. They are represented by double ovals in the ER diagram.
- **Derived attributes:** Attributes whose values are derived from other attributes in the database. They are computed at runtime and are not stored in the database.

Example: In a social media network database, a 'User' entity may have a multi-valued attribute 'Interests' representing the user's hobbies or interests. Additionally, a 'User' entity may have a derived attribute 'Age' computed from the 'Date_of_Birth' attribute.

2. Identifying and Non-Identifying Relationships:

- In EER model, relationships can be either identifying or non-identifying.
- An identifying relationship means that the child entity's primary key includes the primary key of the parent entity.
- A non-identifying relationship means that the child entity's primary key does not include the primary key of the parent entity.

Example: In a hospital database, the relationship between 'Patient' and 'Visit' may be identifying, as a patient's visit is uniquely identified by the patient's ID along with the visit date.

The Extended Entity-Relationship model provides more expressive power than the original ER model, allowing for more accurate representation of complex data relationships and constraints in database designs. It's particularly useful when modeling real-world scenarios where entities have varying attributes and relationships.

Anomalies in Relational Model

Anomalies in the relational model refer to inconsistencies or errors that can arise when working with relational databases.

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

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 have to 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.

Advantages of 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.

Functional Dependency

Functional dependencies (FDs) are a fundamental concept in database theory, particularly in the context of database normalization. A functional dependency is a constraint between two sets of attributes in a relation from a database. It specifies the relationship between the values of one set of attributes and the values of another set of attributes.

Functional dependency (FD) is a set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A₁, A₂,..., A_n, then those two tuples must have to have same values for attributes B₁, B₂, ..., B_n.

Functional dependency is represented by an arrow sign (\rightarrow) that is, X \rightarrow Y, where X functionally determines Y. The left-hand side attributes determine the values of attributes on the right-hand side.

In simple terms, a functional dependency describes how the value of one attribute determines the value of another attribute within a given dataset.

Here's the formal definition:

Given a relation R, let X and Y be sets of attributes within R. X functionally determines Y (denoted as X \rightarrow Y) if, for every valid instance of X in R, there is precisely one instance of Y in R. In other words, the value(s) of Y are uniquely determined by the value(s) of X.

For example, let's consider a relation "Employee" with attributes {Emp_ID, Emp_Name, Dept_ID}. We can say that

" $\text{Emp_ID} \rightarrow \text{Emp_Name}$ " if each employee ID uniquely determines the corresponding employee name. This means that for every Emp_ID value, there is only one associated Emp_Name value.

Functional dependencies are crucial for database normalization, which aims to organize the attributes of a relation to minimize redundancy and dependency. By identifying and enforcing functional dependencies, we can ensure data integrity and eliminate anomalies such as update anomalies, insertion anomalies, and deletion anomalies.

There are various rules and properties associated with functional dependencies, including:

1. Reflexivity: If Y is a subset of X , then $X \rightarrow Y$.
2. Augmentation: If $X \rightarrow Y$, then adding attributes to either X or Y does not change the dependency. For example, if $\text{Emp_ID} \rightarrow \text{Emp_Name}$, then $\text{Emp_ID}, \text{Dept_ID} \rightarrow \text{Emp_Name}$.
3. Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. This means that if one attribute functionally determines another attribute, and the second attribute functionally determines a third, then the first attribute also functionally determines the third.

Functional dependencies play a crucial role in database design and optimization, especially in the process of normalization, where the database schema is organized to reduce redundancy and improve data integrity.

Trivial Functional Dependency

- **Trivial** – If a functional dependency (FD) $X \rightarrow Y$ holds, where Y is a subset of X , then it is called a trivial FD. Trivial FDs always hold.
- **Non-trivial** – If an FD $X \rightarrow Y$ holds, where Y is not a subset of X , then it is called a non-trivial FD.
- **Completely non-trivial** – If an FD $X \rightarrow Y$ holds, where X intersects $Y = \emptyset$, it is said to be a completely non-trivial FD.

Example to illustrate functional dependencies:

Let's say we have a relation called "Student_Course" with the following attributes:

- Student_ID (Student ID)
- Course_Code (Course Code)
- Semester
- Grade

In this relation, we might observe the following functional dependencies:

1. Student_ID -> Grade:

- This functional dependency indicates that the grade of a student in a particular course is uniquely determined by the student's ID.
- In other words, for each student, there is only one grade associated with that student in a given course.

2. Course_Code, Semester -> Grade:

- This functional dependency suggests that the grade of a student in a particular course during a specific semester is uniquely determined by the combination of course code and semester.
- In other words, for each combination of course code and semester, there is only one grade associated with that course and semester.

3. Student_ID, Course_Code -> Semester:

- This functional dependency implies that the semester in which a student takes a specific course is uniquely determined by both the student's ID and the course code.
- In other words, for each combination of student ID and course code, there is only one semester associated with that student and course.

Here's a sample instance of the "Student_Course" relation:

Student_ID	Course_Code	Semester	Grade
101	COMP101	Fall	A
102	MATH101	Fall	B
101	COMP101	Spring	B+
103	PHYS101	Fall	A-
102	MATH101	Spring	A

In this example:

- The functional dependency "Student_ID -> Grade" holds because each student has only one grade per course.

- The functional dependency "Course_Code, Semester \rightarrow Grade" holds because each course in a semester has only one grade.
 - The functional dependency "Student_ID, Course_Code \rightarrow Semester" holds because each student takes a specific course only once in a semester.
- Identifying and understanding these functional dependencies helps in normalization, query optimization, and maintaining data integrity in a database system.

Normalization in DBMS

Normalization is the process of organizing data in a database efficiently by reducing redundancy and dependency. The primary goal of normalization is to design a database schema that minimizes redundancy and ensures data integrity by eliminating or reducing the likelihood of data anomalies. There are several normal forms, each with its own criteria, that a database schema can be brought to during the normalization process.

Here's an overview of the normalization process, typically performed in several steps:

1. First Normal Form (1NF):

- Ensures that each attribute in a table contains only atomic values (values that cannot be divided further).
- Eliminates repeating groups by putting each group of related attributes into a separate table.
- Example: Splitting a table containing customer information and orders into separate tables for customers and orders.

2. Second Normal Form (2NF):

- Builds upon 1NF by ensuring that non-key attributes are fully functional dependent on the entire primary key.
- Eliminates partial dependencies, where non-key attributes depend on only part of the primary key.
- Example: Moving attributes that depend on part of the primary key into separate tables.

3. Third Normal Form (3NF):

- Builds upon 2NF by ensuring that non-key attributes are transitively dependent only on the primary key.
- Eliminates transitive dependencies, where non-key attributes depend on other non-key attributes.
- Example: Moving attributes that depend on other non-key attributes into separate tables.

4. Boyce-Codd Normal Form (BCNF):

- A stricter form of 3NF, where every determinant is a candidate key.
- Eliminates non-trivial functional dependencies where a non-prime attribute determines another non-prime attribute.
- Example: Ensuring that every non-trivial functional dependency has a candidate key as its determinant.

5. Fourth Normal Form (4NF):

- Addresses multi-valued dependencies where there are relationships between non-key attributes.

- Eliminates multi-valued dependencies by splitting tables into smaller, more atomic tables.
- Example: Splitting a table containing information about courses and instructors into separate tables for courses, instructors, and relationships between them.

6. Fifth Normal Form (5NF):

- Addresses join dependencies where a table is affected by a combination of keys from other tables.
- Ensures that there are no join dependencies by decomposing tables into smaller, more atomic tables.
- Example: Decomposing a table affected by join dependencies into smaller tables that are not affected by such dependencies.

Each normal form represents a level of normalization, with higher normal forms indicating a more refined database schema with reduced redundancy and dependency. The normalization process aims to ensure data integrity, minimize redundancy, and optimize database performance by organizing data efficiently.

Example of normalizing a database schema using the normalization process.

Consider a simple database for a library that stores information about books and authors. Initially, we have one table named "Books" with the following attributes:

- Book_ID (Primary Key)
- Title
- Author
- Genre
- Publisher
- Year_Published

Here's the initial table:

Book_ID	Title	Author	Genre	Publisher	Year_Published
1	The Great Gatsby	F. Scott Fitzgerald	Fiction	Scribner	1925
2	To Kill a Mockingbird	Harper Lee	Fiction	J. B. Lippincott	1960
3	Pride and Prejudice	Jane Austen	Fiction	Thomas Egerton	1813

Book_ID	Title	Author	Genre	Publisher	Year_Published
4	1984	George Orwell	Dystopian	Secker & Warburg	1949

Step 1: First Normal Form (1NF):

- Ensure that each attribute contains only atomic values.
- We can see that the "Author" attribute contains multiple values (for example, "F. Scott Fitzgerald"). We need to split it into separate attributes (such as "Author_First_Name" and "Author_Last_Name") or into a separate table if multiple authors are possible for one book.

Step 2: Second Normal Form (2NF):

- Eliminate partial dependencies.
- We need to ensure that each non-key attribute is fully functionally dependent on the entire primary key.
- In this case, the primary key is "Book_ID", and all other attributes depend on it fully. Therefore, the table is already in 2NF.

Step 3: Third Normal Form (3NF):

- Eliminate transitive dependencies.
- We need to ensure that non-key attributes are not dependent on other non-key attributes.
- The "Publisher" and "Year_Published" attributes are functionally dependent only on the "Book_ID" (the primary key), so the table is already in 3NF.

The resulting normalized tables might look like this:

Books Table:

Book_ID	Title	Publisher	Year_Published
1	The Great Gatsby	Scribner	1925
2	To Kill a Mockingbird	J. B. Lippincott	1960
3	Pride and Prejudice	Thomas Egerton	1813
4	1984	Secker & Warburg	1949

Authors Table:

Author_ID	First_Name	Last_Name
1	F. Scott	Fitzgerald
2	Harper	Lee
3	Jane	Austen
4	George	Orwell

Book_Authors Table:

Book_ID	Author_ID
1	1
2	2
3	3
4	4

In this normalized schema:

- The "Books" table contains information about each book, with the "Book_ID" as the primary key.
- The "Authors" table contains information about each author, with the "Author_ID" as the primary key.
- The "Book_Authors" table serves as a bridge table between books and authors, indicating which author(s) contributed to each book.

This normalization process eliminates redundancy, ensures data integrity, and facilitates efficient querying and maintenance of the database.

Boyce-Codd Normal Form (BCNF): In BCNF, every determinant (attribute or set of attributes on which another attribute is functionally dependent) must be a candidate key. Since our example already satisfies the conditions of BCNF, there is no further modification needed.

Fourth Normal Form (4NF): Fourth Normal Form addresses multi-valued dependencies. It ensures that there are no non-trivial multi-valued dependencies between attributes within a table.

In our example, we don't have any multi-valued dependencies as all attributes are fully functionally dependent on the primary key. Therefore, the table is already in 4NF.

Fifth Normal Form (5NF): Fifth Normal Form addresses join dependencies. It ensures that there are no join dependencies between two or more candidate keys.

In our example, we only have one candidate key, which is the primary key "Book_ID". Since there is only one key, there are no join dependencies. Therefore, the table is already in 5NF.

So, to summarize, the normalized schema we obtained earlier is already in BCNF, 4NF, and 5NF. We achieved this by decomposing the original table into separate tables to eliminate redundancy, ensure data integrity, and avoid anomalies.

Domain-Key Normal Form (DKNF)

The Domain-Key Normal Form (DKNF) is an extension of the Boyce-Codd Normal Form (BCNF), which itself is an extension of the Third Normal Form (3NF). DKNF addresses certain limitations of BCNF by ensuring that all constraints on a relation (table) are enforced by the domain (attribute) constraints and the key constraints. In simpler terms, DKNF aims to eliminate all redundancy and update anomalies by allowing only those relations that are completely free of any redundancy, dependency, or inconsistency.

To understand DKNF better, let's first review the concepts of BCNF and then delve into DKNF:

1. Boyce-Codd Normal Form (BCNF):

- BCNF is a stricter form of normalization compared to 3NF. It ensures that every determinant (attribute or set of attributes on which another attribute is functionally dependent) is a candidate key.
- A relation is in BCNF if, for every non-trivial functional dependency $X \rightarrow Y$, X is a superkey.

2. Domain-Key Normal Form (DKNF):

- DKNF takes BCNF a step further by ensuring that all constraints in a relation are enforced by the domain constraints and key constraints.
- A relation is in DKNF if and only if it is in BCNF and all constraints are domain constraints (constraints on individual attributes) and key constraints (constraints on combinations of attributes that form keys).
- In DKNF, no non-trivial functional dependency exists between attributes that are not a part of any key.

To achieve DKNF, it's essential to ensure that all attributes in a relation are functionally dependent on the entire key (candidate key) and that there are no non-trivial functional dependencies between attributes that are not part of any key.

Example to illustrate DKNF:

Consider a relation "Employee" with attributes {Employee_ID, Employee_Name, Department, Salary}, where {Employee_ID} is the key.

If we have functional dependencies:

- Employee_ID \rightarrow Employee_Name
- Employee_ID \rightarrow Department
- Employee_ID \rightarrow Salary

This relation is in BCNF because each determinant (Employee_ID) is a candidate key. However, it's not yet in DKNF because the attribute Salary might have domain constraints such as "Salary > 0" which are not solely based on the key.

To achieve DKNF, we must ensure that all constraints, including domain constraints, are enforced solely based on the key attributes. If the Salary attribute had domain constraints such as "Salary > 0", we would need to separate it into a separate relation with its own key, ensuring that all constraints are enforced solely by the key attributes.

DKNF ensures that a relation is completely free of any redundancy, dependency, or inconsistency by enforcing all constraints through domain constraints and key constraints.

Denormalization of database:

Database optimization is an essential step to improve website performance. Typically, developers normalize a relational database, meaning they restructure it to reduce data redundancy and enhance data integrity. However, sometimes normalizing a database isn't enough, so to improve database performance even further developers go the other way around and resort to database denormalization.

Denormalization in a database refers to the process of combining tables that were separated during the normalization process to reduce the number of joins needed to retrieve data, thus improving read performance at the cost of potential data redundancy and increased complexity in data maintenance.

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database. Note that *denormalization* does not mean ‘reversing normalization’ or ‘not to normalize’. It is an optimization technique that is applied after normalization.

Basically, The process of taking a normalized schema and making it non-normalized is called denormalization, and designers use it to tune the performance of systems to support time-critical operations.

In a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in a database.

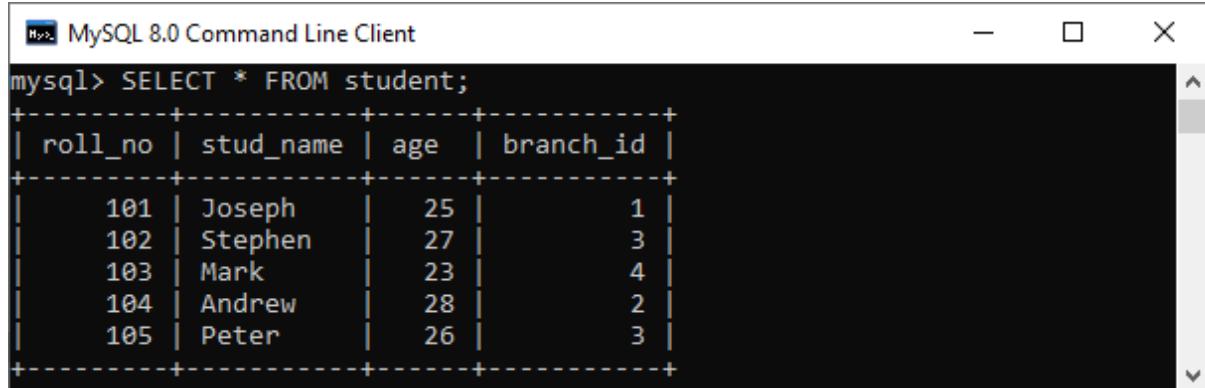
For example, in a normalized database, we might have a Courses table and a Teachers table. Each entry in Courses would store the teacherID for a Course but not the teacherName. When we need to retrieve a list of all Courses with the Teacher's name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback is that if tables are large, we may spend an unnecessarily long time doing joins on tables.

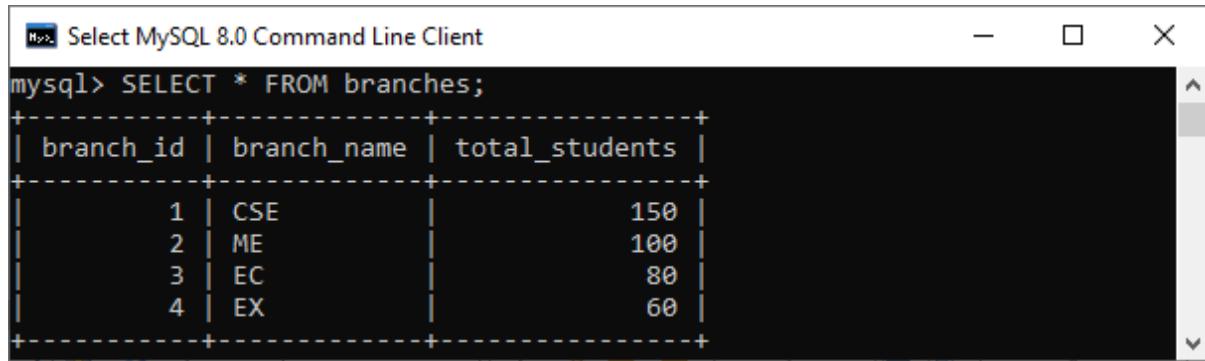
Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

For Example, We have two table students and branch after performing normalization. The student table has the attributes roll_no, stud-name, age, and branch_id.



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM student;
+-----+-----+-----+-----+
| roll_no | stud_name | age | branch_id |
+-----+-----+-----+-----+
|    101 | Joseph    |   25 |        1 |
|    102 | Stephen   |   27 |        3 |
|    103 | Mark      |   23 |        4 |
|    104 | Andrew    |   28 |        2 |
|    105 | Peter     |   26 |        3 |
+-----+-----+-----+-----+
```

Additionally, the branch table is related to the student table with branch_id as the student table's foreign key.



```
Select MySQL 8.0 Command Line Client
mysql> SELECT * FROM branches;
+-----+-----+-----+
| branch_id | branch_name | total_students |
+-----+-----+-----+
|      1 | CSE          |       150 |
|      2 | ME           |       100 |
|      3 | EC           |        80 |
|      4 | EX           |        60 |
+-----+-----+-----+
```

A [JOIN operation](#) between these two tables is needed when we need to retrieve all student names as well as the branch name. Suppose we want to change the student name only, then it is great if the table is small. The issue here is that if the tables are big, joins on tables can take an excessively long time.

In this case, we'll update the database with denormalization, redundancy, and extra effort to maximize the efficiency benefits of fewer joins. Therefore, we can add the branch name's data from the Branch table to the student table and optimizing the database.

How is denormalization different from normalization?

The denormalization is different from normalization in the following manner:

- Denormalization is a technique used to merge data from multiple tables into a single table that can be queried quickly. Normalization, on the other hand, is used to delete redundant data from a database and replace it with non-redundant and reliable data.

- Denormalization is used when joins are costly, and queries are run regularly on the tables. Normalization, on the other hand, is typically used when a large number of insert/update/delete operations are performed, and joins between those tables are not expensive.

Pros of Denormalization:

1. Retrieving data is faster since we do fewer joins
2. Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.

Cons of Denormalization:

1. Updates and inserts are more expensive.
2. Denormalization can make *update* and *insert* code harder to write.
3. Data may be inconsistent.
4. Data redundancy necessitates more storage.

In a system that demands scalability, like that of any major tech company, we almost always use elements of both normalized and denormalized databases.

Advantages of Denormalization:

Improved Query Performance: Denormalization can improve query performance by reducing the number of joins required to retrieve data.

Reduced Complexity: By combining related data into fewer tables, denormalization can simplify the database schema and make it easier to manage.

Easier Maintenance and Updates: Denormalization can make it easier to update and maintain the database by reducing the number of tables.

Improved Read Performance: Denormalization can improve read performance by making it easier to access data.

Better Scalability: Denormalization can improve the scalability of a database system by reducing the number of tables and improving the overall performance.

Disadvantages of Denormalization:

Reduced Data Integrity: By adding redundant data, denormalization can reduce data integrity and increase the risk of inconsistencies.

Increased Complexity: While denormalization can simplify the database schema in some cases, it can also increase complexity by introducing redundant data.

Increased Storage Requirements: By adding redundant data, denormalization can increase storage requirements and increase the cost of maintaining the database.

Increased Update and Maintenance Complexity: Denormalization can increase the complexity of updating and maintaining the database by introducing redundant data.

Limited Flexibility: Denormalization can reduce the flexibility of a database system by introducing redundant data and making it harder to modify the schema.

Example: Suppose after normalization we have two tables first, Student table and second, Branch table. The student has the attributes as *Roll_no* , *Student-name* , *Age* , and *Branch_id* .

Student table

Roll_no	Student_name	Age	Branch_id
1	Andrew	18	10
2	Angel	19	10
3	Priya	20	10
4	Analisa	21	11
5	Anna	21	12

The branch table is related to the Student table with *Branch_id* as the foreign key in the Student table.

Branch table

Branch_id	Branch_name	HOD
10	CSE	Mr.abc
11	EC	Dr.xyz
12	EX	Dr.pqr

If we want the name of students along with the name of the branch name then we need to perform a join operation. The problem here is that if the table is large we need a lot of time to perform the join operations. So, we can add the data of *Branch_name* from Branch table to the Student table and this will help in reducing the time that would have been used in join operation and thus optimize the database.

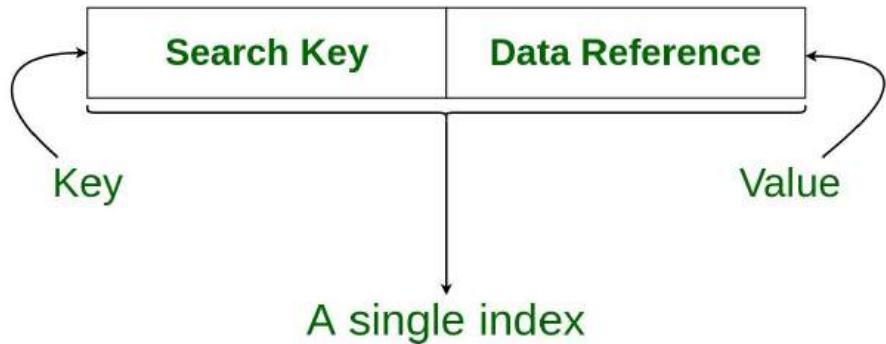
Indexing in Databases

- Indexing is a data structure technique that allows the DBMS to quickly locate and access the data in a database. It works much like an index in a book, where an index is created for each piece of data, enabling the DBMS to find data without scanning the entire table.
- Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
-

Indexes are created using a few database columns.

- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
Note: The data may or may not be stored in sorted order.
- 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.

Structure of an Index in Database



Types of file organization mechanism which are followed by the indexing methods to store the data:

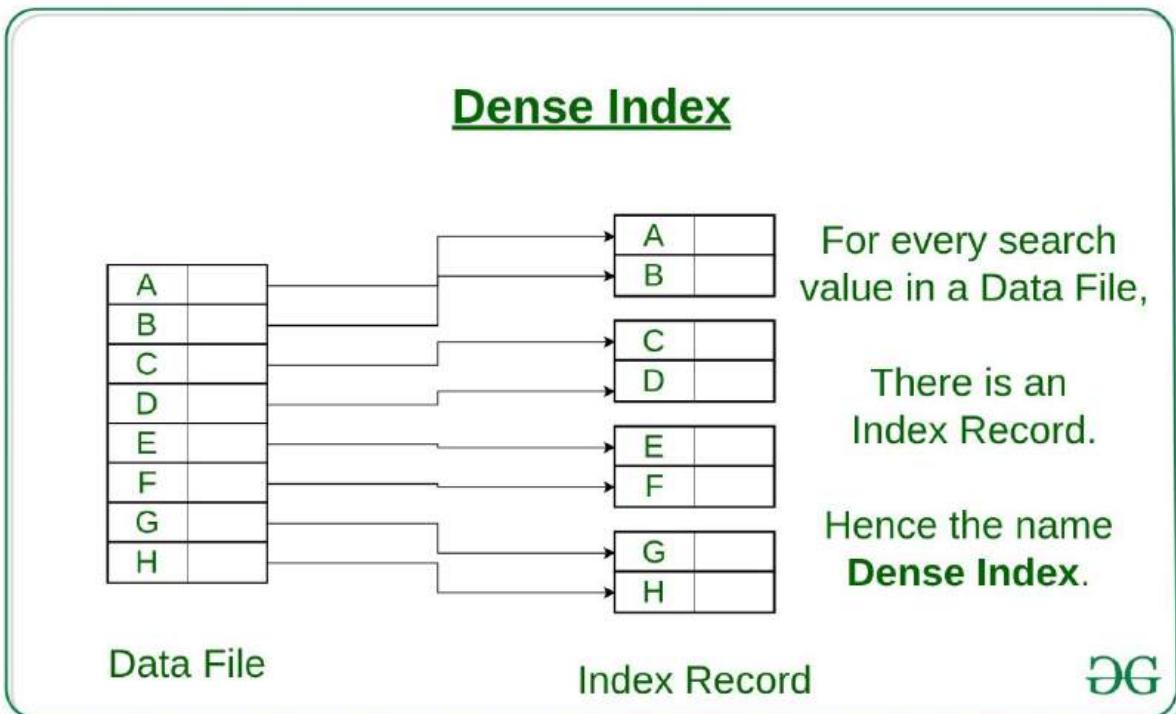
1. Sequential File Organization or Ordered Index File

2. Hash File organization

1. 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 organization might store the data in a dense or sparse format:

(i) Dense Index:

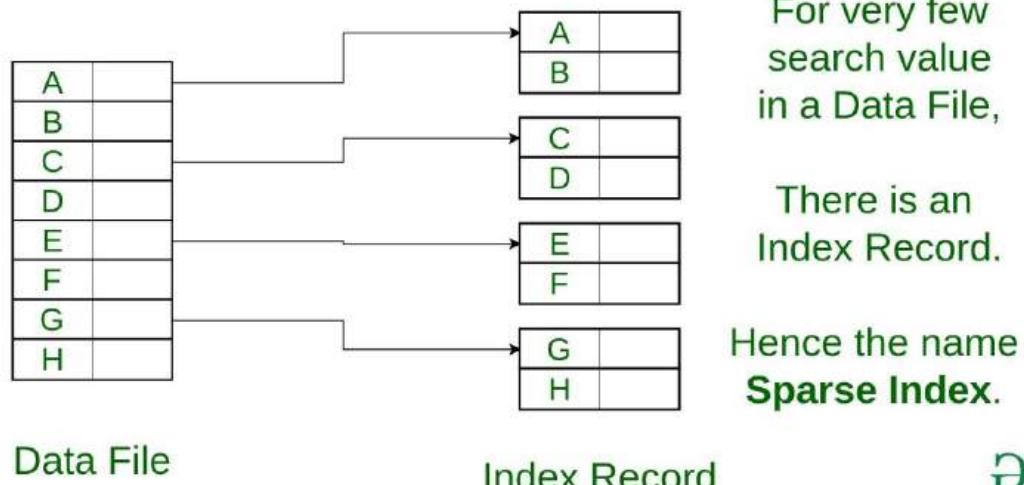
- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.



(ii) 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)

Sparse Index



DB

Benefits of Indexing

1. Improved Query Performance:
 - Indexes significantly speed up the search process for queries, especially SELECT queries that filter rows based on a WHERE clause.
2. Efficient Sorting:
 - Indexes help in faster sorting of records in queries with ORDER BY clauses.
3. Quick Access to Data:
 - Indexes reduce the amount of data that needs to be scanned to find specific data, thus improving data access speed.

Drawbacks of Indexing

1. Additional Storage:
 - Indexes require additional disk space as they store copies of indexed columns along with pointers to the actual data rows.
2. Maintenance Overhead:
 - Every INSERT, DELETE, and UPDATE operation can become slower due to the need to update the indexes.
3. Complexity:
 - Designing efficient indexes requires careful consideration and understanding of query patterns.

Best Practices

1. Analyze Query Patterns:
 - Create indexes based on the columns frequently used in WHERE, JOIN, ORDER BY, and GROUP BY clauses.
2. Limit the Number of Indexes:
 - While indexes improve read performance, they can degrade write performance. Balance is key.
3. Use Composite Indexes Wisely:
 - Create composite indexes for queries that filter on multiple columns.
4. Regularly Monitor and Maintain Indexes:
 - Regularly monitor the performance impact of indexes and remove those that are not being used.
5. Consider Index Fill Factor:
 - Adjust the fill factor of indexes to reduce page splits and improve performance.

Indexing Attributes

The indexing has various attributes:

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

Types of file organization mechanism

Types of file organization mechanism which are followed by the indexing methods to store the data:

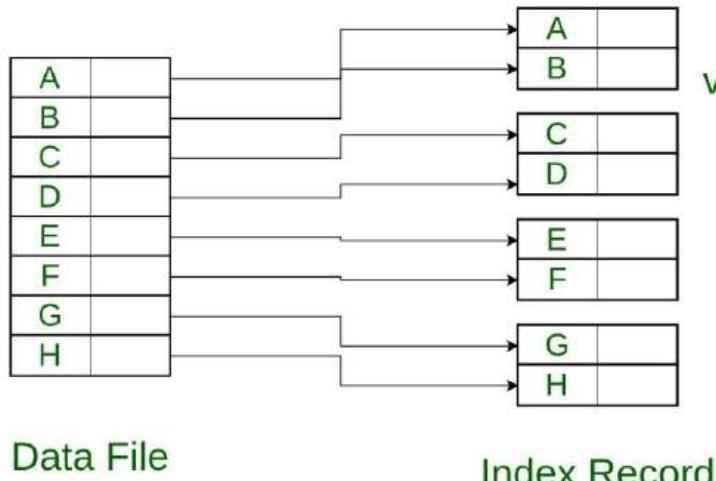
1. **Sequential File Organization or Ordered Index File**
2. **Hash File organization**

1. 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 organization might store the data in a dense or sparse format:

(i) Dense Index:

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

Dense Index



For every search value in a Data File,

There is an Index Record.

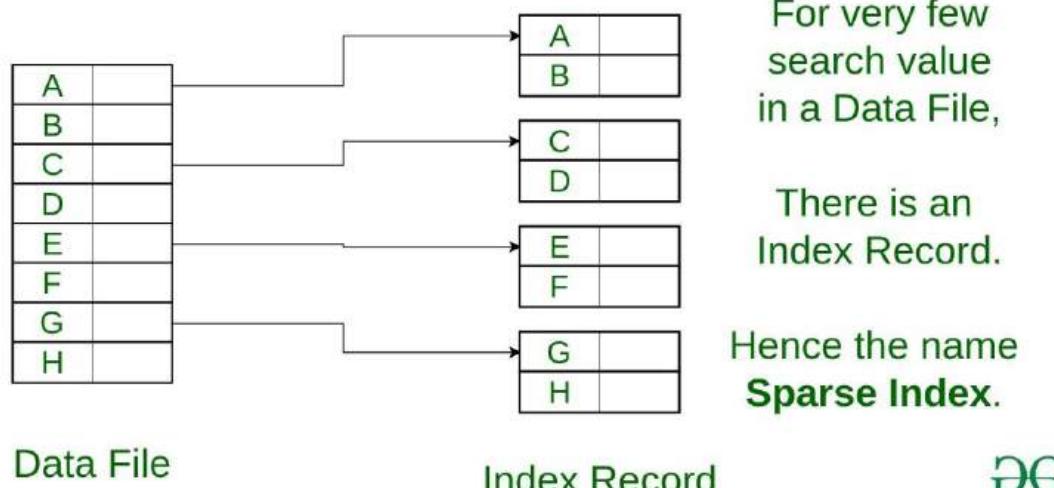
Hence the name **Dense Index**.

DG

(ii) 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)

Sparse Index



For very few
search value
in a Data File,

There is an
Index Record.

Hence the name
Sparse Index.

DG

2. 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 is determined by a function called a hash function.

There are primarily three methods of indexing:

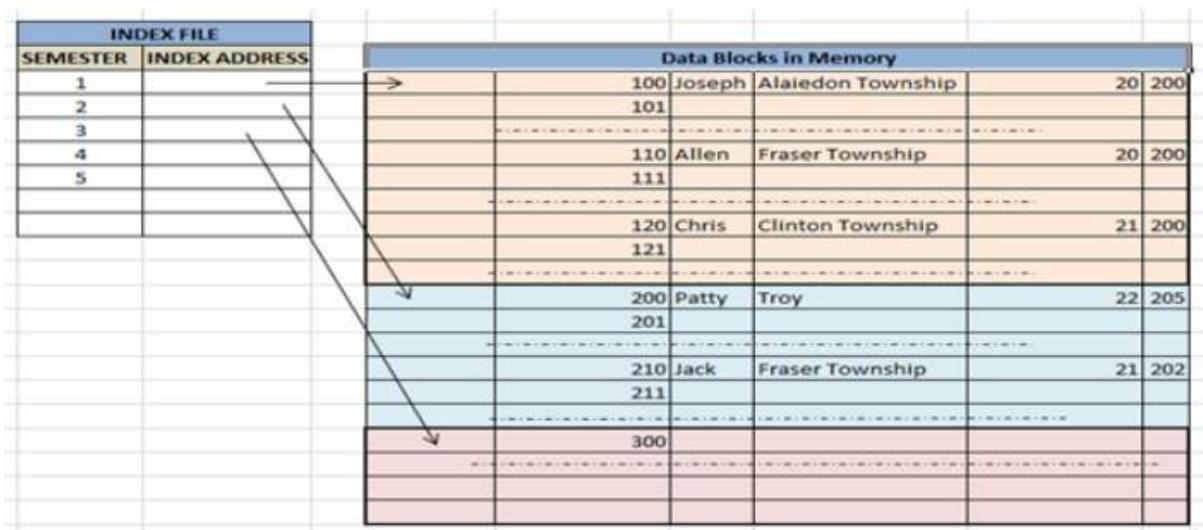
- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. Clustered Indexing

When more than two records are stored in the same file these types of storing known as cluster indexing. By using the cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored at one place and it also gives the frequent joining of more than two tables (records).

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 index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc. are grouped.



Clustered index sorted according to first name (Search key)

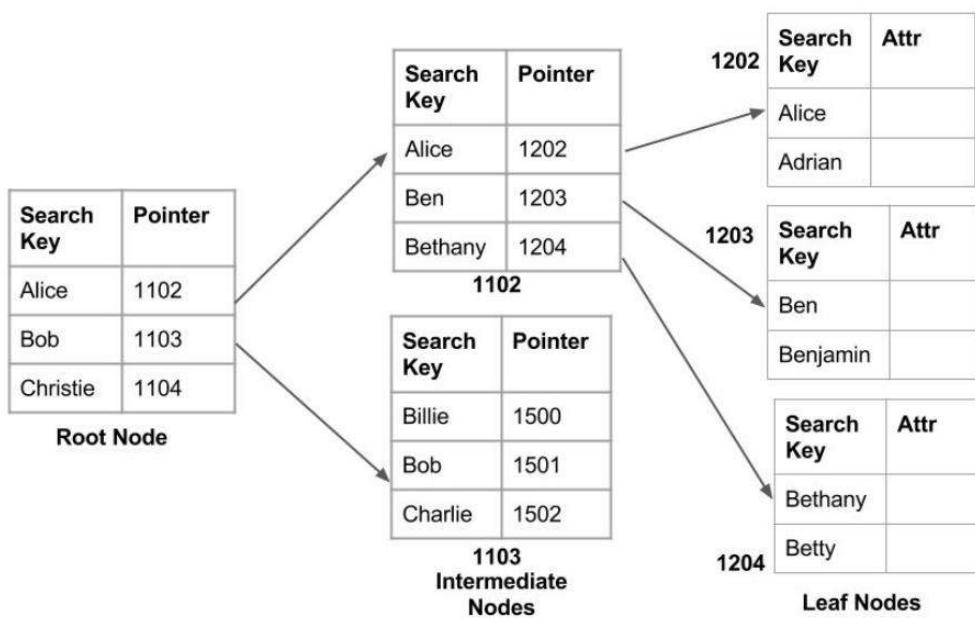
Primary Indexing:

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.

2. 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 actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. 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 actually 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.

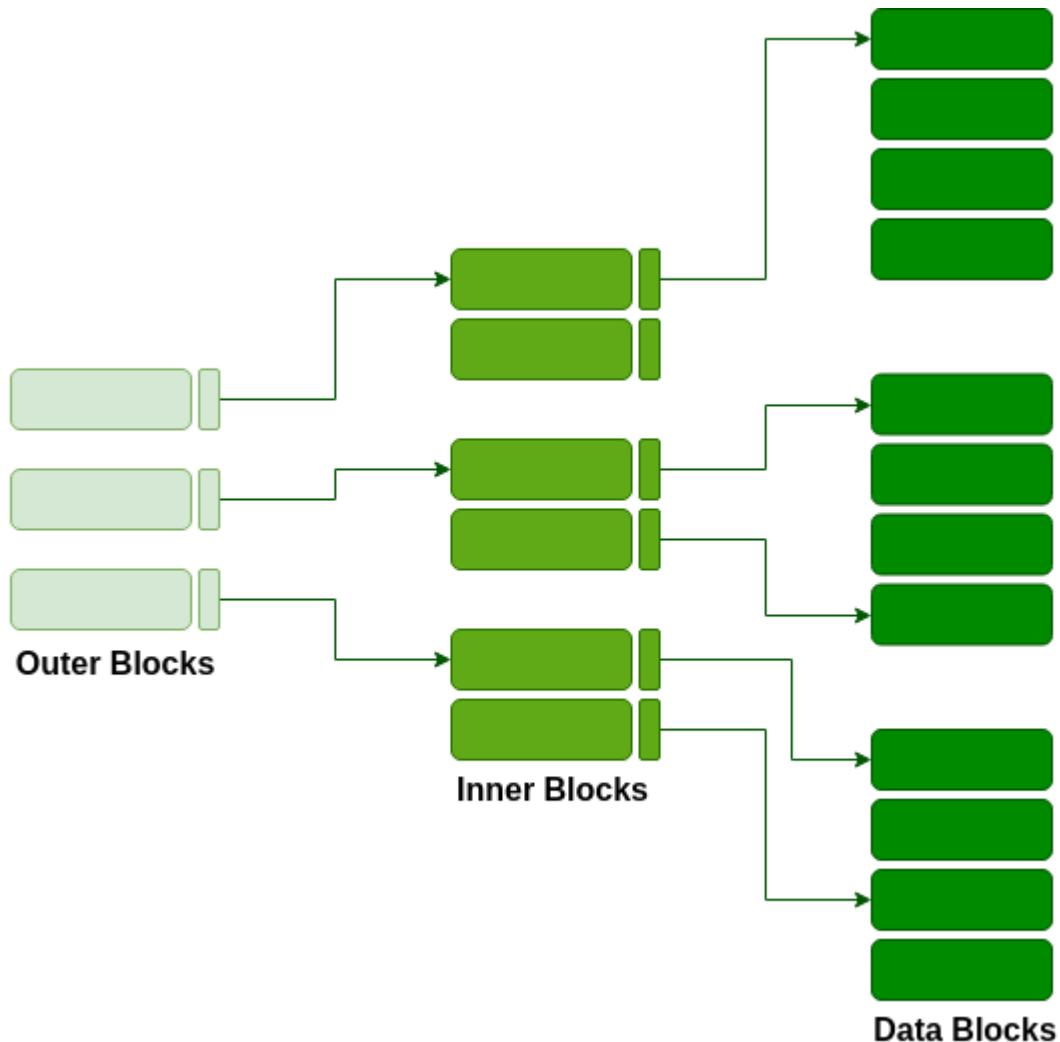
It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Non clustered index

3. 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.



Single Level Indexing

It is somewhat like the index (or the table of contents) found in a book. Index of a book contains topic names along with the page number similarly the index table of the database contains keys and their corresponding block address.

Single Level Indexing is further divided into three categories:

1. Primary Indexing: The indexing or the index table created using Primary keys is known as Primary Indexing. It is defined on ordered data. As the index is comprised of primary keys, they are unique, not null, and possess one to one relationship with the data blocks.

Characteristics of Primary Indexing:

- Search Keys are unique.

- Search Keys are in sorted order.
 - Search Keys cannot be null as it points to a block of data.
 - Fast and Efficient Searching.
2. Secondary Indexing: It is a two-level indexing technique used to reduce the mapping size of the primary index. The secondary index points to a certain location where the data is to be found but the actual data is not sorted like in the primary indexing. Secondary Indexing is also known as non-clustered Indexing.
- Characteristics of Secondary Indexing:
- Search Keys are Candidate Keys.
 - Search Keys are sorted but actual data may or may not be sorted.
 - Requires more time than primary indexing.
 - Search Keys cannot be null.
 - Faster than clustered indexing but slower than primary indexing.
3. Cluster Indexing: Clustered Indexing is used when there are multiple related records found at one place. It is defined on ordered data. The important thing to note here is that the index table of clustered indexing is created using non-key values which may or may not be unique. To achieve faster retrieval, we group columns having similar characteristics. The indexes are created using these groups and this process is known as Clustering Index.

- Characteristics of Clustered Indexing:
- Search Keys are non-key values.
 - Search Keys are sorted.
 - Search Keys cannot be null.
 - Search Keys may or may not be unique.
 - Requires extra work to create indexing.

Ordered Indexing:

Ordered indexing is the traditional way of storing that gives fast retrieval. The indices are stored in a sorted manner hence it is also known as ordered indices.

Ordered Indexing is further divided into two categories:

1. Dense Indexing: In dense indexing, the index table contains records for every search key value of the database. This makes searching faster but requires a lot more space. It is like primary indexing but contains a record for every search key.
2. Sparse Indexing: Sparse indexing consumes lesser space than dense indexing, but it is a bit slower as well. We do not include a search key for every record despite that we store a Search key that points to a block. The pointed block further contains a group of data. Sometimes we have to perform double searching this makes sparse indexing a bit slower.

Multi-Level Indexing

Since the index table is stored in the main memory, single-level indexing for a huge amount of data requires a lot of memory space. Hence, multilevel indexing was introduced in which we divide the main data block into smaller blocks. This makes the outer block of the index table small enough to be stored in the main memory.

Advantages of Indexing

- Indexing helps in faster query results or quick data retrieval.
- Indexing helps in faster sorting and grouping of records

- Some Indexing uses sorted and unique keys which helps to retrieve sorted queries even faster.
- Index tables are smaller in size so require lesser memory.
- As Index tables are smaller in size, they are stored in the main memory.
- Since CPU speed and secondary memory speed have a large difference, the CPU uses this main memory index table to bridge the gap of speeds.
- Indexing helps in better CPU utilization and better performance.

Implementation of Indexing using MYSQL

```
mysql> set profiling=1;  
  
mysql> show profiles;  
  
mysql> show profiles;  
  
mysql> show profile for query 1;  
  
mysql> show index from employees;  
  
mysql> explain select * from employees;  
  
mysql> explain select * from employees where employeeid=2;
```

Consider the Employees table with the following data:

EmployeeID	Name	Position	Salary
1	Alice	Manager	80000
2	Bob	Analyst	60000
3	Carol	Manager	85000
4	Dave	Developer	75000
5	Eve	Analyst	62000

- **Primary Index:** Ensures each EmployeeID is unique and sorted.

```
CREATE TABLE Employees(EmployeeID INT PRIMARY KEY,  
    Name VARCHAR(50),  
    Position VARCHAR(10),  
    salary int  
) ;
```

- **Secondary Index:** Fast retrieval by Position

```
CREATE INDEX idx_name ON Employees(Position);  
  
SELECT * FROM Employees WHERE Position = 'Manager';
```

- **Clustered Index:** Table rows sorted by Salary for quick range queries.

```
CREATE CLUSTERED INDEX idx_sal ON Employees(Salary);
```

- **Non-clustered Index:** Quick lookup by Name without altering row order.

```
CREATE NONCLUSTERED INDEX idx_Name ON Employees(Name);
```

- **Unique Index**

```
CREATE UNIQUE INDEX idx_unique_name ON Employees(Name);
```

- **Composite Index:** Fast retrieval for queries using both Name and Position

```
CREATE INDEX idx_name_position ON Employees(Name, Position);  
SELECT * FROM Employees WHERE Name = 'Alice' AND Position =  
'Manager';
```

- **Bitmap Index**

```
CREATE BITMAP INDEX idx_position_bitmap ON Employees(Position);
```

- **Function-based Index**

```
CREATE INDEX idx_upper_name ON Employees(UPPER(Name));
```

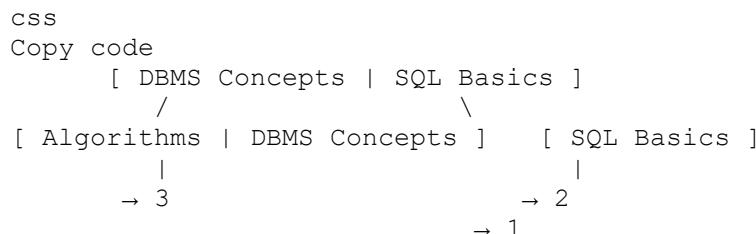
- **Full-Text Index** is used for performing text searches in large text columns.

```
CREATE FULLTEXT INDEX idx_fulltext_name ON Employees(Name);
```

Dynamic Multilevel Indexing

Dynamic Multilevel Indexing uses dynamic data structures like B-trees or B+ trees to manage the index entries. These structures automatically adjust as data is inserted, deleted, or updated, ensuring balanced trees and efficient access.

B+ Tree Example: For the Books table, a B+ tree index on Title might look like this:



In a B+ tree:

- Internal nodes direct the search.
- Leaf nodes contain the actual pointers to data.
- The tree is balanced, maintaining optimal search performance.

TRANSACTIONS

Transaction Concept: A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

EX: transaction to transfer \$50 from account A to account B:

```
read(A)
A := A - 50
write(A)
read(B)
B := B + 50
write(B)
```

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Atomicity requirement — If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware
- ✓ System should ensure that updates of a partially executed transaction are not reflected in database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction. In general, consistency requirements include

- ▶ Explicitly specified integrity constraints such as primary keys and foreign keys
- ▶ Implicit integrity constraints
- EX: sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
 - A transaction must see a consistent database.
 - During transaction execution the database may be temporarily inconsistent.
 - When the transaction completes successfully the database must be consistent
 - Erroneous transaction logic can lead to inconsistency

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. read(A)	
2. $A := A - 50$	
3. write(A)	read(A), read(B), print(A+B)
4. read(B)	
5. $B := B + 50$	
6. write(B)	

Isolation can be ensured trivially by running transactions **serially**

-that is, one after the other.

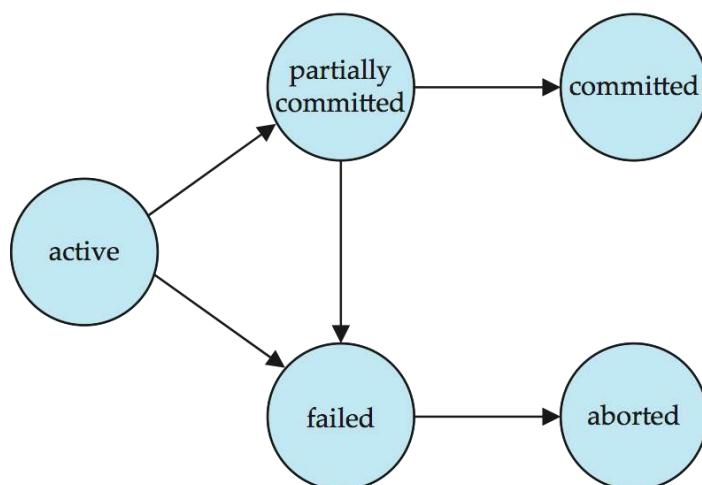
However, executing multiple transactions concurrently has significant benefits.

ACID Properties: A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

1. **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
2. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
 - a. That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
4. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State:

1. **Active** – the initial state; the transaction stays in this state while it is executing
2. **Partially committed** – after the final statement has been executed.
3. **Failed** -- after the discovery that normal execution can no longer proceed.
4. **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
 - a. restart the transaction
 - i. can be done only if no internal logical error
 - b. kill the transaction
5. **Committed** – after successful completion.



Implementation of Atomicity and Durability: The **recovery-management** component of a database system implements the support for atomicity and durability.

EX: the **shadow-database** scheme:

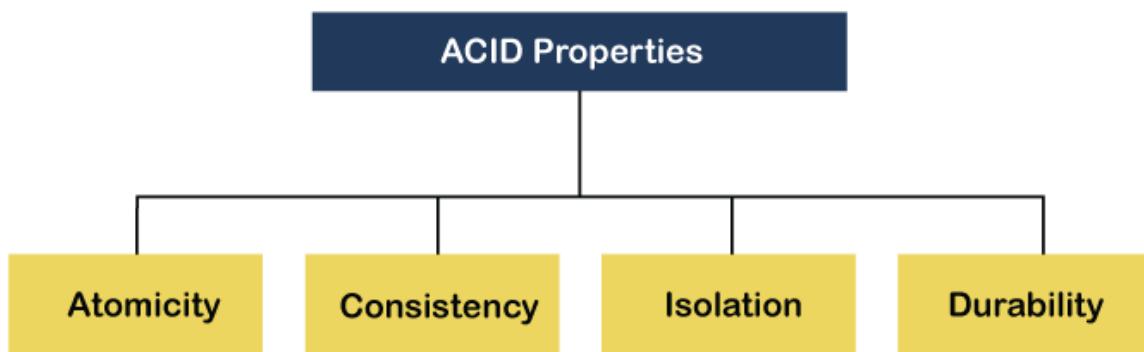
-all updates are made on a *shadow copy* of the database

ACID Properties in DBMS

DBMS is the management of data that should remain integrated when any changes are done in it. It is because if the integrity of the data is affected, whole data will get disturbed and corrupted. Therefore, to maintain the integrity of the data, there are four properties described in the database management system, which are known as the **ACID** properties. The ACID properties are meant for the transaction that goes through a different group of tasks, and there we come to see the role of the ACID properties.

ACID Properties

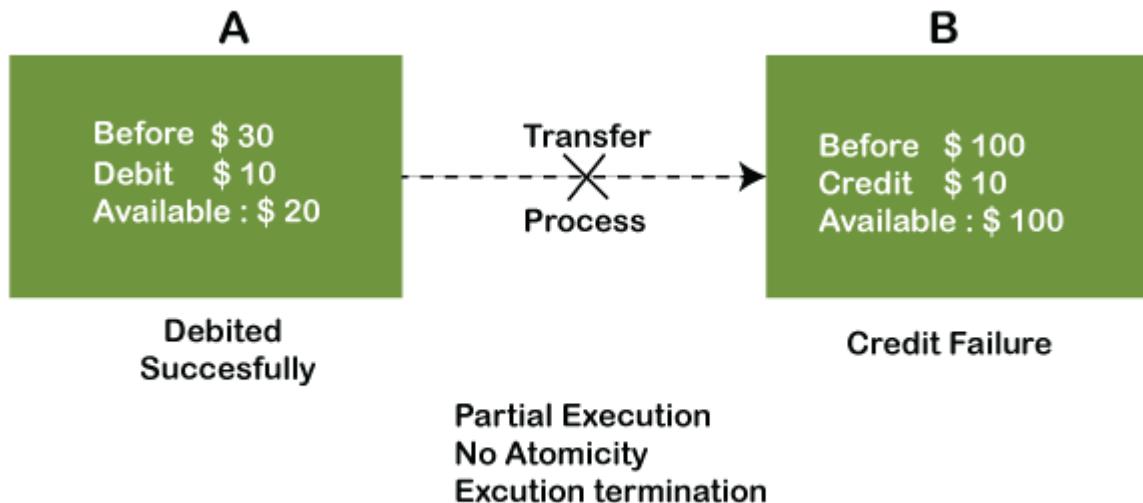
The expansion of the term ACID defines for:



1) Atomicity

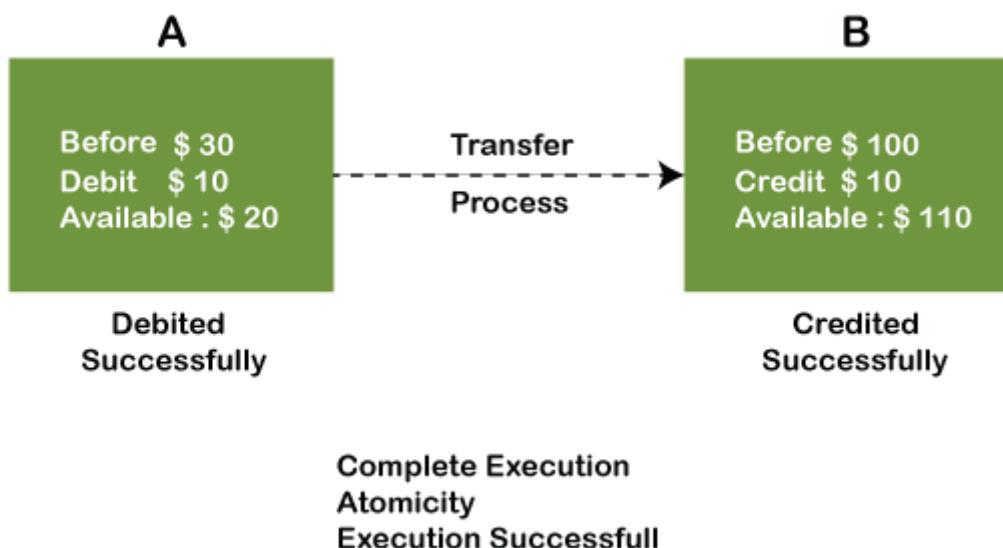
The term atomicity defines that the data remains atomic. It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all. It further means that the operation should not break in between or execute partially. In the case of executing operations on the transaction, the operation should be completely executed and not partially.

Example: If Remo has account A having \$30 in his account from which he wishes to send \$10 to Sheero's account, which is B. In account B, a sum of \$ 100 is already present. When \$10 will be transferred to account B, the sum will become \$110. Now, there will be two operations that will take place. One is the amount of \$10 that Remo wants to transfer will be debited from his account A, and the same amount will get credited to account B, i.e., into Sheero's account. Now, what happens - the first operation of debit executes successfully, but the credit operation, however, fails. Thus, in Remo's account A, the value becomes \$20, and to that of Sheero's account, it remains \$100 as it was previously present.



In the above diagram, it can be seen that after crediting \$10, the amount is still \$100 in account B. So, it is not an atomic transaction.

The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic.

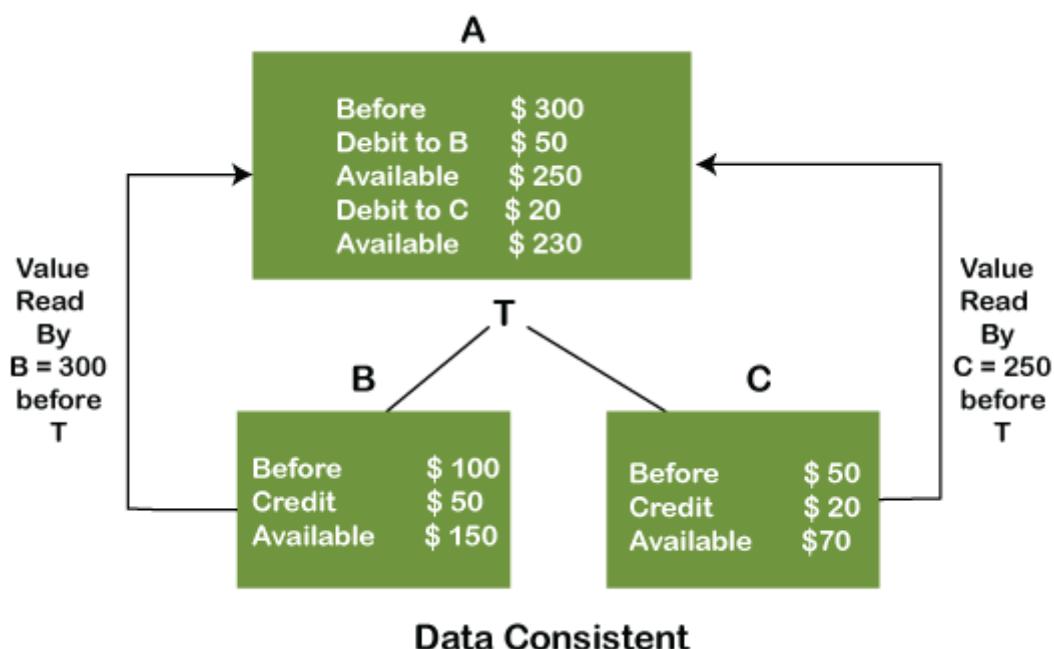


Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

2) Consistency

The word **consistency** means that the value should remain preserved always. In DBMS, the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always. In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction. The data should always be correct.

Example:



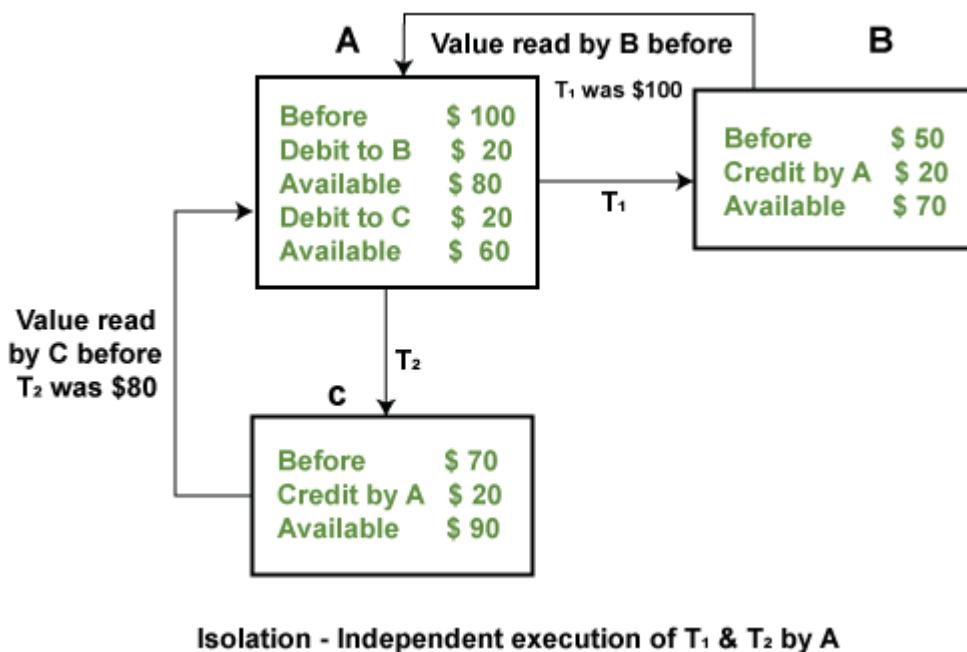
In the above figure, there are three accounts, A, B, and C, where A is making a transaction T one by one to both B & C. There are two operations that take place, i.e., Debit and Credit. Account A firstly debits \$50 to account B, and the amount in account A is read \$300 by B before the transaction. After the successful transaction T, the available amount in B becomes \$150. Now, A debits \$20 to account C, and that time, the value read by C is \$250 (that is correct as a debit of \$50 has been successfully done to B). The debit and credit operation from account A to C has been done successfully. We can see that the transaction is done successfully, and the value is also read correctly. Thus, the data is consistent. In case the value read by B and C is \$300, which means that data is inconsistent because when the debit operation executes, it will not be consistent.

3) Isolation

The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the

operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.

Example: If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected. The value should remain persistent. As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.



4) Durability

Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives. However, if gets lost, it becomes the responsibility of the recovery manager for ensuring the durability of the database. For committing the values, the COMMIT command must be used every time we make changes.

Therefore, the ACID property of DBMS plays a vital role in maintaining the consistency and availability of data in the database.

To demonstrate ACID properties in a bank transfer scenario involving two main tables, `Accounts` and `Transactions`, let's assume the following structures for these tables:

```
CREATE TABLE Accounts (
    account_id INT PRIMARY KEY,
    account_holder VARCHAR(100),
    balance DECIMAL(10, 2)
);
```

```
CREATE TABLE Transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    from_account_id INT,
    to_account_id INT,
    amount DECIMAL(10, 2),
    transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (from_account_id) REFERENCES Accounts(account_id),
    FOREIGN KEY (to_account_id) REFERENCES Accounts(account_id)
);
```

Atomicity and consistency:

```
START TRANSACTION;
-- Subtract $100 from Alice's account
UPDATE Accounts
SET balance = balance - 100
WHERE account_holder = 'Alice';
-- Add $100 to Bob's account
UPDATE Accounts
SET balance = balance + 100
WHERE account_holder = 'Bob';
-- Check for potential errors and commit if all is well
COMMIT;
```

If any error occurs during these operations, we can roll back to ensure that neither account is updated partially:

```
START TRANSACTION;

BEGIN TRY

    -- Subtract $100 from Alice's account

    UPDATE Accounts

    SET balance = balance - 100

    WHERE account_holder = 'Alice';

    -- Add $100 to Bob's account

    UPDATE Accounts

    SET balance = balance + 100

    WHERE account_holder = 'Bob';

    -- Log the transaction

    INSERT INTO Transactions (from_account_id, to_account_id, amount)
    VALUES ((SELECT account_id FROM Accounts WHERE account_holder = 'Alice'),
            (SELECT account_id FROM Accounts WHERE account_holder = 'Bob'),
            100);

    -- Commit the transaction

    COMMIT;

END TRY

BEGIN CATCH

    -- Rollback if there is any error

    ROLLBACK;

END CATCH;

Isolation

-- Set the isolation level to Serializable to ensure complete isolation

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

-- Subtract $100 from Alice's account

UPDATE Accounts

SET balance = balance - 100

WHERE account_holder = 'Alice';
```

```
-- Add $100 to Bob's account

UPDATE Accounts

SET balance = balance + 100

WHERE account_holder = 'Bob';

-- Log the transaction

INSERT INTO Transactions (from_account_id, to_account_id, amount)

VALUES ((SELECT account_id FROM Accounts WHERE account_holder = 'Alice'),

        (SELECT account_id FROM Accounts WHERE account_holder = 'Bob'),

        100);

COMMIT;
```

Durability

Durability is ensured by the database system itself once the transaction is committed. The committed transaction data is written to disk, ensuring it persists even after a system failure.

Example of Full Transaction with ACID Compliance

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

BEGIN TRY

    -- Subtract $100 from Alice's account

    UPDATE Accounts

    SET balance = balance - 100

    WHERE account_holder = 'Alice';

    -- Add $100 to Bob's account

    UPDATE Accounts

    SET balance = balance + 100

    WHERE account_holder = 'Bob';

    -- Log the transaction

    INSERT INTO Transactions (from_account_id, to_account_id, amount)

    VALUES ((SELECT account_id FROM Accounts WHERE account_holder = 'Alice'),

            (SELECT account_id FROM Accounts WHERE account_holder = 'Bob'),

            100);
```

```
-- Commit the transaction  
COMMIT;  
END TRY  
  
BEGIN CATCH  
    -- Rollback if there is any error  
    ROLLBACK;  
END CATCH;
```

This ensures that the \$100 transfer from Alice's account to Bob's account adheres to the ACID properties, maintaining the integrity and reliability of the transaction within the bank's database system.

Concurrent transaction processing & Concurrent control problems

Concurrency control is an essential aspect of database management systems (DBMS) that ensures transactions can execute concurrently without interfering with each other. However, concurrency control can be challenging to implement, and without it, several problems can arise, affecting the consistency of the database. In this article, we will discuss some of the concurrency problems that can occur in DBMS transactions and explore solutions to prevent them.

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in a database environment.

The five concurrency problems that can occur in the database are:

- Temporary Update Problem
- Incorrect Summary Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem

These are explained as following below.

Temporary Update Problem:

Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Example:

T1	T2
<pre>read_item(X) X = X - N write_item(X)</pre>	<pre>read_item(X) X = X + M write_item(X)</pre>

In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

Incorrect Summary Problem:

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:

T1	T2
<pre>read_item(X) X = X - N write_item(X) read_item(Y) Y = Y + N write_item(Y)</pre>	<pre>sum = 0 read_item(A) sum = sum + A read_item(X) sum = sum + X read_item(Y) sum = sum + Y</pre>

In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

Lost Update Problem:

In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

T1	T2
<pre>read_item(X) X = X + N</pre>	<pre>X = X + 10 write_item(X)</pre>

In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X (not shown in the image above). Therefore, the update done by transaction 2 will be lost. Basically, the write commit done by the last transaction will overwrite all previous write commits.

Unrepeatable Read Problem:

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

Example:

T1	T2
Read(X)	
Write(X)	Read(X)
	Read(X)

In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

Phantom Read Problem:

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

Example:

T1	T2
Read(X)	
Delete(X)	Read(X)
	Read(X)

In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Solution :

To prevent concurrency problems in DBMS transactions, several concurrency control techniques can be used, including locking, timestamp ordering, and optimistic concurrency control.

Locking involves acquiring locks on the data items used by transactions, preventing other transactions from accessing the same data until the lock is released. There are different types of locks, such as shared and exclusive locks, and they can be used to prevent Dirty Read and Non-Repeatable Read.

Timestamp ordering assigns a unique timestamp to each transaction and ensures that transactions execute in timestamp order. Timestamp ordering can prevent Non-Repeatable Read and Phantom Read.

Optimistic concurrency control assumes that conflicts between transactions are rare and allows transactions to proceed without acquiring locks initially. If a conflict is detected, the transaction is rolled back, and the conflict is resolved. Optimistic concurrency control can prevent Dirty Read, Non-Repeatable Read, and Phantom Read.

In conclusion, concurrency control is crucial in DBMS transactions to ensure data consistency and prevent concurrency problems such as Dirty Read, Non-Repeatable Read, and Phantom Read. By using techniques like locking, timestamp ordering, and optimistic concurrency control, developers can build robust database systems that support concurrent access while maintaining data consistency.

Concurrent Executions: Multiple transactions are allowed to run concurrently in the system.

Advantages are:

- **increased processor and disk utilization**, leading to better transaction *throughput*
 - ▶ EX: one transaction can be using the CPU while another is reading from or writing to the disk
- **reduced average response time** for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – These are the mechanisms to achieve isolation. That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules: A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed. a schedule for a set of transactions must consist of all instructions of those transactions and must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have commit instructions as the last statement. by default transaction assumed to execute commit instruction as its last step. A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedule 1: Let T_1 transfer \$50 from A to B , and T_2 transfer 10% of the balance from A to B .

A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
<pre> read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit </pre>	<pre> read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit </pre>

Schedule 2:

T_1	T_2
<pre> read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit </pre>	<pre> read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit </pre>

Schedule 3:

T_1	T_2
<pre> read (A) A := A - 50 write (A) </pre> <pre> read (B) B := B + 50 write (B) commit </pre>	<pre> read (A) temp := A * 0.1 A := A - temp write (A) </pre> <pre> read (B) B := B + temp write (B) commit </pre>

Let T_1 and T_2 be the transactions defined previously. The above schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

Note: In Schedules 1, 2 and 3, the sum $A + B$ is preserved.

Schedule 4: The following concurrent schedule does not preserve the value of $(A + B)$.

T ₁	T ₂
read (A) $A := A - 50$	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	$B := B + temp$ write (B) commit

Serializability

Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**
2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j , and at least one of these instructions wrote Q .

1. $l_i = \text{read}(Q)$, $l_j = \text{read}(Q)$. l_i and l_j don't conflict.
2. $l_i = \text{read}(Q)$, $l_j = \text{write}(Q)$. They conflict.
3. $l_i = \text{write}(Q)$, $l_j = \text{read}(Q)$. They conflict
4. $l_i = \text{write}(Q)$, $l_j = \text{write}(Q)$. They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability: If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.

We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule. Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1 , by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable

Schedule 3		Schedule 6	
T_1	T_2	T_1	T_2
read (A) write (A)	read (A) write (A)	read (A) write (A) read (B) write (B)	read (A) write (A) read (B) write (B)
read (B) write (B)	read (B) write (B)		

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
write (Q)	write (Q)

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

View Serializability

Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q ,

1. If in schedule S , transaction T_i reads the initial value of Q , then in schedule S' also transaction T_i must read the initial value of Q .
2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S' .

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

A schedule S is **view serializable** if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable. Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
write (Q)	write (Q)	write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Other Notions of Serializability

The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

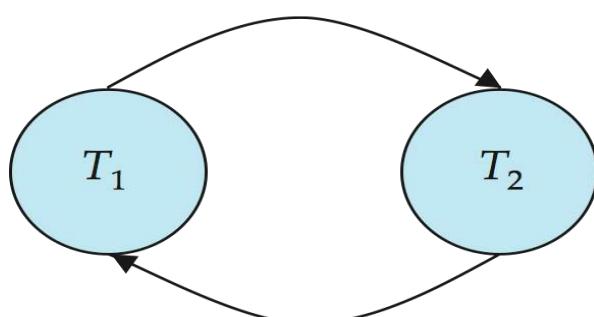
T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Determining such equivalence requires analysis of operations other than read and write.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Example



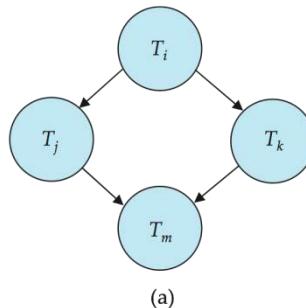
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y) read(Z)				read(V) read(W) read(W)
	read(Y) write(Y)	write(Z)		
read(U)			read(Y) write(Y) read(Z) write(Z)	
read(U) write(U)				

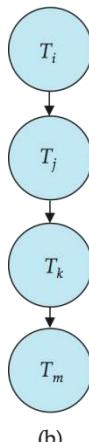
Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.

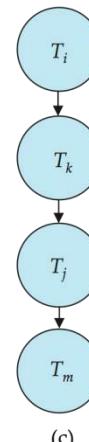
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
 - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
 - This is a linear order consistent with the partial order of the graph.
 - For example, a Serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
 - Extension to test for view Serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP-complete* problems.
 - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view Serializability can still be used.

Recoverable Schedules: Need to address the effect of transaction failures on concurrently running transactions.

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j .

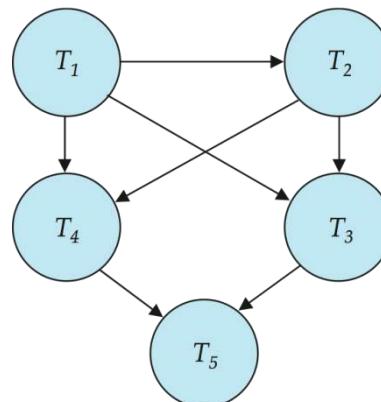
The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks: A single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
abort		read (A)



If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- ✓ Can lead to the undoing of a significant amount of work.

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of T_j .

Every cascadeless schedule is also recoverable. It is desirable to restrict the schedules to those that are cascadeless.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
 - either conflict or view serializable, and
 - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
 - Are serial schedules recoverable/cascadeless?
- Testing a schedule for Serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
 - Instead a protocol imposes a discipline that avoids nonserializable schedules.
 - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for Serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- EX: a read-only transaction that wants to get an approximate total balance of all accounts
- EX: database statistics computed for query optimization can be approximate

Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default

EX: Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL: In SQL, a transaction begins implicitly

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- A transaction in SQL ends by:
 - **Commit work** commits current transaction and begins a new one.
 - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
 - Implicit commit can be turned off by a database directive
 - EX: in JDBC, connection.setAutoCommit(false);

2. Time stamp ordering Protocol

The **Timestamp Ordering Protocol** is a concurrency control method used in database management systems to maintain the serializability of transactions. This method uses a timestamp for each transaction to determine its order in relation to other transactions. Instead of using locks, it ensures transaction order based on their timestamps.

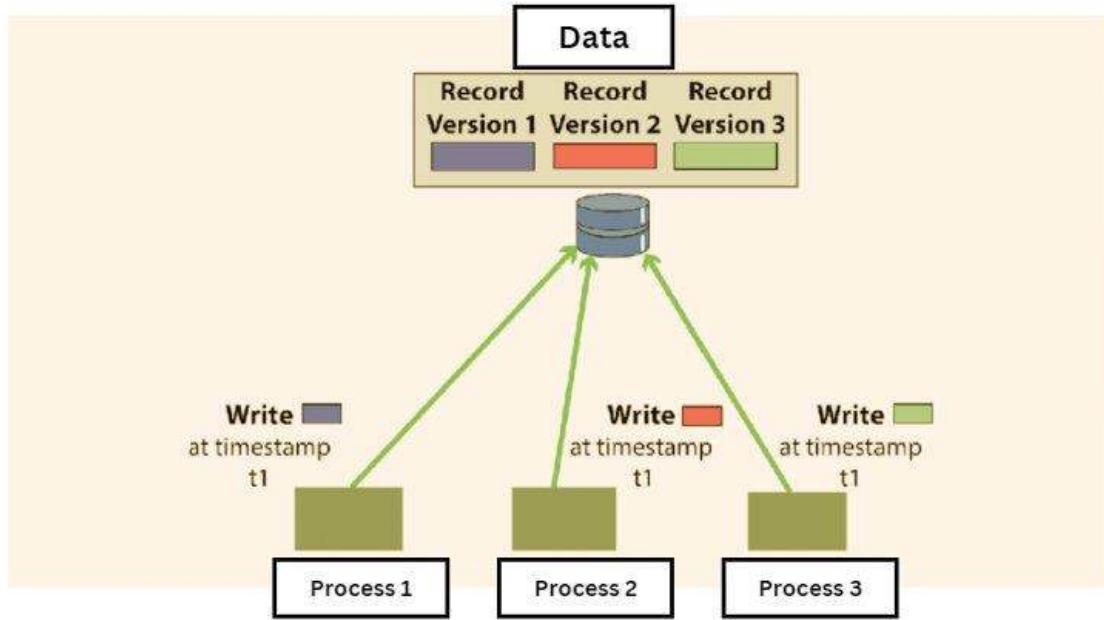
Breakdown of the Time stamp ordering protocol

- **Read Timestamp (RTS):**
 - This is the latest or most recent timestamp of a transaction that has read the data item.
 - Every time a data item X is read by a transaction T with timestamp TS, the RTS of X is updated to TS if TS is more recent than the current RTS of X.
- **Write Timestamp (WTS):**
 - This is the latest or most recent timestamp of a transaction that has written or updated the data item.
 - Whenever a data item X is written by a transaction T with timestamp TS, the WTS of X is updated to TS if TS is more recent than the current WTS of X.

The timestamp ordering protocol uses these timestamps to determine whether a transaction's request to read or write a data item should be granted. The protocol ensures a consistent ordering of operations based on their timestamps, preventing the formation of cycles and, therefore, deadlocks.

3. Multi version concurrency control

Multi version Concurrency Control (MVCC) is a technique used in database management systems to handle concurrent operations without conflicts, using multiple versions of a data item. Instead of locking the items for write operations (which can reduce concurrency and lead to bottlenecks or deadlocks), MVCC will create a separate version of the data item being modified.



Breakdown of the Multi version concurrency control (MVCC)

- **Multiple Versions:** When a transaction modifies a data item, instead of changing the item in place, it creates a new version of that item. This means that multiple versions of a database object can exist simultaneously.
- **Reads aren't Blocked:** One of the significant advantages of MVCC is that read operations don't get blocked by write operations. When a transaction reads a data item, it sees a version of that item consistent with the last time it began a transaction or issued a read, even if other transactions are currently modifying that item.
- **Timestamps or Transaction IDs:** Each version of a data item is tagged with a unique identifier, typically a timestamp or a transaction ID. This identifier determines which version of the data item a transaction sees when it accesses that item. A transaction will always see its own writes, even if they are uncommitted.
- **Garbage Collection:** As transactions create newer versions of data items, older versions can become obsolete. There's typically a background process that cleans up these old versions, a procedure often referred to as "garbage collection."
- **Conflict Resolution:** If two transactions try to modify the same data item concurrently, the system will need a way to resolve this. Different systems have different methods for conflict resolution. A common one is that the first transaction to commit will succeed, and the other transaction will be rolled back or will need to resolve the conflict before proceeding.

4. Validation concurrency control

Validation (or Optimistic) Concurrency Control (VCC) is an advanced database concurrency control technique. Instead of acquiring locks on data items, as is done in most traditional (pessimistic) concurrency control techniques, validation concurrency control allows transactions to work on private copies of database items and validates the transactions only at the time of commit.

The central idea behind optimistic concurrency control is that conflicts between transactions are rare, and it's better to let transactions run to completion and only check for conflicts at commit time.

Breakdown of Validation Concurrency Control (VCC):

- **Phases:** Each transaction in VCC goes through three distinct phases:
 - **Read Phase:** The transaction reads values from the database and makes changes to its private copy without affecting the actual database.
 - **Validation Phase:** Before committing, the transaction checks if the changes made to its private copy can be safely written to the database without causing any conflicts.
 - **Write Phase:** If validation succeeds, the transaction updates the actual database with the changes made to its private copy.
- **Validation Criteria:** During the validation phase, the system checks for potential conflicts with other transactions. If a conflict is found, the system can either roll back the transaction or delay it for a retry, depending on the specific strategy implemented.

Concurrency Control Schemes Protocols

Concurrency control schemes manage the execution of concurrent transactions to avoid conflicts and ensure consistency.

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.

Pessimistic Concurrency Control

Definition: Assumes that conflicts will occur and prevents them by locking data items.

1. Locked based concurrency control protocol
2. Timestamp based concurrency control protocol

Optimistic Concurrency Control

- **Definition:** Assumes that conflicts are rare and resolves them as they occur, rather than preventing them.
- 3 .Multi version concurrency control
 - 4 .Validation concurrency control

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.

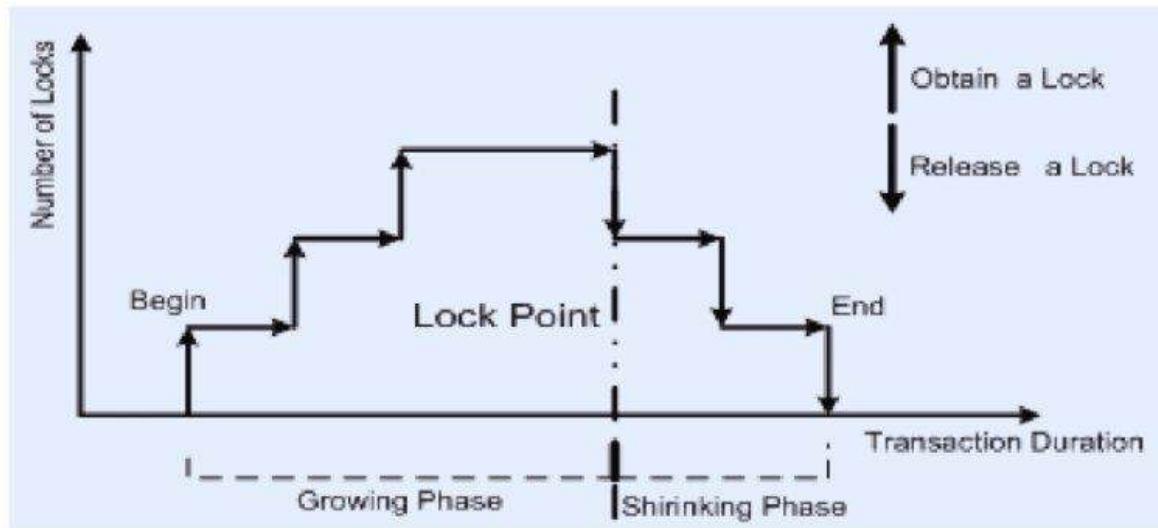
- **Shared Lock :** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item but it can not modify the data item.
- **Exclusive Lock :** Exclusive lock is also known as the write 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 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 (2PL)** is a protocol used in database management systems to control concurrency and ensure transactions are executed in a way that preserves the consistency of a database. It's called "two-phase" because, during each transaction, there are two distinct phases: the Growing phase and the Shrinking phase.

Example:

```
BEGIN TRANSACTION; LOCK TABLE Accounts IN EXCLUSIVE MODE; UPDATE Accounts SET balance = balance - 100 WHERE account_id = 'Alice'; UPDATE Accounts SET balance = balance + 100 WHERE account_id = 'Bob'; COMMIT;
```



Breakdown of the Two-Phase Locking protocol

- **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. Locks are held until the transaction commits or aborts, ensuring recoverability and avoiding cascading aborts.
- **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.
- **Lock Point:** The exact moment when the transaction switches from the Growing phase to the Shrinking phase (i.e. when it releases its first lock) is termed the lock point.

The primary purpose of the Two-Phase Locking protocol is to ensure conflict-serializability, as the protocol ensures a transaction does not interfere with others in ways that produce inconsistent results.

Recovery in DBMS

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

Failure Classification

To see where the problem has occurred, we generalize a failure into various categories, as follows –

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be –

- **Logical errors** – Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** – Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems – external to the system – that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following –

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction –

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows –

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

<T_n, Start>

- When the transaction modifies an item X, it writes logs as follows –

<T_n, X, V₁, V₂>

It reads T_n has changed the value of X, from V₁ to V₂.

- When the transaction finishes, it logs –

$\langle T_n, \text{commit} \rangle$

The database can be modified using two approaches –

- **Deferred database modification** – All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** – Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

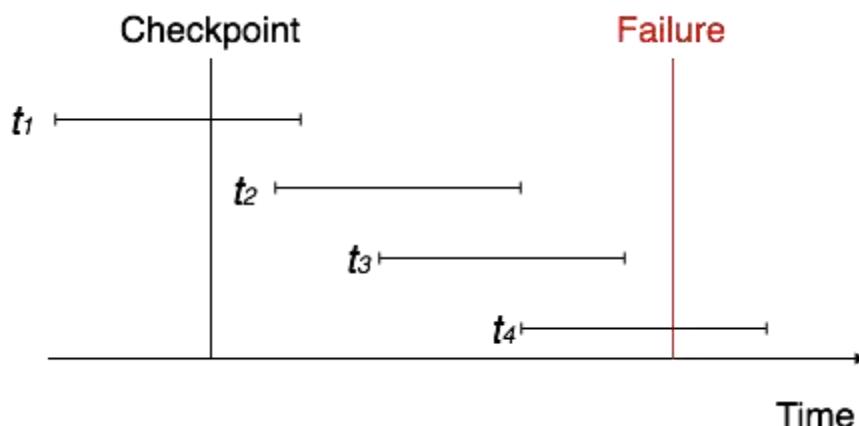
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner –



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.

- If the recovery system sees a log with $\langle T_n, \text{Start} \rangle$ and $\langle T_n, \text{Commit} \rangle$ or just $\langle T_n, \text{Commit} \rangle$, it puts the transaction in the redo-list.
- If the recovery system sees a log with $\langle T_n, \text{Start} \rangle$ but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

RELATIONAL MODEL

RELATIONAL MODEL

Relational model is simple model in which database is represented as a collection of “relations” where each relation is represented by two-dimensional table.

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

The relational model was founded by E. F. Codd of the IBM in 1972. The basic concept in the relational model is that of a relation.

Properties:

- It is column homogeneous. In other words, in any given column of a table, all items are of the same kind.
- Each item is a simple number or a character string. That is a table must be in first normal form.
- All rows of a table are distinct.
- The ordering of rows within a table is immaterial.
- The columns of a table are assigned distinct names and the ordering of these columns is immaterial.

Domain, attributes tuples and relational:

Tuple:

Each row in a table represents a record and is called a tuple .A table containing ‘n’ attributes in a record is called n-tuple.

Attributes:

The name of each column in a table is used to interpret its meaning and is called an attribute. Each table is called a relation. In the above table, account_number, branch name, balance are the attributes.

Domain:

A domain is a set of values that can be given to an attribute. So every attribute in a table has a specific domain. Values to these attributes can not be assigned outside their domains.

Relation:

A relation consists of

- **Relational schema**
- **Relation instance**

Relational Schema:

A relational schema specifies the relation’s name, its attributes and the domain of each attribute. If

R is the name of a relation and A₁, A₂,...A_n is a list of attributes representing R then R(A₁,A₂,...,A_n) is called a Relational Schema. Each attribute in this relational schema takes a value from some specific domain called domain(A_i).

Example:

PERSON (PERSON_ID:INTEGER, NAME:STRING, AGE:INTEGER, ADDRESS:STRING)

Total number of attributes in a relation denotes the degree of a relation since the PERSON relation scheme contains four attributes, so this relation is of degree 4.

Relation Instance:

A relational instance denoted as r is a collection of tuples for a given relational schema at a specific point of time.

A relation state r to the relations schema R(A₁, A₂..., A_n) also denoted by r(R) is a set of n-tuples
R{t₁, t₂, ..., t_m}

Where each n-tuple is an ordered list of n values

T=<v₁, v₂, ..., v_n>

Where each v_i belongs to domain (A_i) or contains null values.

The relation schema is also called ‘intension’ and the relation state is also called ‘extension’.

Eg: Relation schema for Student

STUDENT(rollno:string, name:string, city:string, age:integer)

Relation instance:

Student:

Rollno	Name	City	Age
101	Sujit	Bam	23
102	kunal	bbsr	22

SQL Constraints

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into the following two types,

1. **Column level constraints:** Limits only column data.
2. **Table level constraints:** Limits whole table data.

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

- NOT NULL
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK
 - DEFAULT
-

NOT NULL Constraint

By default, a [column](#) can hold NULL values. If you do not want a column to have a NULL value, use the NOT NULL constraint.

- It restricts a column from having a NULL value.
- We use [ALTER](#) statement and [MODIFY](#) statement to specify this constraint.

One important point to note about this constraint is that it cannot be defined at table level.

Example using **NOT NULL** constraint:

```
CREATE TABLE Student  
  
(    s_id int NOT NULL,  
  
     name varchar(60),  
  
     age   int  
  
);
```

The above query will declare that the **s_id** field of **Student** table will not take NULL value.

If you wish to alter the table after it has been created, then we can use the **ALTER** command for it:

```
ALTER TABLE Student  
  
MODIFY s_id int NOT NULL;
```

UNIQUE Constraint

It ensures that a column will only have unique values. A UNIQUE constraint field cannot have any duplicate data.

- It prevents two records from having identical values in a column
- We use **ALTER** statement and **MODIFY** statement to specify this constraint.

Example of UNIQUE Constraint:

Here we have a simple **CREATE** query to create a table, which will have a column **s_id** with unique values.

```
CREATE TABLE Student  
  
(    s_id int NOT NULL,  
  
     name varchar(60),  
  
     age int NOT NULL UNIQUE  
  
) ;
```

The above query will declare that the **s_id** field of **Student** table will only have unique values and wont take NULL value.

If you wish to alter the table after it has been created, then we can use the **ALTER** command for it:

```
ALTER TABLE Student  
  
MODIFY age INT NOT NULL UNIQUE;
```

The above query specifies that **s_id** field of **Student** table will only have unique value.

Primary Key Constraint

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

PRIMARY KEY constraint at Table Level

```
CREATE table Student  
(      s_id int PRIMARY KEY,  
        Name varchar(60) NOT NULL,  
        Age int);
```

The above command will creates a PRIMARY KEY on the `s_id`.

PRIMARY KEY constraint at Column Level

```
ALTER table Student  
ADD PRIMARY KEY (s_id);
```

The above command will creates a PRIMARY KEY on the `s_id`.

Foreign Key Constraint

Foreign Key is used to relate two tables. The relationship between the two tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

- This is also called a referencing key.
- We use ALTER statement and ADD statement to specify this constraint.

To understand FOREIGN KEY, let's see its use, with help of the below tables:

Customer_Detail Table

c_id	Customer_Name	address
101	Adam	Noida
102	Alex	Delhi
103	Stuart	Rohtak

Order_Detail Table

Order_id	Order_Name	c_id
10	Order1	101
11	Order2	103
12	Order3	102

In **Customer_Detail** table, **c_id** is the primary key which is set as foreign key in **Order_Detail** table. The value that is entered in **c_id** which is set as foreign key in **Order_Detail** table must be present in **Customer_Detail** table where it is set as primary key. This prevents invalid data to be inserted into **c_id** column of **Order_Detail** table.

If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

FOREIGN KEY constraint at Table Level

```
CREATE table Order_Detail(
    order_id int PRIMARY KEY,
    order_name varchar(60) NOT NULL,
    c_id int FOREIGN KEY REFERENCES
Customer_Detail(c_id)
);
```

In this query, **c_id** in table Order_Detail is made as foreign key, which is a reference of **c_id** column in Customer_Detail table.

FOREIGN KEY constraint at Column Level

```
ALTER table Order_Detail
ADD FOREIGN KEY (c_id) REFERENCES
Customer_Detail(c_id);
```

Behaviour of Foreign Key Column on Delete

There are two ways to maintain the integrity of data in Child table, when a particular record is deleted in the main table. When two tables are connected with Foreign key, and certain data in the main table is deleted, for which a record exists in the child table, then we must have some mechanism to save the integrity of data in the child table.

1. **On Delete Cascade** : This will remove the record from child table, if that value of foreign key is deleted from the main table.
2. **On Delete Null** : This will set all the values in that record of child table as NULL, for which the value of foreign key is deleted from the main table.
3. If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

```
ERROR : Record in child table exist
```

CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

Using **CHECK** constraint at Table Level

```
CREATE table Student (  
    s_id int NOT NULL CHECK(s_id > 0),  
    Name varchar(60) NOT NULL,  
    Age int  
) ;
```

The above query will restrict the **s_id** value to be greater than zero.

Using **CHECK** constraint at Column Level

```
ALTER table Student ADD CHECK(s_id > 0);
```

Form of basic SQL query in DBMS

The basic form of an SQL query, specifically when retrieving data, is composed of a combination of clauses. The most elementary form of an SQL query for data retrieval can be represented as

Syntax

```
SELECT [DISTINCT] column1, column2, ...
```

```
FROM tablename
```

```
WHERE condition;
```

Let's break it down:

1. **SELECT Clause:** This is where you specify the columns you want to retrieve. Use an asterisk (*) to retrieve all columns.
2. **FROM Clause:** This specifies from which table or tables you want to retrieve the data.
3. **WHERE Clause (optional):** This allows you to filter the results based on a condition.
4. **DISTINCT Clause (optional):** is an optional keyword indicating that the answer should not contain duplicates. Normally if we write the SQL without DISTINCT operator then it does not eliminate the duplicates.

Here are the primary components of SQL queries:

- **SELECT:** Retrieves data from one or more tables.
- **FROM:** Specifies the table from which you're retrieving the data.
- **WHERE:** Filters the results based on a condition.
- **GROUP BY:** Groups rows that have the same values in specified columns.
- **HAVING:** Filters the result of a GROUP BY.
- **ORDER BY:** Sorts the results in ascending or descending order.
- **JOIN:** Combines rows from two or more tables based on related columns.

To provide a more holistic view, here are a few more [SQL](#) examples, keeping them as basic as possible:

1. Retrieve all columns from a table:

Syntax

```
SELECT * FROM tablename;
```

2. Retrieve specific columns from a table:

Syntax

```
SELECT column1, column2 FROM tablename;
```

3. Retrieve data with a condition:

Syntax

```
SELECT column1, column2 FROM tablename WHERE column1 = 'value';
```

4. Sort retrieved data:

Syntax

```
SELECT column1, column2 FROM tablename ORDER BY column1 ASC;
```

Regular expressions in the SELECT Command

SQL provides support for pattern matching through the LIKE operator, along with the use of the wild-card symbols.

Regular expressions: is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching.

Examples:

Finds Names that start or ends with "a"

Finds names that start with "a" and are at least 3 characters in length.

LIKE: The LIKE operator is used in a 'WHERE' clause to search for a specified pattern in a column

wild-card: There are two primary wildcards used in conjunction with the `LIKE` operator

percent sign (%) Represents zero, one, or multiple characters

underscore sign(_) Represents a single character

Here's a breakdown of how you can use these wildcards with the `LIKE` operator:

Using `%` Wildcard

1. Find values that start with a specific pattern:

Syntax

```
SELECT column_name
```

```
FROM table_name
```

```
WHERE column_name LIKE 'pattern%';
```

For example, to find all customers whose names start with "Ma":

Example

```
SELECT FirstName  
      FROM Customers  
     WHERE FirstName LIKE 'Ma%';
```

2. Find values that end with a specific pattern:

Syntax

```
SELECT column_name  
      FROM table_name  
     WHERE column_name LIKE '%pattern';
```

For instance, to find all products that end with "ing":

Example

```
SELECT ProductName  
      FROM Products  
     WHERE ProductName LIKE '%ing';
```

3. Find values that have a specific pattern anywhere:

Syntax

```
SELECT column_name  
      FROM table_name  
     WHERE column_name LIKE '%pattern%';
```

Example, to find all books that have the word "life" anywhere in the title:

Example

```
SELECT BookTitle  
      FROM Books  
     WHERE BookTitle LIKE '%life%';
```

Using `_` Wildcard

1. Find values of a specific length where you only know some characters:

Syntax

```
SELECT column_name  
      FROM table_name  
     WHERE column_name LIKE 'p_tern';
```

For instance, if you're looking for a five-letter word where you know the first letter is "h" and the third letter is "l", you could use:

Example

```
SELECT Word  
      FROM Words  
     WHERE Word LIKE 'h_l__';
```

Combining `%` and `_`

You can use both wildcards in the same pattern. For example, to find any value that starts with "A", followed by two characters, and then "o":

Example

```
SELECT column_name  
      FROM table_name  
     WHERE column_name LIKE 'A__o%';
```

Querying Relational Data in DBMS

A relational [database](#) query is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students AGE less than 18 or all students enrolled in particular course.

The **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax in Mysql

SELECT column1, column2, ...

FROM table_name;

If you want to select all the fields available in the table, use the following syntax:

Syntax in Mysql

*SELECT * FROM table_name;*

The symbol '*' means that we retain all fields of selected tuples in the result.

We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

Example:

```
SELECT * FROM Students WHERE age < 18;
```

The condition **age < 18** in the WHERE clause specifies that we want to select only tuples in which the age field has a value less than 18.

In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. we can compute the student_id and First_name of students who are younger than 18 with the following query:

Example:

```
SELECT ID, FirstName FROM Students WHERE age < 18;
```

SQL Aliases

Aliases are the temporary names given to tables or columns. An alias is created with the **AS** keyword.

Alias Column Syntax in Mysql

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

Alias Table Syntax in Mysql

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

Example:

```
SELECT studentID AS ID,  
FROM students AS S;
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

SELECT data from Multiple Tables

We can also combine information from multiple tables.

Syntax in Mysql

```
SELECT table1.column1, table2.column2  
FROM table1, table2  
WHERE table1.column1 = table2.column1;
```

Example:

```
SELECT S.name, E.cid  
FROM Students AS S, Enrolled AS E  
WHERE S.sid = E.sid;
```

Views in DBMS

A view is a table whose rows are not explicitly stored, a view is a virtual table based on the result-set of an [SQL](#) statement. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written [SQL query](#) to create a view.

A view is generated to show the information that the end-user requests the data according to specified needs rather than complete information of the table.

Advantages of View over [database](#) tables

- Using Views, we can join multiple tables into a single virtual table.
- Views hide data complexity.
- In the [database](#), views take less space than tables for storing data because the database contains only the view definition.
- Views indicate the subset of that data, which is contained in the tables of the database.

Creating Views

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

Syntax in Mysql

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

Example:

```
CREATE VIEW Students_CSE AS  
SELECT Roll_no, Name  
FROM Students  
WHERE Branch = 'CSE';
```

Updating a View

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

Syntax in Mysql

```
CREATE OR REPLACE VIEW view_name AS  
SELECT column1, column2, ...
```

FROM table_name

WHERE condition;

The following [SQL](#) adds the "Mobile" column to the "Students_CSE" view:

Example:

```
CREATE OR REPLACE VIEW Students_CSE AS
```

```
SELECT Roll_no, Name, Mobile
```

```
FROM Students
```

```
WHERE Branch = 'CSE';
```

CREATE VIEW defines a view on a set of tables or views or both.

REPLACE VIEW redefines an existing view or, if the specified view does not exist,

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 in Mysql

```
INSERT INTO view_name(column1, column2, ...)
```

```
VALUES(value1, value2, ....);
```

Example:

```
INSERT INTO Students_CSE(Roll_no, Name, Mobile)
```

```
VALUES(521, 'ram', 9988776655);
```

Deleting a row in a view

Deleting rows from a view is also as simple as deleting rows from a table. We can use the **DELETE** statement of [SQL](#) to delete rows from a view.

Syntax in Mysql

```
DELETE FROM view_name
```

```
WHERE condition;
```

Example:

```
DELETE FROM Students_CSE  
WHERE Name="ram";
```

Querying a View

We can query the view as follows

Syntax in Mysql

```
SELECT * FROM view_name
```

Example:

```
SELECT * FROM Students_CSE;
```

Dropping a View

In order to delete a view in a [database](#), we can use the DROP VIEW statement.

Syntax in Mysql

```
DROP FROM view_name
```

Example:

```
DROP FROM Students_CSE;
```

RELATIONAL CALCULUS

Relational calculus is a non-procedural query language used in database management systems (DBMS). Unlike procedural query languages, such as SQL, which specify how to retrieve data, relational calculus focuses on what data to retrieve without specifying the procedure for fetching that data. There are two main types of relational calculus:

1. Tuple Relational Calculus (TRC):

- **Definition:** TRC uses tuple variables to represent tuples in a relation. A tuple variable can take any tuple as its value from the relation.
- **Syntax:** The basic form of a TRC query is $\{T \mid P(T)\}$, where T is a tuple variable and $P(T)$ is a condition or predicate that T must satisfy.
- **Example:** To retrieve the names of employees who work in the 'Sales' department, we might write: $\{ t.Name \mid Employee(t) \wedge t.Department = 'Sales' \}$ Here, t is a tuple variable ranging over the `Employee` relation, and we are selecting the `Name` attribute of tuples where the `Department` is 'Sales'.

2. Domain Relational Calculus (DRC):

- **Definition:** DRC uses domain variables that take values from domains (attribute sets) rather than entire tuples.
- **Syntax:** The basic form of a DRC query is $\{ <d_1, d_2, \dots, d_n \mid P(d_1, d_2, \dots, d_n) \}$, where $<d_1, d_2, \dots, d_n \rangle$ is a list of domain variables and $P(d_1, d_2, \dots, d_n)$ is a predicate that these variables must satisfy.
- **Example:** To retrieve the names of employees who work in the 'Sales' department, we might write: $\{ <Name> \mid \exists Department (Employee(Name, Department) \wedge Department = 'Sales') \}$ Here, `Name` and `Department` are domain variables, and we are selecting the `Name` where the `Department` is 'Sales'.

Key Points

- **Non-Procedural:** Both TRC and DRC are non-procedural, meaning they specify what to retrieve rather than how to retrieve it.
- **Predicate Logic:** They are based on predicate logic, involving the use of logical connectives and quantifiers (\forall for "for all" and \exists for "there exists").
- **Expressive Power:** Relational calculus is powerful and expressive, capable of expressing complex queries in a concise manner.
- **Safety:** Queries in relational calculus must be safe, meaning they should produce a finite set of results. Unsafe queries, which could theoretically produce an infinite number of results, are not allowed.

Safety of Queries

A query in relational calculus is considered safe if it guarantees to return a finite set of results. This is important to ensure that the query execution terminates and produces a usable output.

Examples

- Tuple Relational Calculus Example:** Retrieve the names of employees who earn more than \$50,000. { t.Name | Employee(t) \wedge t.Salary > 50000 }
- Domain Relational Calculus Example:** Retrieve the names and salaries of employees who work in the 'IT' department. { <Name, Salary> | \exists Department (Employee(Name, Salary, Department) \wedge Department = 'IT') }

Comparison with SQL

- **Declarative Nature:** Similar to SQL, relational calculus is declarative in nature.
- **Foundation:** Relational calculus provides a formal foundation for relational query languages like SQL.
- **Usability:** While relational calculus is more theoretical and used mainly for formal definitions and proofs, SQL is more practical and widely used in real-world applications.

Understanding relational calculus is essential for grasping the theoretical underpinnings of query languages and database management systems, providing insights into how complex queries can be formulated and optimized.

Expressive Power of Algebra and calculus

Relational Algebra and Relational Calculus are both fundamental concepts in the theory of relational databases. Both are used to express queries, but they do so in different ways. Here's a detailed look at the expressive power of each:

Relational Algebra

Definition: Relational Algebra is a procedural query language, meaning it defines a sequence of operations to be performed on the database to retrieve the desired results. It uses a set of operations to manipulate relations (tables).

Operations:

- **Basic Operations:** Selection (σ), Projection (π), Cartesian Product (\times), Union (\cup), Set Difference ($-$), and Rename (ρ).
- **Derived Operations:** Join (natural join, equi-join, etc.), Division, Intersection (\cap), etc.

Expressive Power:

- **Procedural Nature:** Relational Algebra explicitly specifies how to perform operations to obtain the result. This procedural nature allows detailed control over the query execution.
- **Complex Queries:** By combining basic operations, complex queries can be constructed. For example, a join operation can be achieved by a combination of Cartesian Product and Selection.

- **Closure Property:** Relational Algebra is closed under its operations, meaning the result of any operation is a relation which can be further used as input for other operations.

Example: Retrieve the names of employees who work in the 'Sales' department.
 $\pi \text{ Name} (\sigma_{\text{Department} = \text{'Sales'}} (\text{Employee}))$

Relational Calculus

Definition: Relational Calculus is a non-procedural query language, meaning it focuses on what data to retrieve rather than how to retrieve it. It uses logical formulas to describe the desired result set.

Types:

- **Tuple Relational Calculus (TRC):** Uses tuple variables to range over tuples.
 - **Example:** $\{t.\text{Name} \mid \text{Employee}(t) \wedge t.\text{Department} = \text{'Sales'}\}$
- **Domain Relational Calculus (DRC):** Uses domain variables to range over domain values.
 - **Example:** $\{\langle \text{Name} \rangle \mid \exists \text{Department} (\text{Employee}(\text{Name}, \text{Department}) \wedge \text{Department} = \text{'Sales'})\}$

Expressive Power:

- **Declarative Nature:** Relational Calculus specifies what to retrieve without specifying how. This makes it more abstract and closer to natural language queries.
- **Predicate Logic:** Based on predicate logic, it uses logical connectives (AND, OR, NOT) and quantifiers (\forall, \exists) to express queries.
- **Complex Conditions:** It can naturally express complex conditions and nested queries.

Comparison:

1. **Equivalence:**
 - Relational Algebra and Relational Calculus are equivalent in expressive power, meaning any query that can be expressed in one can also be expressed in the other. This equivalence is formalized by Codd's Theorem.
2. **Optimization:**
 - Relational Algebra is often used as an intermediate language in query optimization and execution. Its procedural nature makes it suitable for optimizing the sequence of operations.
 - Relational Calculus, being more abstract, is less concerned with optimization and more with the specification of what is needed.
3. **Usability:**
 - Relational Algebra is closer to implementation and is often used in query engines.
 - Relational Calculus is more suited for theoretical work and reasoning about queries.
4. **Examples:**

- **Relational Algebra:** Retrieve the names and departments of employees who earn more than \$50,000. $\pi \text{ Name, Department} (\sigma \text{ Salary} > 50000 (\text{Employee}))$
- **Tuple Relational Calculus:** Retrieve the same information. $\{t.\text{Name}, t.\text{Department} \mid \text{Employee}(t) \wedge t.\text{Salary} > 50000\}$
- **Domain Relational Calculus:** Retrieve the same information. $\{<\text{Name}, \text{Department}> \mid \exists \text{Salary} (\text{Employee}(\text{Name}, \text{Salary}, \text{Department}) \wedge \text{Salary} > 50000)\}$

In summary, while both Relational Algebra and Relational Calculus are equivalent in terms of expressive power, they serve different purposes. Relational Algebra is procedural and better suited for optimization and execution, whereas Relational Calculus is declarative and ideal for specifying the requirements of queries in a more abstract manner.

Relational Calculus

There is an alternate way of formulating queries known as Relational Calculus. Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results. The relational calculus tells what to do but never explains how to do. Most commercial relational languages are based on aspects of relational calculus including SQL-QBE and QUIL.

Why it is called Relational Calculus?

It is based on Predicate calculus, a name derived from branch of symbolic language. A predicate is a truth-valued function with arguments. On substituting values for the arguments, the function result in an expression called a proposition. It can be either true or false. It is a tailored version of a subset of the Predicate Calculus to communicate with the relational database.

Many of the calculus expressions involves the use of Quantifiers. There are two types of quantifiers:

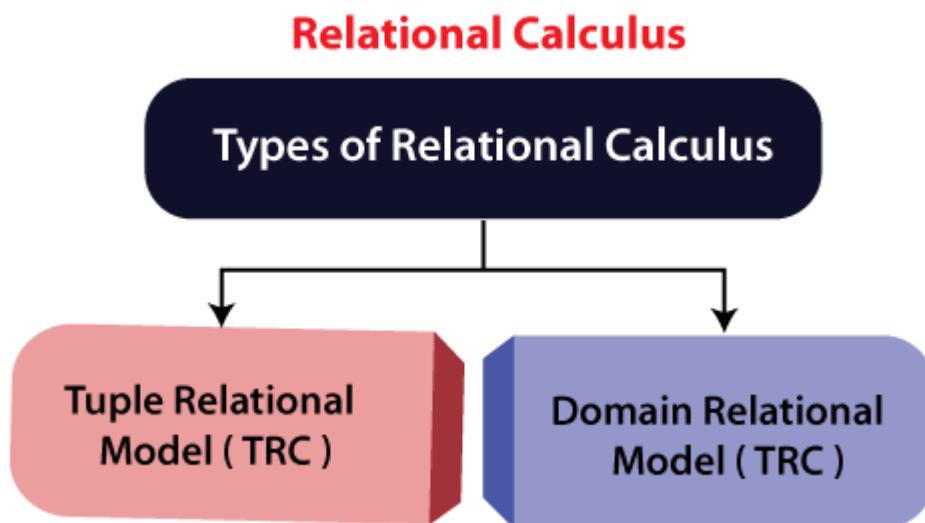
- **Universal Quantifiers:** The universal quantifier denoted by \forall is read as for all which means that in a given set of tuples exactly all tuples satisfy a given condition.
- **Existential Quantifiers:** The existential quantifier denoted by \exists is read as for all which means that in a given set of tuples there is at least one occurrences whose value satisfy a given condition.

Before using the concept of quantifiers in formulas, we need to know the concept of Free and Bound Variables.

A tuple variable t is bound if it is quantified which means that if it appears in any occurrences a variable that is not bound is said to be free.

Free and bound variables may be compared with global and local variable of programming languages.

Types of Relational calculus:



1. Tuple Relational Calculus (TRC)

It is a non-procedural query language which is based on finding a number of tuple variables also known as range variable for which predicate holds true. It describes the desired information without giving a specific procedure for obtaining that information. The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation. The result of the relation can have one or more tuples.

Notation:

A Query in the tuple relational calculus is expressed as following notation

1. $\{T \mid P(T)\}$ or $\{T \mid \text{Condition}(T)\}$

Where

T is the resulting tuples

$P(T)$ is the condition used to fetch T .

For example:

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

Output: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (\exists) and Universal Quantifiers (\forall).

For example:

1. { R | $\exists T \in \text{Authors}(T.\text{article} = \text{'database'} \text{ AND } R.\text{name} = T.\text{name})$ }

Output: This query will yield the same result as the previous one.

2. Domain Relational Calculus (DRC)

The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes. Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives \wedge (and), \vee (or) and \neg (not). It uses Existential (\exists) and Universal Quantifiers (\forall) to bind the variable. The QBE or Query by example is a query language related to domain relational calculus.

Notation:

1. { a₁, a₂, a₃, ..., a_n | P (a₁, a₂, a₃, ..., a_n)}

Where

a₁, a₂ are attributes

P stands for formula built by inner attributes

For example:

1. {< article, page, subject > | $\in \text{javatpoint} \wedge \text{subject} = \text{'database'}$ }

Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

Tuple Relational Calculus (TRC) in DBMS

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 t 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 " \wedge " 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

$\{t \mid P(t)\}$

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 (\vee), AND (\wedge), NOT(\neg).

It also uses quantifiers:
 $\exists t \in r (Q(t))$ = "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true.

$\forall t \in r (Q(t)) = Q(t)$ is true "for all" tuples in relation r .

Domain Relational Calculus (DRC)

Domain Relational Calculus is similar to Tuple Relational Calculus, where it makes a list of the attributes that are to be chosen from the relations as per the conditions.

$\{< a_1, a_2, a_3, \dots, a_n > \mid P(a_1, a_2, a_3, \dots, a_n)\}$

where a_1, a_2, \dots, a_n are the attributes of the relation and P is the condition.

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 amount.

```
{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, $t[\text{amount}]$ is known as a tuple variable.

Example 2: Find the loan number for each loan of an amount greater or equal to 10000.

```
{t | ∃ s ∈ loan(t[loan number] = s[loan number]
                  ∧ s[amount] >= 10000)}
```

Resulting relation:

Loan number
L33
L35
L98

Example 3: Find the names of all customers who have a loan and an account at the bank.

```
{t | ∃ s ∈ borrower( t[customer-name] = s[customer-name])
      ∧ ∃ u ∈ depositor( t[customer-name] = u[customer-name])}
```

Resulting relation:

Customer name
Saurabh
Mehak

Example 4: Find the names of all customers having a loan at the “ABC” branch.

```
{t | ∃ s ∈ borrower( t[customer-name] = s[customer-name]
                  ∧ ∃ u ∈ loan(u[branch-name] = “ABC” ∧ u[loan-number] = s[loan-number]))}
```

Resulting relation:

Customer name
Saurabh

Relational Calculus uses declarative Language, unlike Relational Algebra which uses Procedural Language.

Relational Calculus in DBMS

Relational Calculus in DBMS is just another way of formulating queries. It is non-procedural and a declarative query language. Let's first understand the difference between procedural and declarative query language.

Procedural Language	Declarative Language
These languages define how to get the results from the database.	These languages define what to get from the database.
An example of procedural language is Relational Algebra.	An example of Declarative Language is Relational Calculus.

Hence relational calculus is concerned with what to do rather than how to do it. Simply put, it only provides the information on the query description rather than the methods to do it.

Why is it called Relational Calculus?

It is called Relational Calculus because it is used to manipulate and describe relational data in DBMS using a mathematical and logical approach. The “relational” part refers to representing and organizing the data in a tabular form with the help of relations or tables. The “calculus” part signifies a system of calculation and reasoning.

Relational Calculus is based on predicate calculus, a truth-valued function with arguments. When the values of the arguments are substituted in the predicate calculus, the resulting expression is called a proposition which can be either true or false.

Relational calculus is a tailored version of a subset of Predicate Calculus, which helps communicate with the relational database.

Also read, [File System vs DBMS](#)

Common Notations Used in Relational Calculus

\wedge - represents AND.

\vee - represents OR.

\neg - represents NOT.

\in - represents Belongs to.

\forall - represents that in a set of tuples, all of them satisfy the condition (Universal Quantifier).

\exists - represents that in a set of tuples, there is at least one occurrence that satisfies the condition (Universal Quantifier).

Types of Relational Calculus

There are two types of Relational Calculus.

Tuple Relational Calculus (TRC)

It is a non-procedural query language with which we find tuples that hold true for a given condition. These given conditions are called predicates. Tuple Relational Calculus describes the desired information without providing specific information for obtaining that information, i.e., specifies 'what' but not 'how.'

It goes to each table row and checks whether the predicate condition is satisfied (true) or not (false). It returns the tuple(s) which hold true for the predicate condition.

Syntax

```
{t | P(t)}
```

Here, t represents the tuples returned as results, and P(t) is the predicate logic condition or expression.

**Note: TRC is used as a theoretical foundation for optimizing the queries. However, it cannot be executed in SQL-based RDBMS such as MySQL, SQLite, or PostgreSQL.*

Example

Consider the following Ninja table

Ninja_Name	Course_Enrolled	Age
Ninja_1	Java	18
Ninja_5	C++	16
Ninja_3	Web development	20
Ninja_4	C++	17

Query 1: Create a TRC query to get data of all the Ninja's whose age>=17.

TRC Query

```
{t | t.Ninja (t) ^ t.Age >= 17}
```

Explanation: The tuple variable (t) goes to every tuple of the Ninja table. Each row checks the age of the Ninjas, and only those tuples are returned in the result whose age is greater than or equal to 17.

The query can be interpreted as "Return all the tuples t which belong to table Ninja and have an age greater than or equal to 17."

Result

Ninja_Name	Course_Enrolled	Age
Ninja_1	Java	18
Ninja_3	Web development	20

Ninja_Name	Course_Enrolled	Age
Ninja_4	C++	17

Query 2: Create a TRC query to get names of all the Ninja's who are enrolled in C++ course.

TRC Query

```
{t| ∃ n ∈ Ninja(t.Ninja_Name = n.Ninja_Name ∧ t.Course_Enrolled ="C++")}
```

Explanation: The above query returns the Ninja_Name for all the tuples which have Course Enrolled as C++. The result is returned in the tuple t.

Result

Ninja_Name
Ninja_5
Ninja_4

Domain Relational Calculus (DRC)

The domain of the attributes is used in Domain Relational Calculus. It uses domain variables to get the column values needed based on predicate conditions.

Syntax:

```
{<x1,x2,...,xn> | P(x1,x2,...,xn)}
```

Here, $\langle x_1, x_2, \dots, x_n \rangle$ represents the domain variables used to get the column values, and $P(x_1, x_2, \dots, x_n)$ is the predicate which is the condition for results.

**Note: DRC is a theoretical foundation for manipulating and describing data in a relational database. However, it cannot be executed in SQL-based RDBMS such as MySQL, SQLite, or PostgreSQL.*

Example

Consider the following Ninja table

Ninja_id	Gender	Age
CN_1	M	18
CN_2	F	16
CN_3	F	20
CN_4	M	17

Query 1: Create a DRC query to get data of all the Ninja's whose Gender is M.

DRC Query:

$$\{ \langle x_1, x_2, x_3 \rangle \mid \langle x_1, x_3 \rangle \in \text{Ninja} \wedge x_2 = 'M' \}$$

Explanation: In the above query, $x_1(\text{Ninja_id})$, $x_2(\text{Gender})$, and $x_3(\text{Age})$ represent the columns that are needed in the result. The predicate condition states that x_1 and x_3 should be present while x_2 has to follow a matching condition for each row; that is, x_2 should be equal to 'M.'

Result

Ninja_id	Gender	Age
CN_1	M	18
CN_4	M	17

Query 2: Create a DRC query to get Ninja_id of all the Ninja's whose Age < 18.

DRC Query:

$$\{ \langle x_1 \rangle \mid \langle x_1 \rangle \in \text{Ninja} \wedge x_3 < 18 \}$$

Explanation: In the above query, $x_1(\text{Ninja_id})$ represents the column needed in the result. The predicate condition states that x_3 has to follow a matching condition for each row that $x_3 < 18$.

Result

Ninja_id
CN_2
CN_4

Use Cases of Relational Calculus in DBMS

Below are the use cases of relational calculus in DBMS.

1. Relational Calculus helps in formulating complex queries which involves multiple tables, aggregations and conditions. It helps the users to retrieve the desired data from the database.
2. Relational Calculus also helps in data validation. It enables users to check whether data satisfies the conditions and also helps in identifying errors or inconsistencies.
3. Relational Calculus plays an important role in Database Modelling and Design. It helps in defining the schema, integrity constraints, primary and foreign key constraints, and other things required in database design.
4. It can perform advanced data analysis like sorting, filtering, aggregation and grouping. It helps users to get meaningful insights from large datasets.

Limitations of Relational Calculus in DBMS

Below are the limitations of relational calculus in DBMS.

1. It is not as user friendly as other query languages such as SQL. It lacks operations that make complex queries efficient and concise to write.

2. It is a declarative language which only focuses on what to retrieve rather than how to retrieve.
3. It does not provide a mechanism which optimizes the queries automatically and also it does not support recursive queries.
4. It may not be suitable for dealing with semi-structured or non relational data such as graph data or JSON documents.

MYSQL

It is freely available open source Relational Database Management System (RDBMS) that uses **Structured Query Language(SQL)**. In MySQL database , information is stored in Tables. A single MySQL database can contain many tables at once and store thousands of individual records.

SQL (Structured Query Language)

SQL is a language that enables you to create and operate on relational databases, which are sets of related information stored in tables.

DIFFERENT DATA MODELS

A **data model** refers to a set of concepts to describe the structure of a **database**, and certain constraints (restrictions) that the database should obey. The four data model that are used for database management are :

1. **Relational data model** : In this data model, the data is organized into tables (i.e. rows and columns). These tables are called relations.
2. **Hierarchical data model**
3. **Network data model**
4. **Object Oriented data model**

RELATIONAL MODEL TERMINOLOGY

1. **Relation** : A table storing logically related data is called a Relation.
2. **Tuple** : A **row of a relation** is generally referred to as a tuple.
3. **Attribute** : A **column** of a relation is generally referred to as an attribute.
4. **Degree** : This refers to the **number of attributes** in a relation.
5. **Cardinality** : This refers to the **number of tuples** in a relation.
6. **Primary Key** : This refers to a set of one or more attributes that can uniquely identify tuples within the relation.
7. **Candidate Key** : All attribute combinations inside a relation that can serve as primary key are candidate keys as these are candidates for primary key position.
8. **Alternate Key** : A candidate key that is not primary key, is called an alternate key.
9. **Foreign Key** : A non-key attribute, whose values are derived from the primary key of some other table, is known as foreign key in its current table.

REFERENTIAL INTEGRITY

- A referential integrity is a system of rules that a DBMS uses to ensure that relationships between records in related tables are valid, and that users don't accidentally delete or change related data. This integrity is ensured by foreign key.

CLASSIFICATION OF SQL STATEMENTS

SQL commands can be mainly divided into following categories:

1. Data Definition Language(DDL) Commands

Commands that allow you to perform task, related to data definition e.g;

- Creating, altering and dropping.
- Granting and revoking privileges and roles.
- Maintenance commands.

2. Data Manipulation Language(DML) Commands

Commands that allow you to perform data manipulation e.g., retrieval, insertion, deletion and modification of data stored in a database.

3. Transaction Control Language(TCL) Commands

Commands that allow you to manage and control the transactions e.g.,

- Making changes to database, permanent
- Undoing changes to database, permanent
- Creating savepoints
- Setting properties for current transactions.

MySQL ELEMENTS

1. Literals
2. Datatypes
3. Nulls
4. Comments

LITERALS

It refer to a fixed data value. This fixed data value may be of character type or numeric type. For example, 'replay' , 'Raj' , '8' , '306' are all character literals.

Numbers not enclosed in quotation marks are numeric literals. E.g. 22 , 18 , 1997 are all numeric literals.

Numeric literals can either be integer literals i.e., without any decimal or be real literals i.e. with a decimal point e.g. 17 is an integer literal but 17.0 and 17.5 are real literals.

DATA TYPES

Data types are means to identify the type of data and associated operations for handling it. MySQL data types are divided into three categories:

- Numeric
- Date and time
- String types

Numeric Data Type

1. int – used for number without decimal.
2. Decimal(m,d) – used for floating/real numbers. m denotes the total length of number and d is number of decimal digits.

Date and Time Data Type

1. date – used to store date in YYYY-MM-DD format.
2. time – used to store time in HH:MM:SS format.

String Data Types

1. char(m) – used to store a fixed length string. m denotes max. number of characters.
2. varchar(m) – used to store a variable length string. m denotes max. no. of characters.

DIFFERENCE BETWEEN CHAR AND VARCHAR DATA TYPE

S.NO.	Char Datatype	Varchar Datatype
1.	It specifies a fixed length character String.	It specifies a variable length character string.
2.	When a column is given datatype as CHAR(n), then MySQL ensures that all values stored in that column have this length i.e. n bytes. If a value is shorter than this length n then blanks are added, but the size of value remains n bytes.	When a column is given datatype as VARCHAR(n), then the maximum size a value in this column can have is n bytes. Each value that is stored in this column store exactly as you specify it i.e. no blanks are added if the length is shorter than maximum length n.

NULL VALUE

If a column in a row has no value, then column is said to be **null** , or to contain a null. You should use a null value when the actual value is not known or when a value would not be meaningful.

DATABASE COMMANDS

1. VIEW EXISTING DATABASE

To view existing database names, the command is : **SHOW DATABASES ;**

2. CREATING DATABASE IN MYSQL

For creating the database in MySQL, we write the following command : **CREATE DATABASE <database name> ;**

e.g. In order to create a database Student, command is :

CREATE DATABASE Student ;

3. ACCESSING DATABASE

For accessing already existing database , we write :

USE <database name> ;

e.g. to access a database named Student , we write command as :

USE Student ;

4. DELETING DATABASE

For deleting any existing database , the command is :

DROP DATABASE <database name> ;

e.g. to delete a database , say student, we write command as ; **DROP DATABASE Student ;**

5. VIEWING TABLE IN DATABASE

In order to view tables present in currently accessed database , command is : **SHOW TABLES ;**

CREATING TABLES IN MYSQL

- Tables are created with the CREATE TABLE command. When a table is created, its columns are named, data types and sizes are supplied for each column.

Syntax of CREATE TABLE command

is : **CREATE TABLE <table-name>**

(<column name> <data type> ,
 <column name> <data type> ,
 );

E.g. in order to create table EMPLOYEE given below :

ECODE	ENAME	GENDER	GRADE	GROSS
-------	-------	--------	-------	-------

We write the following command :

```
CREATE TABLE employee  
( ECODE integer ,  
  ENAME varchar(20) ,  
  GENDER char(1) ,  
  GRADE char(2) ,  
  GROSS integer );
```

INSERTING DATA INTO TABLE

- The rows are added to relations(table) using INSERT command of SQL. Syntax of INSERT is : **INSERT INTO <tablename> [<column list>]
VALUE (<value1> , <value2> ,) ;**

e.g. to enter a row into EMPLOYEE table (created above), we write command as :

```
INSERT INTO employee  
VALUES(1001 , 'Ravi' , 'M' , 'E4' , 50000);
```

OR

```
INSERT INTO employee (ECODE , ENAME , GENDER , GRADE , GROSS)  
VALUES(1001 , 'Ravi' , 'M' , 'E4' , 50000);
```

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000

In order to insert another row in EMPLOYEE table , we write again INSERT command :

```
INSERT INTO employee  
VALUES(1002 , 'Akash' , 'M' , 'A1' , 35000);
```

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000
1002	Akash	M	A1	35000

INSERTING NULL VALUES

- To insert value NULL in a specific column, we can type NULL without quotes and NULL will be inserted in that column. E.g. in order to insert NULL value in ENAME column of above table, we write INSERT command as :

```
INSERT INTO EMPLOYEE  
VALUES (1004 , NULL , 'M' , 'B2' , 38965 ) ;
```

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000
1002	Akash	M	A1	35000
1004	NULL	M	B2	38965

SIMPLE QUERY USING SELECT COMMAND

- The SELECT command is used to pull information from a table. Syntax of SELECT command is :
`SELECT <column name>,<column name>
FROM <tablename>
WHERE <condition name> ;`

SELECTING ALL DATA

- In order to retrieve everything (all columns) from a table, SELECT command is used as :
`SELECT * FROM <tablename> ;`

e.g.

In order to retrieve everything from Employee table, we write SELECT command as :
EMPLOYEE

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000
1002	Akash	M	A1	35000
1004	NULL	M	B2	38965

SELECT * FROM Employee ;

SELECTING PARTICULAR COLUMNS

EMPLOYEE

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000
1002	Akash	M	A1	35000
1004	Neela	F	B2	38965
1005	Sunny	M	A2	30000
1006	Ruby	F	A1	45000
1009	Neema	F	A2	52000

- A particular column from a table can be selected by specifying column-names with SELECT command. E.g. in above table, if we want to select ECODE and ENAME column, then command is :

```
SELECT ECODE , ENAME
FROM EMPLOYEE ;
```

E.g.2 in order to select only ENAME, GRADE and GROSS column, the command is :

```
SELECT ENAME , GRADE ,
GROSS FROM EMPLOYEE ;
```

SELECTING PARTICULAR ROWS

We can select particular rows from a table by specifying a condition through **WHERE clause** along with SELECT statement. E.g. In employee table if we want to select rows where Gender is female, then command is :

```
SELECT * FROM EMPLOYEE
WHERE GENDER = 'F' ;
```

E.g.2. in order to select rows where salary is greater than 48000, then command is :

```
SELECT * FROM EMPLOYEE
WHERE GROSS > 48000 ;
```

ELIMINATING REDUNDANT DATA

The **DISTINCT** keyword eliminates duplicate rows from the results of a SELECT statement. For example ,

```
SELECT GENDER FROM EMPLOYEE ;
```

GENDER
M
M
F
M
F
F

```
SELECT DISTINCT(GENDER) FROM EMPLOYEE ;
```

DISTINCT(GENDER)
M
F

VIEWING STRUCTURE OF A TABLE

- If we want to know the structure of a table, we can use DESCRIBE or DESC command, as per following syntax :

```
DESCRIBE | DESC <tablename> ;
```

e.g. to view the structure of table **EMPLOYEE**, command is : **DESCRIBE EMPLOYEE ; OR DESC EMPLOYEE ;**

USING COLUMN ALIASES

- The columns that we select in a query can be given a different name, i.e. column alias name for output purpose.

Syntax :

```
SELECT <columnname> AS column alias , <columnname> AS column alias ....
FROM <tablename> ;
```

e.g. In output, suppose we want to display ECODE column as EMPLOYEE_CODE in output , then command is :

```
SELECT ECODE AS "EMPLOYEE_CODE"
FROM EMPLOYEE ;
```

CONDITION BASED ON A RANGE

- The **BETWEEN** operator defines a range of values that the column values must fall in to make the condition true. The range include both lower value and upper value.

e.g. to display ECODE, ENAME and GRADE of those employees whose salary is between 40000 and 50000, command is:

```
SELECT ECODE , ENAME ,GRADE
FROM EMPLOYEE
WHERE GROSS BETWEEN 40000 AND 50000 ;
```

Output will be :

ECODE	ENAME	GRADE
1001	Ravi	E4
1006	Ruby	A1

CONDITION BASED ON A LIST

- To specify a list of values, IN operator is used. The IN operator selects value that match any value in a given list of values. E.g.

```
SELECT * FROM EMPLOYEE
WHERE GRADE IN ('A1' , 'A2');
```

Output will be :

ECODE	ENAME	GENDER	GRADE	GROSS
1002	Akash	M	A1	35000
1006	Ruby	F	A1	45000
1005	Sunny	M	A2	30000
1009	Neema	F	A2	52000

- The **NOT IN** operator finds rows that do not match in the list. E.g.

```
SELECT * FROM EMPLOYEE
WHERE GRADE NOT IN ('A1' , 'A2');
```

Output will be :

ECODE	ENAME	GENDER	GRADE	GROSS
1001	Ravi	M	E4	50000
1004	Neela	F	B2	38965

CONDITION BASED ON PATTERN MATCHES

- LIKE operator is used for pattern matching in SQL. Patterns are described using two special wildcard characters:

1. percent(%) – The % character matches any substring.
2. underscore(_) – The _ character matches any character.

e.g. to display names of employee whose name starts with R in EMPLOYEE table, the command is :

```

SELECT ENAME
FROM EMPLOYEE
WHERE ENAME LIKE 'R%';

```

Output will be :

ENAME
Ravi
Ruby

e.g. to display details of employee whose second character in name is 'e'.

```

SELECT *
FROM EMPLOYEE
WHERE ENAME LIKE '_e%';

```

Output will be :

ECODE	ENAME	GENDER	GRADE	GROSS
1004	Neela	F	B2	38965
1009	Neema	F	A2	52000

e.g. to display details of employee whose name ends with 'y'.

```

SELECT *
FROM EMPLOYEE
WHERE ENAME LIKE '%y';

```

Output will be :

ECODE	ENAME	GENDER	GRADE	GROSS
1005	Sunny	M	A2	30000
1006	Ruby	F	A1	45000

SEARCHING FOR NULL

- The NULL value in a column can be searched for in a table using IS NULL in the WHERE clause. E.g. to list employee details whose salary contain NULL, we use the command :

```

SELECT *
FROM EMPLOYEE
WHERE GROSS IS NULL;

```

e.g.

STUDENT

Roll_No	Name	Marks
1	ARUN	NULL
2	RAVI	56
4	SANJAY	NULL

to display the names of those students whose marks is NULL, we use the command :

```

SELECT Name
FROM EMPLOYEE
WHERE Marks IS NULL ;

```

Output will be :

Name
ARUN
SANJAY

SORTING RESULTS

Whenever the SELECT query is executed , the resulting rows appear in a predecided order.The **ORDER BY clause** allow sorting of query result. The sorting can be done either in ascending or descending order, the default is ascending.

The **ORDER BY clause** is used as :

```
SELECT <column name> , <column name>....  
FROM <tablename>  
WHERE <condition>  
ORDER BY <column name> ;
```

e.g. to display the details of employees in EMPLOYEE table in alphabetical order, we use command :

```
SELECT *  
FROM EMPLOYEE  
ORDER BY ENAME ;
```

Output will be :

ECODE	ENAME	GENDER	GRADE	GROSS
1002	Akash	M	A1	35000
1004	Neela	F	B2	38965
1009	Neema	F	A2	52000
1001	Ravi	M	E4	50000
1006	Ruby	F	A1	45000
1005	Sunny	M	A2	30000

e.g. display list of employee in descending alphabetical order whose salary is greater than 40000.

```
SELECT ENAME  
FROM EMPLOYEE  
WHERE GROSS > 40000  
ORDER BY ENAME desc ;
```

Output will be :

ENAME
Ravi
Ruby
Neema

MODIFYING DATA IN TABLES

you can modify data in tables using UPDATE command of SQL. The UPDATE command specifies the rows to be changed using the WHERE clause, and the new data using the SET keyword. Syntax of update command is :

```
UPDATE <tablename>  
SET <columnname>=value , <columnname>=value  
WHERE <condition> ;
```

e.g. to change the salary of employee of those in EMPLOYEE table having employee code 1009 to 55000.

```
UPDATE EMPLOYEE  
SET GROSS = 55000  
WHERE ECODE = 1009 ;
```

UPDATING MORE THAN ONE COLUMNS

e.g. to update the salary to 58000 and grade to B2 for those employee whose employee code is 1001.

```
UPDATE EMPLOYEE  
SET GROSS = 58000, GRADE='B2'  
WHERE ECODE = 1009 ;
```

OTHER EXAMPLES

e.g.1. Increase the salary of each employee by 1000 in the EMPLOYEE table.

```
UPDATE EMPLOYEE  
SET GROSS = GROSS +100 ;
```

e.g.2. Double the salary of employees having grade as 'A1' or 'A2' .

```
UPDATE EMPLOYEE  
SET GROSS = GROSS * 2 ;  
WHERE GRADE='A1' OR GRADE='A2' ;
```

e.g.3. Change the grade to 'A2' for those employees whose employee code is 1004 and name is Neela.

```
UPDATE EMPLOYEE  
SET GRADE='A2'  
WHERE ECODE=1004 AND GRADE='NEELA' ;
```

DELETING DATA FROM TABLES

To delete some data from tables, DELETE command is used. **The DELETE command removes rows from a table.** The syntax of DELETE command is :

```
DELETE FROM <tablename>  
WHERE <condition> ;
```

For example, to remove the details of those employee from EMPLOYEE table whose grade is A1.

```
DELETE FROM EMPLOYEE  
WHERE GRADE ='A1' ;
```

TO DELETE ALL THE CONTENTS FROM A TABLE

```
DELETE FROM EMPLOYEE ;
```

So if we do not specify any condition with WHERE clause, then all the rows of the table will be deleted. Thus above line will delete all rows from employee table.

DROPPING TABLES

The DROP TABLE command lets you drop a table from the database. The **syntax of DROP TABLE** command is :

```
DROP TABLE <tablename> ;
```

e.g. to drop a table employee, we need to write :

```
DROP TABLE employee ;
```

Once this command is given, the table name is no longer recognized and no more commands can be given on that table.

After this command is executed, all the data in the table along with table structure will be deleted.

S.NO.	DELETE COMMAND	DROP TABLE COMMAND
1	It is a DML command.	It is a DDL Command.
2	This command is used to delete only rows of data from a table	This command is used to delete all the data of the table along with the structure of the table. The table is no longer recognized when this command gets executed.
3	Syntax of DELETE command is: DELETE FROM <tablename> WHERE <condition> ;	Syntax of DROP command is : DROP TABLE <tablename> ;

ALTER TABLE COMMAND

The ALTER TABLE command is used to change definitions of existing tables.(adding columns,deleting columns etc.). The ALTER TABLE command is used for :

1. adding columns to a table

2. Modifying column-definitions of a table.
3. Deleting columns of a table.
4. Adding constraints to table.
5. Enabling/Disabling constraints.

ADDING COLUMNS TO TABLE

To add a column to a table, ALTER TABLE command can be used as per following syntax:

```
ALTER TABLE <tablename>
ADD <Column name> <datatype> <constraint>;
```

e.g. to add a new column ADDRESS to the EMPLOYEE table, we can write command as :

```
ALTER TABLE EMPLOYEE
ADD ADDRESS VARCHAR(50);
```

A new column by the name ADDRESS will be added to the table, where each row will contain NULL value for the new column.

ECODE	ENAME	GENDER	GRADE	GROSS	ADDRESS
1001	Ravi	M	E4	50000	NULL
1002	Akash	M	A1	35000	NULL
1004	Neela	F	B2	38965	NULL
1005	Sunny	M	A2	30000	NULL
1006	Ruby	F	A1	45000	NULL
1009	Neema	F	A2	52000	NULL

However if you specify NOT NULL constraint while adding a new column, MySQL adds the new column with the default value of that datatype e.g. for INT type it will add 0 , for CHAR types, it will add a space, and so on.

e.g. Given a table namely Testt with the following data in it.

Col1	Col2
1	A
2	G

Now following commands are given for the table. Predict the table contents after each of the following statements:

- (i) ALTER TABLE testt ADD col3 INT ;
- (ii) ALTER TABLE testt ADD col4 INT NOT NULL ;
- (iii) ALTER TABLE testt ADD col5 CHAR(3) NOT NULL ;
- (iv) ALTER TABLE testt ADD col6 VARCHAR(3);

MODIFYING COLUMNS

Column name and data type of column can be changed as per following syntax :

```
ALTER TABLE <table name>
CHANGE <old column name> <new column name> <new datatype>;
```

If Only data type of column need to be changed, then

```
ALTER TABLE <table name>
MODIFY <column name> <new datatype>;
```

e.g.1. In table EMPLOYEE, change the column GROSS to SALARY.

```
ALTER TABLE EMPLOYEE  
CHANGE GROSS SALARY INTEGER;
```

e.g.2. In table EMPLOYEE , change the column ENAME to EM_NAME and data type from VARCHAR(20) to VARCHAR(30).

```
ALTER TABLE EMPLOYEE  
CHANGE ENAME EM_NAME VARCHAR(30);
```

e.g.3. In table EMPLOYEE , change the datatype of GRADE column from CHAR(2) to VARCHAR(2).

```
ALTER TABLE EMPLOYEE  
MODIFY GRADE VARCHAR(2);
```

DELETING COLUMNS

To delete a column from a table, the ALTER TABLE command takes the following form :

```
ALTER TABLE <table name>  
DROP <column name>;
```

e.g. to delete column GRADE from table EMPLOYEE, we will write :

```
ALTER TABLE EMPLOYEE  
DROP GRADE ;
```

ADDING/REMOVING CONSTRAINTS TO A TABLE

ALTER TABLE statement can be used to add constraints to your existing table by using it in following manner:

❖ **TO ADD PRIMARY KEY CONSTRAINT**
ALTER TABLE <table name>
ADD PRIMARY KEY (Column name);

e.g. to add PRIMARY KEY constraint on column ECODE of table EMPLOYEE , the command is :

```
ALTER TABLE EMPLOYEE  
ADD PRIMARY KEY (ECODE) ;
```

❖ **TO ADD FOREIGN KEY CONSTRAINT**

```
ALTER TABLE <table name>  
ADD FOREIGN KEY (Column name) REFERENCES Parent Table (Primary key of Parent Table);
```

REMOVING CONSTRAINTS

- To remove primary key constraint from a table, we use ALTER TABLE command
as : **ALTER TABLE <table name>**
DROP PRIMARY KEY ;
- To remove foreign key constraint from a table, we use ALTER TABLE command
as : **ALTER TABLE <table name>**
DROP FOREIGN KEY ;

ENABLING/DISABLING CONSTRAINTS

Only foreign key can be disabled/enabled in MySQL.

To disable foreign keys : **SET FOREIGN_KEY_CHECKS = 0 ;**

To enable foreign keys : **SET FOREIGN_KEY_CHECKS = 1 ;**

INTEGRITY CONSTRAINTS/CONSTRAINTS

- A constraint is a condition or check applicable on a field(column) or set of fields(columns).
- Common types of constraints include :

S.No.	Constraints	Description
1	NOT NULL	Ensures that a column cannot have NULL value
2	DEFAULT	Provides a default value for a column when none is specified
3	UNIQUE	Ensures that all values in a column are different
4	CHECK	Makes sure that all values in a column satisfy certain criteria
5	PRIMARY KEY	Used to uniquely identify a row in the table
6	FOREIGN KEY	Used to ensure referential integrity of the data

NOT NULL CONSTRAINT

By default, a column can hold NULL. If you do not want to allow NULL value in a column, then NOT NULL constraint must be applied on that column. E.g.

```
CREATE TABLE Customer
(   SID integer NOT NULL ,
    Last_Name varchar(30) NOT NULL ,
    First_Name varchar(30) );
```

Columns **SID** and **Last_Name** cannot include NULL, while **First_Name** can include NULL.

An attempt to execute the following SQL statement,

```
INSERT INTO Customer
VALUES (NULL , 'Kumar' , 'Ajay');
```

will result in an error because this will lead to column SID being NULL, which violates the NOT NULL constraint on that column.

DEFAULT CONSTARINT

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value. E.g.

```
CREATE TABLE Student
( Student_ID integer ,
  Name varchar(30) ,
  Score integer DEFAULT 80);
```

When following SQL statement is executed on table created above:

```
INSERT INTO Student
VALUES (10 , 'Ravi' );
```

Then table **Student** looks like the following:

Student_ID	Name	Score
10	Ravi	80

score field has got the default value

UNIQUE CONSTRAINT

- The UNIQUE constraint ensures that all values in a column are distinct. In other words, no two rows can hold the same value for a column with UNIQUE constraint.

e.g.

```
CREATE TABLE Customer
(
    SID integer Unique ,
    Last_Name varchar(30) ,
    First_Name varchar(30) );
```

Column SID has a unique constraint, and hence cannot include duplicate values. So, if the table already contains the following rows :

SID	Last_Name	First_Name
1	Kumar	Ravi
2	Sharma	Ajay
3	Devi	Raj

The executing the following SQL statement,

```
INSERT INTO Customer
VALUES ('3' , 'Cyrus' , 'Grace');
```

will result in an error because the value 3 already exist in the SID column, thus trying to insert another row with that value violates the UNIQUE constraint.

CHECK CONSTRAINT

- The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the table will only insert a new row or update an existing row if the new value satisfies the CHECK constraint.

e.g.

```
CREATE TABLE Customer
(
    SID integer CHECK (SID > 0),
    Last_Name varchar(30) ,
    First_Name varchar(30) );
```

So, attempting to execute the following statement :

```
INSERT INTO Customer
VALUES (-2 , 'Kapoor' , 'Raj');
```

will result in an error because the values for SID must be greater than 0.

PRIMARY KEY CONSTRAINT

- A primary key is used to identify each row in a table. A primary key can consist of one or more fields(column) on a table. When multiple fields are used as a primary key, they are called a **composite key**.
- You can define a primary key in CREATE TABLE command through keywords PRIMARY KEY. e.g.

```
CREATE TABLE Customer
(
    SID integer NOT NULL PRIMARY KEY,
    Last_Name varchar(30) ,
    First_Name varchar(30) );
```

Or

```
CREATE TABLE Customer
(
    SID integer,
    Last_Name varchar(30) ,
    First_Name varchar(30),
    PRIMARY KEY (SID) ;
)
```

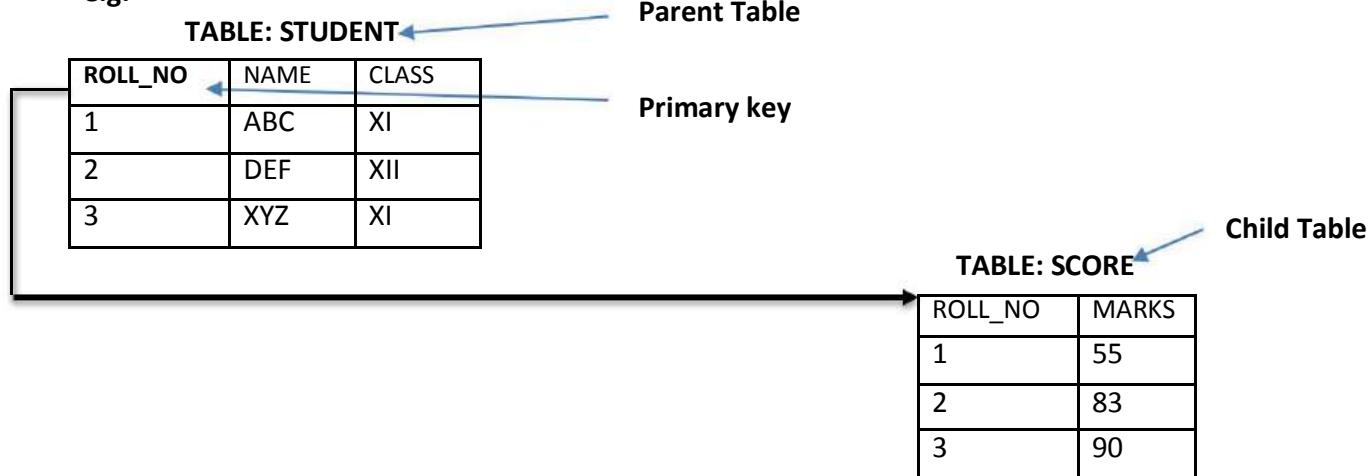
- The latter way is useful if you want to specify a composite primary key, e.g.

```
CREATE TABLE Customer
(
    Branch integer NOT NULL,
    SID integer NOT NULL ,
    Last_Name varchar(30) ,
    First_Name varchar(30),
    PRIMARY KEY (Branch , SID) );
)
```

FOREIGN KEY CONSTRAINT

- Foreign key is a non key column of a table (**child table**) that draws its values from **primary key** of another table(**parent table**).
- The table in which a foreign key is defined is called a **referencing table or child table**. A table to which a foreign key points is called **referenced table or parent table**.

e.g.



Here column Roll_No is a foreign key in table SCORE(Child Table) and it is drawing its values from Primary key (ROLL_NO) of STUDENT table.(Parent Key).

```
CREATE TABLE STUDENT
(
    ROLL_NO integer NOT NULL PRIMARY KEY ,
    NAME VARCHAR(30) ,
    CLASS VARCHAR(3) );
)
```

```
CREATE TABLE SCORE
(
    ROLL_NO integer ,
    MARKS integer ,
    FOREIGN KEY(ROLL_NO) REFERENCES STUDENT(ROLL_NO) );
)
```

* *Foreign key is always defined in the child table.*

Syntax for using foreign key

FOREIGN KEY(column name) REFERENCES Parent_Table(PK of Parent Table);

REFERENCING ACTIONS

Referencing action with ON DELETE clause determines what to do in case of a DELETE occurs in the parent table.
Referencing action with ON UPDATE clause determines what to do in case of a UPDATE occurs in the parent table.

Actions:

1. **CASCADE** : This action states that if a DELETE or UPDATE operation affects a row from the parent table, then automatically delete or update the matching rows in the child table i.e., cascade the action to child table.
2. **SET NULL** : This action states that if a DELETE or UPDATE operation affects a row from the parent table, then set the foreign key column in the child table to NULL.
3. **NO ACTION** : Any attempt for DELETE or UPDATE in parent table is not allowed.
4. **RESTRICT** : This action rejects the DELETE or UPDATE operation for the parent table.

Q: Create two tables

Customer(customer_id, name)

Customer_sales(transaction_id, amount, **customer_id**)

Underlined columns indicate primary keys and bold column names indicate foreign key.

Make sure that no action should take place in case of a DELETE or UPDATE in the parent table.

Sol : CREATE TABLE Customer (

```
customer_id int Not Null Primary Key ,  
name varchar(30) );
```

CREATE TABLE Customer_sales (

```
transaction_id Not Null Primary Key ,  
amount int ,  
customer_id int ,  
FOREIGN KEY(customer_id) REFERENCES Customer (customer_id)  
ON DELETE NO ACTION  
ON UPDATE NO ACTION );
```

Q: Distinguish between a Primary Key and a Unique key in a table.

S.NO.	PRIMARY KEY	UNIQUE KEY
1.	Column having Primary key can't contain NULL value	Column having Unique Key can contain NULL value
2.	There can be only one primary key in Table.	Many columns can be defined as Unique key

Q: Distinguish between ALTER Command and UPDATE command of SQL.

S.NO.	ALTER COMMAND	UPDATE COMMAND
1.	It is a DDL Command	It is a DML command
2.	It is used to change the definition of existing table, i.e. adding column, deleting column, etc.	It is used to modify the data values present in the rows of the table.
3.	Syntax for adding column in a table: ALTER TABLE <tablename> ADD <Column name><Datatype> ;	Syntax for using UPDATE command: UPDATE <Tablename> SET <Columnname>=value WHERE <Condition> ;

AGGREGATE / GROUP FUNCTIONS

Aggregate / Group functions work upon groups of rows , rather than on single row, and return one single output. Different aggregate functions are : COUNT() , AVG() , MIN() , MAX() , SUM ()

Table : EMPL

EMPNO	ENAME	JOB	SAL	DEPTNO
8369	SMITH	CLERK	2985	10
8499	ANYA	SALESMAN	9870	20
8566	AMIR	SALESMAN	8760	30
8698	BINA	MANAGER	5643	20
8912	SUR	NULL	3000	10

1. AVG()

This function computes the average of given data. e.g. SELECT AVG(SAL)

```
FROM EMPL ;
```

Output

AVG(SAL)
6051.6

2. COUNT()

This function counts the number of rows in a given column.

If you specify the COLUMN name in parenthesis of function, then this function returns rows where COLUMN is not null.

If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

e.g. SELECT COUNT(*)

```
FROM EMPL ;
```

Output

COUNT(*)
5

e.g.2 SELECT COUNT(JOB)

```
FROM EMPL ;
```

Output

COUNT(JOB)
4

3. MAX()

This function returns the maximum value from a given column or expression.

e.g. SELECT MAX(SAL)

```
FROM EMPL ;
```

Output

MAX(SAL)
9870

4. MIN()

This function returns the minimum value from a given column or expression.

e.g. `SELECT MIN(SAL)
 FROM EMPL ;`

Output

MIN(SAL)
2985

5. SUM()

This function returns the sum of values in given column or expression.

e.g. `SELECT SUM(SAL)
 FROM EMPL ;`

Output

SUM(SAL)
30258

GROUPING RESULT – GROUP BY

The GROUP BY clause combines all those records(row) that have identical values in a particular field(column) or a group of fields(columns).

GROUPING can be done by a column name, or with aggregate functions in which case the aggregate produces a value for each group.

Table : EMPL

EMPNO	ENAME	JOB	SAL	DEPTNO
8369	SMITH	CLERK	2985	10
8499	ANYA	SALESMAN	9870	20
8566	AMIR	SALESMAN	8760	30
8698	BINA	MANAGER	5643	20

e.g. Calculate the number of employees in each grade.

```
SELECT JOB, COUNT(*)  
  FROM EMPL  
 GROUP BY JOB ;
```

Output

JOB	COUNT(*)
CLERK	1
SALESMAN	2
MANAGER	1

e.g.2. Calculate the sum of salary for each department.

```
SELECT DEPTNO , SUM(SAL)  
  FROM EMPL  
 GROUP BY DEPTNO ;
```

Output

DEPTNO	SUM(SAL)
10	2985
20	15513
30	8760

e.g.3. find the average salary of each department.

Sol:

*** One thing that you should keep in mind is that while grouping , you should include only those values in the SELECT list that either have the same value for a group or contain a group(aggregate) function. Like in e.g. 2 given above, DEPTNO column has one(same) value for a group and the other expression SUM(SAL) contains a group function.*

NESTED GROUP

- To create a group within a group i.e., nested group, you need to specify multiple fields in the GROUP BY expression. e.g. To group records **job wise** within **Deptno wise**, you need to issue a query statement like :

```
SELECT DEPTNO ,JOB , COUNT(EMPNO)
  FROM EMPL
 GROUP BY DEPTNO ,JOB ;
```

Output

DEPTNO	JOB	COUNT(EMPNO)
10	CLERK	1
20	SALESMAN	1
20	MANAGER	1
30	SALESMAN	1

PLACING CONDITION ON GROUPS – HAVING CLAUSE

- The **HAVING clause places conditions on groups** in contrast to WHERE clause that places condition on individual rows. While **WHERE conditions cannot include aggregate functions, HAVING conditions can do so**.
- e.g. To display the jobs where the number of employees is less than 2,

```
SELECT JOB, COUNT(*)
  FROM EMPL
 GROUP BY JOB
 HAVING COUNT(*) < 2 ;
```

Output

JOB	COUNT(*)
CLERK	1
MANAGER	1

DATABASE TRANSACTIONS

TRANSACTION

A Transaction is a logical unit of work that must succeed or fail in its entirety. This statement means that a transaction may involve many sub steps, which should either all be carried out successfully or all be ignored if some failure occurs. A Transaction is an atomic operation which may not be divided into smaller operations.

Example of a Transaction

Begin transaction

```
Get balance from account X
Calculate new balance as X – 1000
Store new balance into database file
Get balance from account Y
Calculate new balance as Y + 1000
Store new balance into database file
```

End transaction

TRANSACTION PROPERTIES (ACID PROPERTIES)

1. **ATOMICITY**(All or None Concept) – This property ensures that either all operations of the transaction are carried out or none are.
2. **CONSISTENCY** – This property implies that if the database was in a consistent state before the start of transaction execution, then upon termination of transaction, the database will also be in a consistent state.
3. **ISOLATION** – This property implies that each transaction is unaware of other transactions executing concurrently in the system.
4. **DURABILITY** – This property of a transaction ensures that after the successful completion of a transaction, the changes made by it to the database persist, even if there are system failures.

TRANSACTION CONTROL COMMANDS (TCL)

The TCL of MySQL consists of following commands :

1. BEGIN or START TRANSACTION – marks the beginning of a transaction.
2. COMMIT – Ends the current transaction by saving database changes and starts a new transaction.
3. ROLLBACK – Ends the current transaction by discarding database changes and starts a new transaction.
4. SAVEPOINT – Define breakpoints for the transaction to allow partial rollbacks.
5. SET AUTOCOMMIT – Enables or disables the default auto commit mode

SQL Commands | DDL, DQL, DML, DCL and TCL Commands

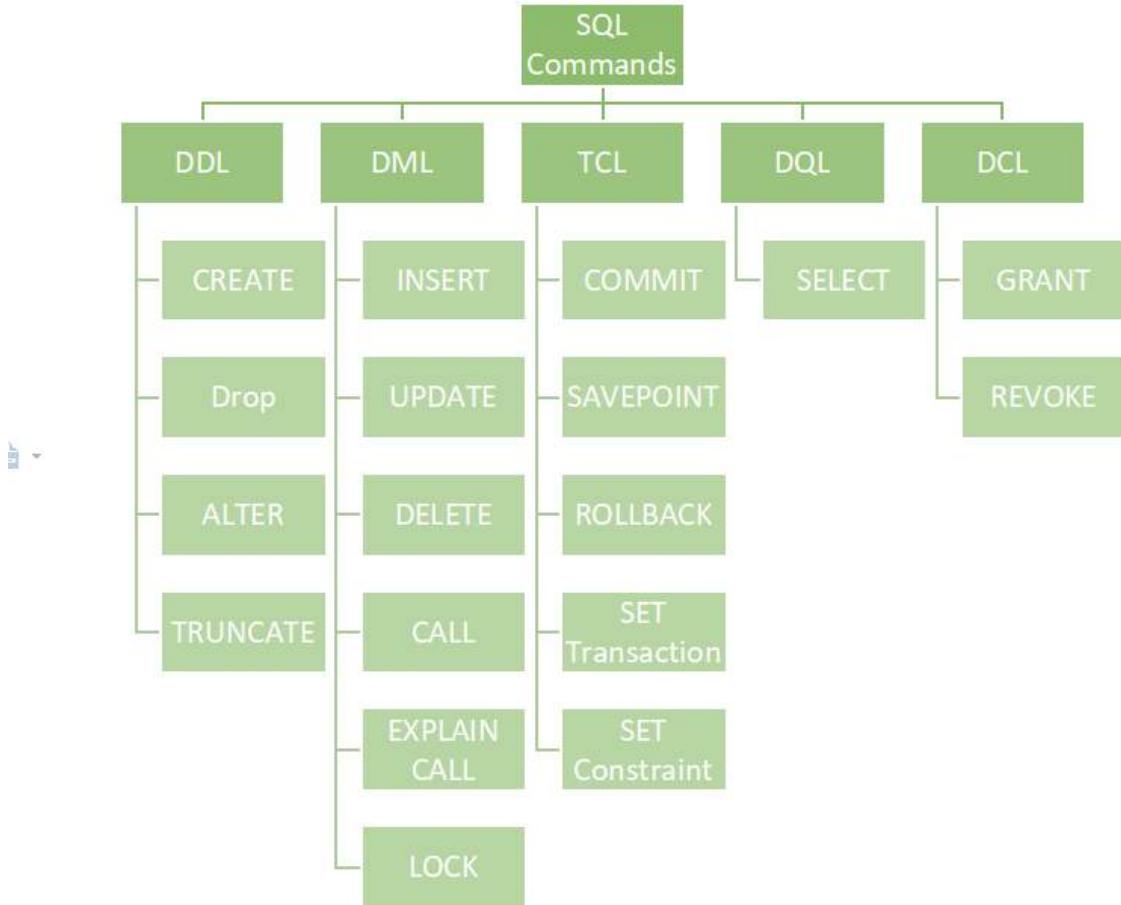
SQL commands are very used to interact with the database. These commands allow users to perform various actions on a database. This article will teach us about **SQL commands** or **SQL sublanguage commands** like **DDL**, **DQL**, **DML**, **DCL**, and **TCL**.

Structured Query Language (SQL) is the database language by which we can perform certain operations on the existing database, and we can also 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:

1. **DDL** – Data Definition Language
2. **DQL** – Data Query Language
3. **DML** – Data Manipulation Language
4. **DCL** – Data Control Language
5. **TCL** – Transaction Control Language



DDL (Data Definition Language)

[DDL](#) or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

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

Some DDL commands and their syntax are:

Command	Description	Syntax
<u>CREATE</u>	Create database or its objects (table, index, function, views, store procedure, and triggers)	CREATE TABLE table_name (column1 data_type, column2 data_type, ...);

Command	Description	Syntax
<u>DROP</u>	Delete objects from the database	DROP TABLE table_name;
<u>ALTER</u>	Alter the structure of the database	ALTER TABLE table_name ADD COLUMN column_name data_type;
<u>TRUNCATE</u>	Remove all records from a table, including all spaces allocated for the records are removed	TRUNCATE TABLE table_name;
<u>COMMENT</u>	Add comments to the data dictionary	COMMENT 'comment_text' ON TABLE table_name;
<u>RENAME</u>	Rename an object existing in the database	RENAME TABLE old_table_name TO new_table_name;

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. a front-end.

DQL Command

There is only one DQL command in SQL i.e.

Command	Description	Syntax
<u>SELECT</u>	It is used to retrieve data from the database	SELECT column1, column2, ...FROM table_name WHERE condition;

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

Some DML commands and their syntax are:

Command	Description	Syntax
<u>INSERT</u>	Insert data into a table	INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
<u>UPDATE</u>	Update existing data within a table	UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;
<u>DELETE</u>	Delete records from a database table	DELETE FROM table_name WHERE condition;
<u>LOCK</u>	Table control concurrency	LOCK TABLE table_name IN lock_mode;
CALL	Call a PL/SQL or JAVA subprogram	CALL procedure_name(arguments);
EXPLAIN PLAN	Describe the access path to data	EXPLAIN PLAN FOR SELECT * FROM table_name;

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:

Two important DCL commands and their syntax are:

Command	Description	Syntax
<u>GRANT</u>	Assigns new privileges to a user account, allowing access to	GRANT privilege_type [(column_list)] ON [object_type]

Command	Description	Syntax
	specific database objects, actions, or functions.	object_name TO user [WITH GRANT OPTION];
<u>REVOKE</u>	Removes previously granted privileges from a user account, taking away their access to certain database objects or actions.	REVOKE [GRANT OPTION FOR] privilege_type [(column_list)] ON [object_type] object_name FROM user [CASCADE];

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. You can explore more about transactions [here](#). Hence, the following TCL commands are used to control the execution of a transaction:

List of TCL Commands

Some TCL commands and their syntax are:

Command	Description	Syntax
<u>BEGIN TRANSACTION</u>	Starts a new transaction	BEGIN TRANSACTION [transaction_name];
<u>COMMIT</u>	Saves all changes made during the transaction	COMMIT;
<u>ROLLBACK</u>	Undoes all changes made during the transaction	ROLLBACK;
<u>SAVEPOINT</u>	Creates a savepoint within the current transaction	SAVEPOINT savepoint_name;

Important SQL Commands

Some of the most important SQL commands are:

1. **SELECT:** Used to retrieve data from a database.

2. **INSERT:** Used to add new data to a database.
3. **UPDATE:** Used to modify existing data in a database.
4. **DELETE:** Used to remove data from a database.
5. **CREATE TABLE:** Used to create a new table in a database.
6. **ALTER TABLE:** Used to modify the structure of an existing table.
7. **DROP TABLE:** Used to delete an entire table from a database.
8. **WHERE:** Used to filter rows based on a specified condition.
9. **ORDER BY:** Used to sort the result set in ascending or descending order.
10. **JOIN:** Used to combine rows from two or more tables based on a related column between them.

SQL Commands With Examples

The examples demonstrates how to use an SQL command. Here is the list of popular SQL commands with Examples.

SQL Command	Example
SELECT	SELECT * FROM employees;
INSERT	INSERT INTO employees (first_name, last_name, email) VALUES ('John', 'Doe', 'john.doe@example.com');
UPDATE	UPDATE employees SET email = 'jane.doe@example.com' WHERE first_name = 'Jane' AND last_name = 'Doe';
DELETE	DELETE FROM employees WHERE employee_id = 123;
CREATE TABLE	CREATE TABLE employees (employee_id INT PRIMARY KEY, first_name VARCHAR(50), last_name VARCHAR(50));
ALTER TABLE	ALTER TABLE employees ADD COLUMN phone VARCHAR(20);
DROP TABLE	DROP TABLE employees;

SQL Command	Example
WHERE	SELECT * FROM employees WHERE department = 'Sales';
ORDER BY	SELECT * FROM employees ORDER BY hire_date DESC;
JOIN	SELECT e.first_name, e.last_name, d.department_name FROM employees e JOIN departments d ON e.department_id = d.department_id;

These are common examples of some important SQL commands. The examples provide better understanding of the SQL commands and teaches correct way to use them.

Grant and Revoke in SQL Examples

Let's look at some examples highlighting the syntax usage of Grant and Revoke commands, the types of privileges applicable, and the database objects involved.

Common Scenarios and Use Cases

Understanding the appropriate syntax for Grant and Revoke commands is essential for implementing access control effectively.

Granting a user SELECT and UPDATE permissions on a specific table:

```
GRANT SELECT, UPDATE ON table_name TO user_name;
```

In this example, the Grant command is used to allow the specified user to read (SELECT) and modify (UPDATE) data from the designated table.

Revoking DELETE permission from a user for a specific table:

```
REVOKE DELETE ON table_name FROM user_name;
```

Here, the Revoke command is used to remove the user's ability to delete (DELETE) data from the specified table.

The ALTER privilege is a common permission granted to allow a user to alter the structure of a table, such as adding or removing columns. However, it's essential to manage this privilege carefully, as altering table structures can lead to unintended consequences.

Granting a role the permission to execute a specific stored procedure:

```
GRANT EXECUTE ON stored_procedure_name TO role_name;
```

This example demonstrates granting a role the EXECUTE permission, allowing users with that role to execute the specified stored procedure.

Revoking all permissions from a user on a specific view:

```
REVOKE ALL ON view_name FROM user_name;
```

Here, the Revoke command is used with the ALL keyword to remove all permissions from the specified user on the designated view.

- **Best practices:** It's essential to restrict user access only to the specific objects and permissions required for their role. This reduces the potential for data breaches or accidental data corruption.
- **When to use Grant and Revoke:** Use these commands when onboarding new users or roles, changing job responsibilities, or removing permissions for users who no longer require access.
- **Objects and permissions:** Ensure that the Grant and Revoke commands are being executed for the correct objects (e.g., tables, views, stored procedures) and the appropriate permissions (e.g., SELECT, DELETE, ALTER).

Understanding the proper use of Grant and Revoke syntax for managing user access and roles is crucial for maintaining a secure and well-organized database environment. By executing these commands effectively and adhering to best practices, you can guarantee that your database remains protected while enabling users to access and manage the data they require.

Grant and Revoke in SQL - Key takeaways

- Grant and Revoke in SQL: Essential commands for managing database access controls, maintaining security, and assigning or removing user privileges.
- Two categories of privileges: System Privileges (allow actions affecting entire database systems) and Object Privileges (determine actions users can perform on specific objects within the database).

- Role-Based Access Control (RBAC): A widely used model for managing user permissions in databases, assigning permissions to roles and users to specific roles for simplified administration.
- Grant and Revoke in SQL Server: Applying the same core concepts for controlling access to database objects, defining roles, and managing both system and object privileges within Microsoft SQL Server.
- Grant and Revoke syntax: Follows specific rules when assigning or removing permissions to users or roles for various types of privileges and database objects (e.g., tables, views, stored procedures).

SQL Commands | DDL, DQL, DML, DCL and TCL Commands

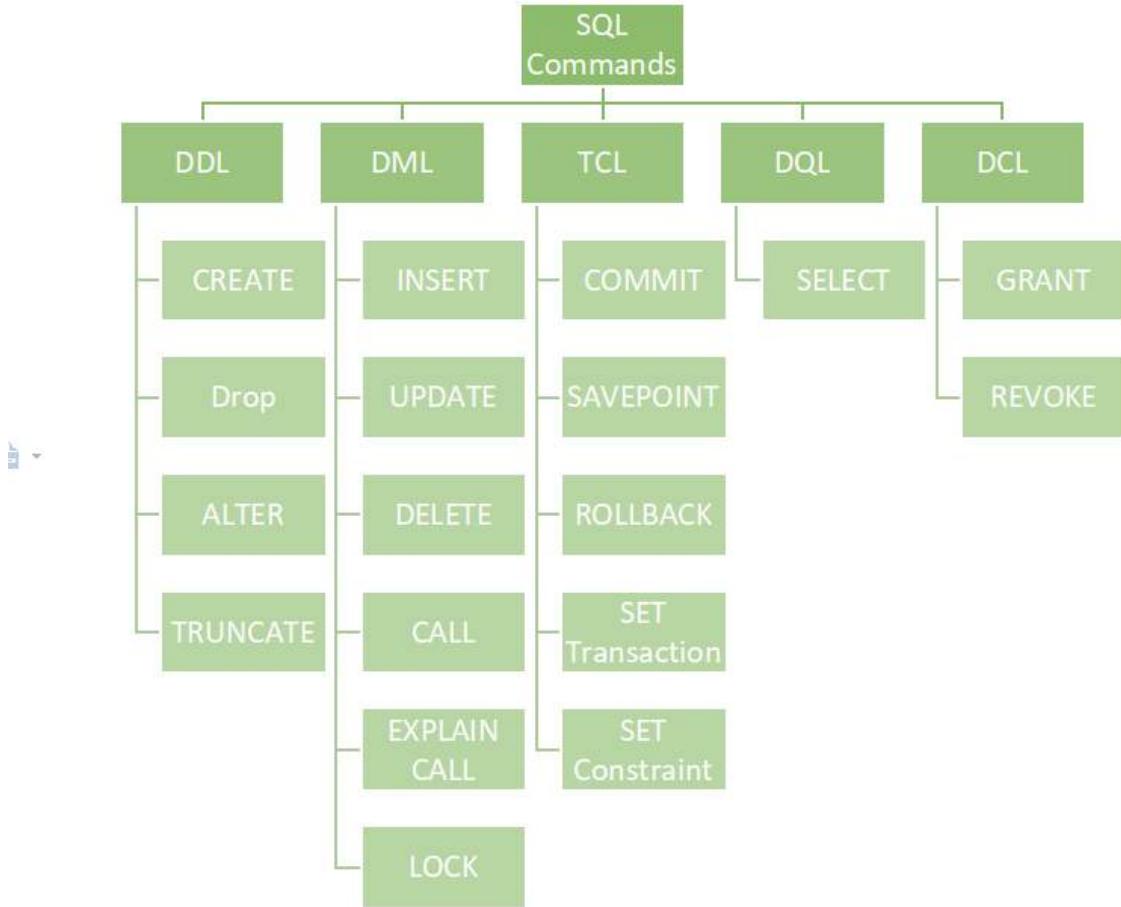
SQL commands are very used to interact with the database. These commands allow users to perform various actions on a database. This article will teach us about **SQL commands** or **SQL sublanguage commands** like **DDL**, **DQL**, **DML**, **DCL**, and **TCL**.

Structured Query Language (SQL) is the database language by which we can perform certain operations on the existing database, and we can also 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:

1. **DDL** – Data Definition Language
2. **DQL** – Data Query Language
3. **DML** – Data Manipulation Language
4. **DCL** – Data Control Language
5. **TCL** – Transaction Control Language



DDL (Data Definition Language)

[DDL](#) or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

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

Some DDL commands and their syntax are:

Command	Description	Syntax
<u>CREATE</u>	Create database or its objects (table, index, function, views, store procedure, and triggers)	CREATE TABLE table_name (column1 data_type, column2 data_type, ...);

Command	Description	Syntax
<u>DROP</u>	Delete objects from the database	DROP TABLE table_name;
<u>ALTER</u>	Alter the structure of the database	ALTER TABLE table_name ADD COLUMN column_name data_type;
<u>TRUNCATE</u>	Remove all records from a table, including all spaces allocated for the records are removed	TRUNCATE TABLE table_name;
<u>COMMENT</u>	Add comments to the data dictionary	COMMENT 'comment_text' ON TABLE table_name;
<u>RENAME</u>	Rename an object existing in the database	RENAME TABLE old_table_name TO new_table_name;

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. a front-end.

DQL Command

There is only one DQL command in SQL i.e.

Command	Description	Syntax
<u>SELECT</u>	It is used to retrieve data from the database	SELECT column1, column2, ...FROM table_name WHERE condition;

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

Some DML commands and their syntax are:

Command	Description	Syntax
<u>INSERT</u>	Insert data into a table	INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
<u>UPDATE</u>	Update existing data within a table	UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;
<u>DELETE</u>	Delete records from a database table	DELETE FROM table_name WHERE condition;
<u>LOCK</u>	Table control concurrency	LOCK TABLE table_name IN lock_mode;
CALL	Call a PL/SQL or JAVA subprogram	CALL procedure_name(arguments);
EXPLAIN PLAN	Describe the access path to data	EXPLAIN PLAN FOR SELECT * FROM table_name;

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:

Two important DCL commands and their syntax are:

Command	Description	Syntax
<u>GRANT</u>	Assigns new privileges to a user account, allowing access to	GRANT privilege_type [(column_list)] ON [object_type]

Command	Description	Syntax
	specific database objects, actions, or functions.	object_name TO user [WITH GRANT OPTION];
<u>REVOKE</u>	Removes previously granted privileges from a user account, taking away their access to certain database objects or actions.	REVOKE [GRANT OPTION FOR] privilege_type [(column_list)] ON [object_type] object_name FROM user [CASCADE];

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. You can explore more about transactions [here](#). Hence, the following TCL commands are used to control the execution of a transaction:

List of TCL Commands

Some TCL commands and their syntax are:

Command	Description	Syntax
<u>BEGIN TRANSACTION</u>	Starts a new transaction	BEGIN TRANSACTION [transaction_name];
<u>COMMIT</u>	Saves all changes made during the transaction	COMMIT;
<u>ROLLBACK</u>	Undoes all changes made during the transaction	ROLLBACK;
<u>SAVEPOINT</u>	Creates a savepoint within the current transaction	SAVEPOINT savepoint_name;

Important SQL Commands

Some of the most important SQL commands are:

1. **SELECT:** Used to retrieve data from a database.

2. **INSERT:** Used to add new data to a database.
3. **UPDATE:** Used to modify existing data in a database.
4. **DELETE:** Used to remove data from a database.
5. **CREATE TABLE:** Used to create a new table in a database.
6. **ALTER TABLE:** Used to modify the structure of an existing table.
7. **DROP TABLE:** Used to delete an entire table from a database.
8. **WHERE:** Used to filter rows based on a specified condition.
9. **ORDER BY:** Used to sort the result set in ascending or descending order.
10. **JOIN:** Used to combine rows from two or more tables based on a related column between them.

SQL Commands With Examples

The examples demonstrates how to use an SQL command. Here is the list of popular SQL commands with Examples.

SQL Command	Example
SELECT	SELECT * FROM employees;
INSERT	INSERT INTO employees (first_name, last_name, email) VALUES ('John', 'Doe', 'john.doe@example.com');
UPDATE	UPDATE employees SET email = 'jane.doe@example.com' WHERE first_name = 'Jane' AND last_name = 'Doe';
DELETE	DELETE FROM employees WHERE employee_id = 123;
CREATE TABLE	CREATE TABLE employees (employee_id INT PRIMARY KEY, first_name VARCHAR(50), last_name VARCHAR(50));
ALTER TABLE	ALTER TABLE employees ADD COLUMN phone VARCHAR(20);
DROP TABLE	DROP TABLE employees;

SQL Command	Example
WHERE	SELECT * FROM employees WHERE department = 'Sales';
ORDER BY	SELECT * FROM employees ORDER BY hire_date DESC;
JOIN	SELECT e.first_name, e.last_name, d.department_name FROM employees e JOIN departments d ON e.department_id = d.department_id;

These are common examples of some important SQL commands. The examples provide better understanding of the SQL commands and teaches correct way to use them.

SET TRANSACTION

Purpose

Use the SET TRANSACTION statement to establish the current transaction as read-only or read/write, establish its isolation level, assign it to a specified rollback segment, or assign a name to the transaction.

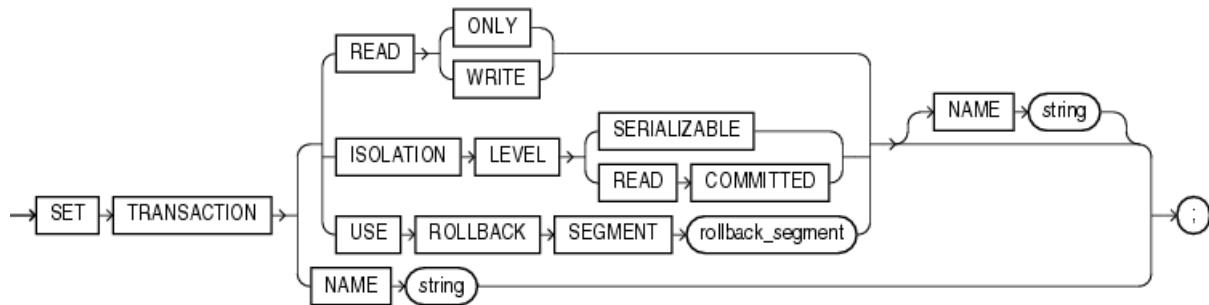
A transaction implicitly begins with any operation that obtains a TX lock:

- When a statement that modifies data is issued
- When a SELECT ... FOR UPDATE statement is issued
- When a transaction is explicitly started with a SET TRANSACTION statement or the DBMS_TRANSACTION package

Issuing either a COMMIT or ROLLBACK statement explicitly ends the current transaction. The operations performed by a SET TRANSACTION statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a COMMIT or ROLLBACK statement. Oracle Database implicitly commits the current transaction before and after executing a data definition language (DDL) statement.

Syntax

set_transaction::=



Semantics

READ ONLY

The READ ONLY clause establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**.

All subsequent queries in that transaction see only changes that were committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

This clause is not supported for the user SYS. Queries by SYS will return changes made during the transaction even if SYS has set the transaction to be READ ONLY.

Restriction on Read-only Transactions

Only the following statements are permitted in a read-only transaction:

- Subqueries—SELECT statements without the *for_update_clause*
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

READ WRITE

Specify READ WRITE to establish the current transaction as a read/write transaction. This clause establishes **statement-level read consistency**, which is the default.

Restriction on Read/Write Transactions

You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL Clause

- The SERIALIZABLE setting specifies serializable transaction isolation mode as defined in the SQL standard. If a serializable transaction contains data

manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.

- The READ COMMITTED setting is the default Oracle Database transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT Clause

Note:

This clause is relevant and valid only if you are using rollback segments for undo. Oracle strongly recommends that you use automatic undo management to handle undo space. If you follow this recommendation and run your database in automatic undo mode, then Oracle Database ignores this clause.

Specify USE ROLLBACK SEGMENT to assign the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read/write transaction.

Parallel DML requires more than one rollback segment. Therefore, if your transaction contains parallel DML operations, then the database ignores this clause.

NAME Clause

Use the NAME clause to assign a name to the current transaction. This clause is especially useful in distributed database environments when you must identify and resolve in-doubt transactions. The *string* value is limited to 255 bytes.

If you specify a name for a distributed transaction, then when the transaction commits, the name becomes the commit comment, overriding any comment specified explicitly in the COMMIT statement.

Examples

Setting Transactions: Examples

The following statements could be run at midnight of the last day of every month to count the products and quantities on hand in the Toronto warehouse in the sample Order Entry (oe) schema. This report would not be affected by any other user who might be adding or removing inventory to a different warehouse.

```
COMMIT;
```

```
SET TRANSACTION READ ONLY NAME 'Toronto';
```

```
SELECT product_id, quantity_on_hand FROM inventories
  WHERE warehouse_id = 5
  ORDER BY product_id;
```

```
COMMIT;
```

The first COMMIT statement ensures that SET TRANSACTION is the first statement in the transaction. The last COMMIT statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

Locks in MySQL

Locks in MySQL are mechanisms that manage the concurrent access of multiple users to database resources. MySQL supports several types of locks to ensure data integrity and consistency. Here's an overview of the main types of locks in MySQL, along with examples:

1. Table Locks

Table locks prevent different users from performing operations that could interfere with each other on the same table. MySQL provides two types of table locks:

- **READ LOCK:** Multiple sessions can acquire read locks on the same table simultaneously, but no session can acquire a write lock on it until all read locks are released.
- **WRITE LOCK:** Only one session can acquire a write lock on a table, and no other session can read or write to it until the write lock is released.

Example:

```
-- Acquiring a read lock
LOCK TABLES my_table READ;

-- Another session trying to acquire a write lock will be blocked
LOCK TABLES my_table WRITE;

-- Releasing the lock
UNLOCK TABLES;
```

2. Row Locks

Row-level locking is available with InnoDB storage engine. It allows multiple transactions to acquire locks on different rows simultaneously, improving concurrency.

Example:

```
-- Transaction 1
START TRANSACTION;
SELECT * FROM my_table WHERE id = 1 FOR UPDATE; -- Acquires a lock on the
row with id=1

-- Transaction 2
START TRANSACTION;
SELECT * FROM my_table WHERE id = 2 FOR UPDATE; -- Acquires a lock on the
row with id=2
```

3. Named Locks

Named locks are user-defined locks that allow you to lock arbitrary names. They are useful for application-level locking.

Example:

```
-- Acquiring a named lock
SELECT GET_LOCK('my_lock', 10); -- Waits up to 10 seconds to acquire the
lock

-- Performing operations while holding the lock
-- ...

-- Releasing the named lock
SELECT RELEASE_LOCK('my_lock');
```

4. Metadata Locks

Metadata locks are automatically acquired by MySQL to manage access to database objects during DDL operations like ALTER TABLE, DROP TABLE, etc.

Example:

```
-- Session 1
START TRANSACTION;
SELECT * FROM my_table; -- Acquires a metadata lock

-- Session 2
ALTER TABLE my_table ADD COLUMN new_column INT; -- Blocked until Session 1
transaction completes

-- Session 1
COMMIT; -- Releases the metadata lock, allowing Session 2 to proceed
```

Practical Example with Transactions

```
-- Transaction 1: Transfer money from account 1 to account 2
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1 FOR UPDATE; -- Lock row
for account 1
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
SELECT balance FROM accounts WHERE account_id = 2 FOR UPDATE; -- Lock row
for account 2
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;

-- Transaction 2: Check balance of account 1 and account 2
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1; -- No lock needed if
just reading
SELECT balance FROM accounts WHERE account_id = 2; -- No lock needed if
just reading
COMMIT;
```

In this example, `FOR UPDATE` is used to lock specific rows during the transaction, ensuring that no other transactions can modify these rows until the current transaction is complete.

These examples demonstrate how locks can be used in MySQL to manage concurrent access to database resources, ensuring data consistency and integrity.

SQL Operators

An operator is a reserved word or a character that is used to query our database in a SQL expression. To query a database using operators, we use a WHERE clause.

Operators are necessary to define a condition in SQL, as they act as a connector between two or more conditions. The operator manipulates the data and gives the result based on the operator's functionality.

Every database administrator and user uses SQL queries for manipulating and accessing the data of database tables and views.

The manipulation and retrieving of the data are performed with the help of reserved words and characters, which are used to perform arithmetic operations, logical operations, comparison operations, compound operations, etc.

Types of Operator

SQL operators are categorized in the following categories:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators
4. SQL Set Operators
5. SQL Bit-wise Operators
6. SQL Unary Operators

SQL Arithmetic Operators

The **Arithmetic Operators** perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.

Following are the various arithmetic operators performed on the SQL data:

1. SQL Addition Operator (+)
2. SQL Subtraction Operator (-)

3. SQL Multiplication Operator (+)
4. SQL Division Operator (-)
5. SQL Modulus Operator (%)

SQL Addition Operator (+)

The **Addition Operator** in SQL performs the addition on the numerical data of the database table. In SQL, we can easily add the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also add the numbers to the existing numbers of the specific column.

Syntax of SQL Addition Operator:

1. `SELECT operand1 + operand2;`

Let's understand the below example which explains how to execute Addition Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_Monthlybonus**.

Emp Id	Emp Name	Emp Salary	Emp Monthlybonus
101	Tushar	25000	4000
102	Anuj	30000	200

- o Suppose, we want to add **20,000** to the salary of each employee specified in the table. Then, we have to write the following query in the SQL:

1. `SELECT Emp_Salary + 20000 as Emp_New_Salary FROM Employee_details;`

In this query, we have performed the SQL addition operation on the single column of the given table.

- o Suppose, we want to add the Salary and monthly bonus columns of the above table, then we have to write the following query in SQL:

1. `SELECT Emp_Salary + Emp_Monthlybonus as Emp_Total_Salary FROM Employee_details;`

In this query, we have added two columns with each other of the above table.

SQL Subtraction Operator (-)

The Subtraction Operator in SQL performs the subtraction on the numerical data of the database table. In SQL, we can easily subtract the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also subtract the number from the existing number of the specific table column.

Syntax of SQL Subtraction Operator:

1. `SELECT operand1 - operand2;`

Let's understand the below example which explains how to execute Subtraction Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_Monthlybonus**.

Emp Id	Emp Name	Emp Salary	Pe
201	Abhay	25000	20
202	Sumit	30000	50

- o Suppose we want to subtract 5,000 from the salary of each employee given in the **Employee_details** table. Then, we have to write the following query in the SQL:
 1. `SELECT Emp_Salary - 5000 as Emp_New_Salary FROM Employee_details;`

In this query, we have performed the SQL subtraction operation on the single column of the given table.

- o If we want to subtract the penalty from the salary of each employee, then we have to write the following query in SQL:
 1. `SELECT Emp_Salary - Penalty as Emp_Total_Salary FROM Employee_details;`

SQL Multiplication Operator (*)

The Multiplication Operator in SQL performs the Multiplication on the numerical data of the database table. In SQL, we can easily multiply the numerical values of two

columns of the same table by specifying both the column names as the first and second operand.

Syntax of SQL Multiplication Operator:

1. SELECT operand1 * operand2;

Let's understand the below example which explains how to execute Multiplication Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_Monthlybonus**.

Emp Id	Emp Name	Emp Salary	Pe
201	Abhay	25000	20
202	Sumit	30000	50

- o Suppose, we want to double the salary of each employee given in the **Employee_details** table. Then, we have to write the following query in the SQL:
 1. SELECT Emp_Salary * 2 as Emp_New_Salary FROM Employee_details;

In this query, we have performed the SQL multiplication operation on the single column of the given table.

- o If we want to multiply **the Emp_Id** column to **Emp_Salary** column of that employee whose **Emp_Id** is **202**, then we have to write the following query in SQL:
 1. SELECT Emp_Id * Emp_Salary as Emp_Id * Emp_Salary FROM Employee_details WHERE **Emp_Id = 202**;

In this query, we have multiplied the values of two columns by using the WHERE clause.

SQL Division Operator (/)

The Division Operator in SQL divides the operand on the left side by the operand on the right side.

Syntax of SQL Division Operator:

1. `SELECT operand1 / operand2;`

In SQL, we can also divide the numerical values of one column by another column of the same table by specifying both column names as the first and second operand.

We can also perform the division operation on the stored numbers in the column of the SQL table.

Let's understand the below example which explains how to execute Division Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	25000
202	Sumit	30000

- o Suppose, we want to half the salary of each employee given in the Employee_details table. For this operation, we have to write the following query in the SQL:

1. `SELECT Emp_Salary / 2 as Emp_New_Salary FROM Employee_details;`

In this query, we have performed the SQL division operation on the single column of the given table.

SQL Modulus Operator (%)

The Modulus Operator in SQL provides the remainder when the operand on the left side is divided by the operand on the right side.

Syntax of SQL Modulus Operator:

1. `SELECT operand1 % operand2;`

Let's understand the below example which explains how to execute Modulus Operator in SQL query:

This example consists of a **Division** table, which has three columns **Number**, **First_operand**, and **Second_operand**.

Number	First operand	Second operand
1	56	4
2	32	8
3	89	9
4	18	10
5	10	5

- o If we want to get the remainder by dividing the numbers of First_operand column by the numbers of Second_operand column, then we have to write the following query in SQL:
1. SELECT First_operand % Second_operand as Remainder FROM Employee_details;

SQL Comparison Operators

The **Comparison Operators** in SQL compare two different data of SQL table and check whether they are the same, greater, and lesser. The SQL comparison operators are used with the WHERE clause in the SQL queries.

Following are the various comparison operators which are performed on the data stored in the SQL database tables:

1. SQL Equal Operator (=)
2. SQL Not Equal Operator (!=)
3. SQL Greater Than Operator (>)
4. SQL Greater Than Equals to Operator (>=)
5. SQL Less Than Operator (<)\
6. SQL Less Than Equals to Operator (<=)

SQL Equal Operator (=)

This operator is highly used in SQL queries. The **Equal Operator** in SQL shows only data that matches the specified value in the query.

This operator returns TRUE records from the database table if the value of both operands specified in the query is matched.

Let's understand the below example which explains how to execute Equal Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	30000
202	Ankit	40000
203	Bheem	30000
204	Ram	29000
205	Sumit	30000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is 30000. Then, we have to write the following query in the SQL database:

1. `SELECT * FROM Employee_details WHERE Emp_Salary = 30000;`

In this example, we used the SQL equal operator with WHERE clause for getting the records of those employees whose salary is 30000.

SQL Equal Not Operator (!=)

The **Equal Not Operator** in SQL shows only those data that do not match the query's specified value.

This operator returns those records or rows from the database views and tables if the value of both operands specified in the query is not matched with each other.

Let's understand the below example which explains how to execute Equal Not Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is not 45000. Then, we have to write the following query in the SQL database:

1. `SELECT * FROM Employee_details WHERE Emp_Salary != 45000;`

In this example, we used the SQL equal not operator with WHERE clause for getting the records of those employees whose salary is not 45000.

SQL Greater Than Operator (>)

The **Greater Than Operator** in SQL shows only those data which are greater than the value of the right-hand operand.

Let's understand the below example which explains how to execute Greater ThanOperator (>) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is greater than 202. Then, we have to write the following query in the SQL database:

1. SELECT * FROM Employee_details WHERE Emp_Id > 202;

Here, SQL greater than operator displays the records of those employees from the above table whose Employee Id is greater than 202.

SQL Greater Than Equals to Operator (\geq)

The **Greater Than Equals to Operator** in SQL shows those data from the table which are greater than and equal to the value of the right-hand operand.

Let's understand the below example which explains how to execute greater than equals to the operator (\geq) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is greater than and equals to 202. For this, we have to write the following query in the SQL database:

1. SELECT * FROM Employee_details WHERE Emp_Id \geq 202;

Here, '**SQL greater than equals to operator**' with WHERE clause displays the rows of those employees from the table whose Employee Id is greater than and equals to 202.

SQL Less Than Operator (<)

The **Less Than Operator** in SQL shows only those data from the database tables which are less than the value of the right-side operand.

This comparison operator checks that the left side operand is lesser than the right side operand. If the condition becomes true, then this operator in SQL displays the data which is less than the value of the right-side operand.

Let's understand the below example which explains how to execute less than operator (<) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is less than 204. For this, we have to write the following query in the SQL database:

1. `SELECT * FROM Employee_details WHERE Emp_Id < 204;`

Here, **SQL less than operator** with WHERE clause displays the records of those employees from the above table whose Employee Id is less than 204.

SQL Less Than Equals to Operator (\leq)

The **Less Than Equals to Operator** in SQL shows those data from the table which are lesser and equal to the value of the right-side operand.

This comparison operator checks that the left side operand is lesser and equal to the right side operand.

Let's understand the below example which explains how to execute less than equals to the operator (\leq) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, and **Emp_Salary**.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- o Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is less and equals **203**. For this, we have to write the following query in the SQL database:

1. `SELECT * FROM Employee_details WHERE Emp_Id <= 203;`

Here, SQL **less than equals to the operator** with WHERE clause displays the rows of those employees from the table whose Employee Id is less than and equals 202.

SQL Logical Operators

The **Logical Operators** in SQL perform the Boolean operations, which give two results **True and False**. These operators provide **True** value if both operands match the logical condition.

Following are the various logical operators which are performed on the data stored in the SQL database tables:

1. SQL ALL operator
2. SQL AND operator
3. SQL OR operator
4. SQL BETWEEN operator
5. SQL IN operator
6. SQL NOT operator
7. SQL ANY operator
8. SQL LIKE operator

SQL ALL Operator

The ALL operator in SQL compares the specified value to all the values of a column from the sub-query in the SQL database.

This operator is always used with the following statement:

1. SELECT,
2. HAVING, and
3. WHERE.

Syntax of ALL operator:

1. `SELECT column_Name1, ..., column_NameN FROM table_Name WHERE column Comparison_operator ALL (SELECT column FROM tablename2)`

Let's understand the below example which explains how to execute ALL logical operators in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Gurgaon
202	Ankit	45000	Delhi
203	Bheem	30000	Jaipur
204	Ram	29000	Mumbai
205	Sumit	40000	Kolkata

- o If we want to access the employee id and employee names of those employees from the table whose salaries are greater than the salary of employees who lives in Jaipur city, then we have to type the following query in SQL.

1. `SELECT Emp_Id, Emp_Name FROM Employee_details WHERE Emp_Salary > ALL (SELECT Emp_Salary FROM Employee_details WHERE Emp_City = Jaipur)`

Here, we used the **SQL ALL operator** with greater than the operator.

SQL AND Operator

The **AND operator** in SQL would show the record from the database table if all the conditions separated by the AND operator evaluated to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of AND operator:

1. SELECT column1,, columnN FROM table_Name WHERE condition1 AND condition2 AND condition3 AND AND conditionN;

Let's understand the below example which explains how to execute AND logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- o Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is 25000 and the city is Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi';

Here, **SQL AND operator** with WHERE clause shows the record of employees whose salary is 25000 and the city is Delhi.

SQL OR Operator

The OR operator in SQL shows the record from the table if any of the conditions separated by the OR operator evaluates to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of OR operator:

1. SELECT column1,, columnN FROM table_Name WHERE condition1 OR condition2 OR condition3 OR OR conditionN;

Let's understand the below example which explains how to execute OR logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- o If we want to access all the records of those employees from the **Employee_details** table whose salary is 25000 or the city is Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE **Emp_Salary** = **25000** OR **Emp_City** = '**Delhi**';

Here, **SQL OR operator** with WHERE clause shows the record of employees whose salary is 25000 or the city is Delhi.

SQL BETWEEN Operator

The **BETWEEN operator** in SQL shows the record within the range mentioned in the SQL query. This operator operates on the numbers, characters, and date/time operands.

If there is no value in the given range, then this operator shows NULL value.

Syntax of BETWEEN operator:

1. SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_nameBETWEEN value1 and value2;

Let's understand the below example which explains how to execute BETWEEN logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to access all the information of those employees from the **Employee_details** table who is having salaries between 20000 and 40000. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary BETWEEN 30000 AND 45000;

Here, we used the **SQL BETWEEN operator** with the Emp_Salary field.

SQL IN Operator

The **IN operator** in SQL allows database users to specify two or more values in a WHERE clause. This logical operator minimizes the requirement of multiple OR conditions.

This operator makes the query easier to learn and understand. This operator returns those rows whose values match with any value of the given list.

Syntax of IN operator:

1. SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_name IN (list_of_values);

Let's understand the below example which explains how to execute IN logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is 202, 204, and 205. For this, we have to write the following query in SQL:

1. `SELECT * FROM Employee_details WHERE Emp_Id IN (202, 204, 205);`

Here, we used the **SQL IN operator** with the Emp_Id column.

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is not equal to 202 and 205. For this, we have to write the following query in SQL:

1. `SELECT * FROM Employee_details WHERE Emp_Id NOT IN (202,205);`
2.

Here, we used the **SQL NOT IN operator** with the Emp_Id column.

SQL NOT Operator

The **NOT operator** in SQL shows the record from the table if the condition evaluates to false. It is always used with the WHERE clause.

Syntax of NOT operator:

1. SELECT column1, column2, columnN FROM table_Name WHERE NOT condition;

Let's understand the below example which explains how to execute NOT logical operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose City is not Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' ;

In this example, we used the **SQL NOT operator** with the Emp_City column.

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose City is not Delhi and Chandigarh. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' AND NOT Emp_City = 'Chandigarh';

In this example, we used the **SQL NOT operator** with the Emp_City column.

SQL ANY Operator

The **ANY operator** in SQL shows the records when any of the values returned by the sub-query meet the condition.

The ANY logical operator must match at least one record in the inner query and must be preceded by any SQL comparison operator.

Syntax of ANY operator:

1. SELECT column1, column2, columnN FROM table_Name WHERE column_name comparison_operator ANY (SELECT column_name FROM table_name WHERE condition(s)) ;

SQL LIKE Operator

The **LIKE operator** in SQL shows those records from the table which match with the given pattern specified in the sub-query.

The percentage (%) sign is a wildcard which is used in conjunction with this logical operator.

This operator is used in the WHERE clause with the following three statements:

1. SELECT statement
2. UPDATE statement
3. DELETE statement

Syntax of LIKE operator:

1. SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_name LIKE pattern;

Let's understand the below example which explains how to execute LIKE logical operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp C
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandi
203	Saket	30000	Delhi

204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- o If we want to show all the information of those employees from the **Employee_details** whose name starts with "s". For this, we have to write the following query in SQL:

1. `SELECT * FROM Employee_details WHERE Emp_Name LIKE 's%' ;`

In this example, we used the SQL LIKE operator with **Emp_Name** column because we want to access the record of those employees whose name starts with s.

- o If we want to show all the information of those employees from the **Employee_details** whose name ends with "y". For this, we have to write the following query in SQL:

1. `SELECT * FROM Employee_details WHERE Emp_Name LIKE '%y' ;`

- o If we want to show all the information of those employees from the **Employee_details** whose name starts with "S" and ends with "y". For this, we have to write the following query in SQL:

1. `SELECT * FROM Employee_details WHERE Emp_Name LIKE 'S%y' ;`

SQL Set Operators

The **Set Operators** in SQL combine a similar type of data from two or more SQL database tables. It mixes the result, which is extracted from two or more SQL queries, into a single result.

Set operators combine more than one select statement in a single query and return a specific result set.

Following are the various set operators which are performed on the similar data stored in the two SQL database tables:

1. SQL Union Operator
2. SQL Union ALL Operator
3. SQL Intersect Operator
4. SQL Minus Operator

SQL Union Operator

The SQL Union Operator combines the result of two or more SELECT statements and provides the single output.

The data type and the number of columns must be the same for each SELECT statement used with the UNION operator. This operator does not show the duplicate records in the output table.

Syntax of UNION Set operator:

1. SELECT column1, column2, columnN FROM table_Name1 [WHERE condition s]
2. UNION
3. SELECT column1, column2, columnN FROM table_Name2 [WHERE condition s];

Let's understand the below example which explains how to execute Union operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

- Suppose, we want to see the employee name and employee id of each employee from both tables in a single output. For this, we have to write the following query in SQL:
 1. SELECT Emp_ID, Emp_Name FROM Employee_details1
 2. UNION
 3. SELECT Emp_ID, Emp_Name FROM Employee_details2 ;

SQL Union ALL Operator

The SQL Union Operator is the same as the UNION operator, but the only difference is that it also shows the same record.

Syntax of UNION ALL Set operator:

1. SELECT column1, column2, columnN FROM table_Name1 [WHERE condition s]
2. UNION ALL
3. SELECT column1, column2, columnN FROM table_Name2 [WHERE condition s];

Let's understand the below example which explains how to execute Union ALL operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
--------	----------	------------	----------

203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

- o If we want to see the employee name of each employee of both tables in a single output. For this, we have to write the following query in SQL:
1. SELECT Emp_Name FROM Employee_details1
 2. UNION ALL
 3. SELECT Emp_Name FROM Employee_details2 ;

SQL Intersect Operator

The SQL Intersect Operator shows the common record from two or more SELECT statements. The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.

Syntax of INTERSECT Set operator:

1. SELECT column1, column2, columnN FROM table_Name1 [WHERE condition s]
2. INTERSECT
3. SELECT column1, column2, columnN FROM table_Name2 [WHERE condition s];

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi

203	Saket	30000	Aligarh
-----	-------	-------	---------

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see a common record of the employee from both the tables in a single output. For this, we have to write the following query in SQL:

1. SELECT Emp_Name FROM Employee_details1
2. INTERSECT
3. SELECT Emp_Name FROM Employee_details2 ;

SQL Minus Operator

The SQL Minus Operator combines the result of two or more SELECT statements and shows only the results from the first data set.

Syntax of MINUS operator:

1. SELECT column1, column2, columnN FROM First_tablename [WHERE conditions]
2. MINUS
3. SELECT column1, column2, columnN FROM Second_tablename [WHERE conditions];

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, **we used two tables**. Both tables have four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see the name of employees from the first result set after the combination of both tables. For this, we have to write the following query in SQL:

1. SELECT Emp_Name FROM Employee_details1
2. MINUS
3. SELECT Emp_Name FROM Employee_details2 ;

SQL Unary Operators

The **Unary Operators** in SQL perform the unary operations on the single data of the SQL table, i.e., these operators operate only on one operand.

These types of operators can be easily operated on the numeric data value of the SQL table.

Following are the various unary operators which are performed on the numeric data stored in the SQL table:

1. SQL Unary Positive Operator
2. SQL Unary Negative Operator

3. SQL Unary Bitwise NOT Operator

SQL Unary Positive Operator

The SQL Positive (+) operator makes the numeric value of the SQL table positive.

Syntax of Unary Positive Operator

1. `SELECT +(column1), +(column2), +(columnN) FROM table_Name [WHERE conditions] ;`

Let's understand the below example which explains how to execute a Positive unary operator on the data of SQL table:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- o Suppose, we want to see the salary of each employee as positive from the Employee_details table. For this, we have to write the following query in SQL:

1. `SELECT +Emp_Salary Employee_details ;`

SQL Unary Negative Operator

The SQL Negative (-) operator makes the numeric value of the SQL table negative.

Syntax of Unary Negative Operator

1. `SELECT -(column_Name1), -(column_Name2), -(column_NameN) FROM table_Name [WHERE conditions] ;`

Let's understand the below example which explains how to execute Negative unary operator on the data of SQL table:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to see the salary of each employee as negative from the Employee_details table. For this, we have to write the following query in SQL:
 1. `SELECT -Emp_Salary Employee_details ;`
 - Suppose, we want to see the salary of those employees as negative whose city is Kolkata in the Employee_details table. For this, we have to write the following query in SQL:
 1. `SELECT -Emp_Salary Employee_details WHERE Emp_City = 'Kolkata';`

SQL Bitwise NOT Operator

The SQL Bitwise NOT operator provides the one's complement of the single numeric operand. This operator turns each bit of numeric value. If the bit of any numerical value is 001100, then this operator turns these bits into 110011.

Syntax of Bitwise NOT Operator

1. `SELECT ~ (column1), ~ (column2), ~ (columnN) FROM table_Name [WHERE conditions] ;`

Let's understand the below example which explains how to execute the Bitwise NOT operator on the data of SQL table:

This example consists of a **Student_details** table, which has four columns **Roll_No**, **Stu_Name**, **Stu_Marks**, and **Stu_City**.

Emp Id	Stu Name	Stu Marks	Stu City
101	Sanjay	85	Delhi
102	Ajay	97	Chandigarh
103	Saket	45	Delhi
104	Abhay	68	Delhi
105	Sumit	60	Kolkata

If we want to perform the Bitwise Not operator on the marks column of **Student_details**, we have to write the following query in SQL:

1. `SELECT ~Stu_Marks Employee_details ;`

SQL Bitwise Operators

The **Bitwise Operators** in SQL perform the bit operations on the Integer values. To understand the performance of Bitwise operators, you just knew the basics of Boolean algebra.

Following are the two important logical operators which are performed on the data stored in the SQL database tables:

1. Bitwise AND (&)
2. Bitwise OR(|)

Bitwise AND (&)

The Bitwise AND operator performs the logical AND operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise AND Operator

1. `SELECT column1 & column2 & & columnN FROM table_Name [WHERE conditions] ;`

Let's understand the below example which explains how to execute Bitwise AND operator on the data of SQL table:

This example consists of the following table, which has two columns. Each column holds numerical values.

When we use the Bitwise AND operator in SQL, then SQL converts the values of both columns in binary format, and the AND operation is performed on the converted bits.

After that, SQL converts the resultant bits into user understandable format, i.e., decimal format.

Column1	Column2
1	1
2	5
3	4
4	2
5	3

- Suppose, we want to perform the Bitwise AND operator between both the columns of the above table. For this, we have to write the following query in SQL:

1. `SELECT Column1 & Column2 From TABLE_AND ;`

Bitwise OR (|)

The Bitwise OR operator performs the logical OR operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise OR Operator

1. `SELECT column1 | column2 | | columnN FROM table_Name [WHERE conditions] ;`

Let's understand the below example which explains how to execute Bitwise OR operator on the data of SQL table:

This example consists of a table that has two columns. Each column holds numerical values.

When we used the Bitwise OR operator in SQL, then SQL converts the values of both columns in binary format, and the OR operation is performed on the binary bits. After that, SQL converts the resultant binary bits into user understandable format, i.e., decimal format.

Column1	Column2
1	1
2	5
3	4
4	2
5	3

- o Suppose, we want to perform the Bitwise OR operator between both the columns of the above table. For this, we have to write the following query in SQL:
 1. `SELECT Column1 | Column2 From TABLE_OR ;`

Precedence of SQL Operator

The precedence of SQL operators is the sequence in which the SQL evaluates the different operators in the same expression. Structured Query Language evaluates those operators first, which have high precedence.

In the following table, the operators at the top have high precedence, and the operators that appear at the bottom have low precedence.

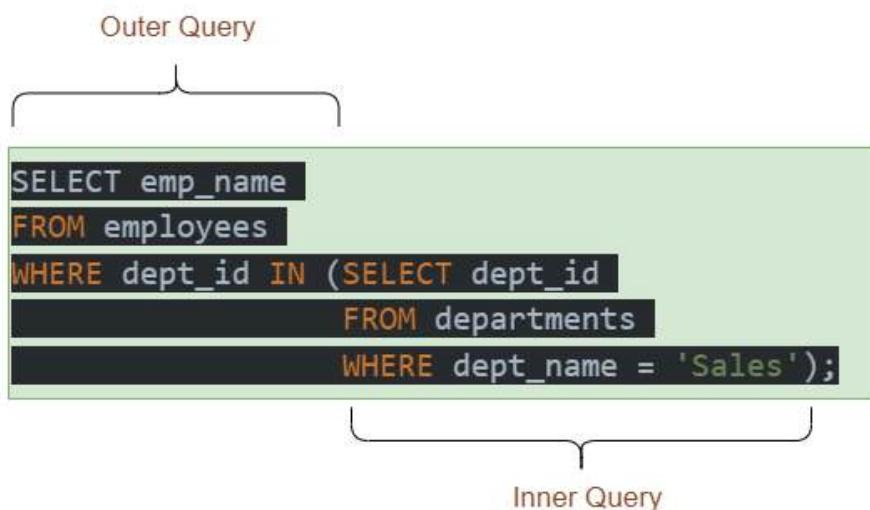
SQL Operator Symbols	Operators
<code>**</code>	Exponentiation operator
<code>+, -</code>	Identity operator, Negation operator

*	Multiplication operator, Division operator
+, -,	Addition (plus) operator, subtraction (minus) operator, String Concatenation operator
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison Operators
NOT	Logical negation operator
&& or AND	Conjunction operator
OR	Inclusion operator

Structured Query Language (SQL) is a programming language. SQL is used to manage data stored in a relational database. SQL has the ability of nest queries. A nested query is a query within another query. Nested query allows for more complex and specific data retrieval. In this article, we will discuss nested queries in SQL, their syntax, and examples.

Nested Query

In SQL, a nested query involves a query that is placed within another query. Output of the inner query is used by the outer query. A nested query has two SELECT statements: one for the inner query and another for the outer query.



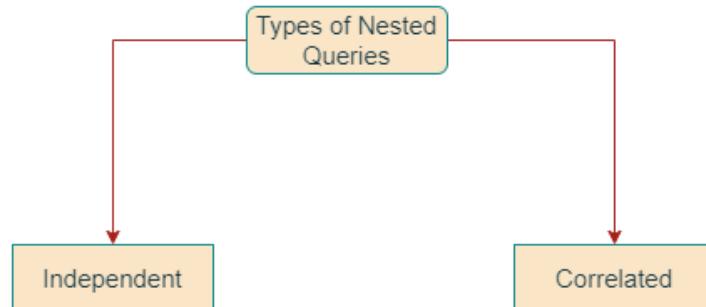
Syntax of Nested Queries

The basic syntax of a nested query involves placing one query inside of another query. Inner query or subquery is executed first and returns a set of values that are then used by the outer query. The syntax for a nested query is as follows:

```
SELECT column1, column2, ...
FROM table1
WHERE column1 IN ( SELECT column1
    FROM table2
    WHERE condition );
```

Types of Nested Queries in SQL

Subqueries can be either correlated or non-correlated



Non-correlated (or Independent) Nested Queries

Non-correlated (or Independent) Nested Queries : Non-correlated (or Independent) subqueries are executed independently of the outer query. Their results are passed to the outer query.

Correlated Nested Queries

Correlated subqueries are executed once for each row of the outer query. They use values from the outer query to return results.

Execution Order in Independent Nested Queries

In independent nested queries, the execution order is from the innermost query to the outer query. An outer query won't be executed until its inner

query completes its execution. The outer query uses the result of the inner query.

Operators Used in Independent Nested Queries

IN Operator

This operator checks if a column value in the outer query's result is present in the inner query's result. The final result will have rows that satisfy the IN condition.

NOT IN Operator

This operator checks if a column value in the outer query's result is not present in the inner query's result. The final result will have rows that satisfy the NOT IN condition.

ALL Operator

This operator compares a value of the outer query's result with all the values of the inner query's result and returns the row if it matches all the values.

ANY Operator

This operator compares a value of the outer query's result with all the inner query's result values and returns the row if there is a match with any value.

Execution Order in Co-related Nested Queries

In correlated nested queries, the inner query uses values from the outer query, and the execution order is different from that of independent nested queries.

- First, the outer query selects the first row.
- Inner query uses the value of the selected row. It executes its query and returns a result set.
- Outer query uses the result set returned by the inner query. It determines whether the selected row should be included in the final output.
- Steps 2 and 3 are repeated for each row in the outer query's result set.
- This process can be resource-intensive. It may lead to performance issues if the query is not optimized properly.

Operators Used in Co-related Nested Queries

In co-related nested queries, the following operators can be used

EXISTS Operator

This operator checks whether a subquery returns any row. If it returns at least one row, the EXISTS operator returns true, and the outer query continues to execute. If the subquery returns no row, the EXISTS operator returns false, and the outer query stops execution.

NOT EXISTS Operator

This operator checks whether a subquery returns no rows. If the subquery returns no row, the NOT EXISTS operator returns true, and the outer query continues to execute. If the subquery returns at least one row, the NOT EXISTS operator returns false, and the outer query stops execution.

ANY Operator

This operator compares a value of the outer query's result with one or more values returned by the inner query. If the comparison is true for any one of the values returned by the inner query, the row is included in the final result.

ALL Operator

This operator compares a value of the outer query's result with all the values returned by the inner query. Only if the comparison is true for all the values returned by the inner query, the row is included in the final result.

These operators are used to create co-related nested queries that depend on values from the outer query for execution.

Examples

Consider the following sample table to execute nested queries on these.

Table: employees table

emp_id	emp_name	dept_id
---------------	-----------------	----------------

1	John	1
2	Mary	2
3	Bob	1
4	Alice	3
5	Tom	1

Table: departments table

dept_id	dept_name
1	Sales
2	Marketing
3	Finance

Table: sales table

sale_id	emp_id	sale_amt
1	1	1000
2	2	2000
3	3	3000
4	1	4000
5	5	5000
6	3	6000
7	2	7000

Example 1: Find the names of all employees in the Sales department.

Required query

```
SELECT emp_name
FROM employees
WHERE dept_id IN (SELECT dept_id
                   FROM departments
                   WHERE dept_name = 'Sales');
```

Output

emp_name

John

Bob

Tom

Example 2: Find the names of all employees who have made a sale

Required query

```
SELECT emp_name
FROM employees
WHERE EXISTS (SELECT emp_id
               FROM sales
               WHERE employees.emp_id = sales.emp_id);
```

Output

emp_name

John

Mary

Bob

Alice

Tom

This query selects all employees from the "employees" table where there exists a sale record in the "sales" table for that employee.

Example 3: Find the names of all employees who have made sales greater than \$1000.

Required query

```
SELECT emp_name  
FROM employees  
WHERE emp_id = ALL (SELECT emp_id  
                      FROM sales  
                     WHERE sale_amt > 1000);
```

Output

emp_name

John

Mary

Bob

Alice

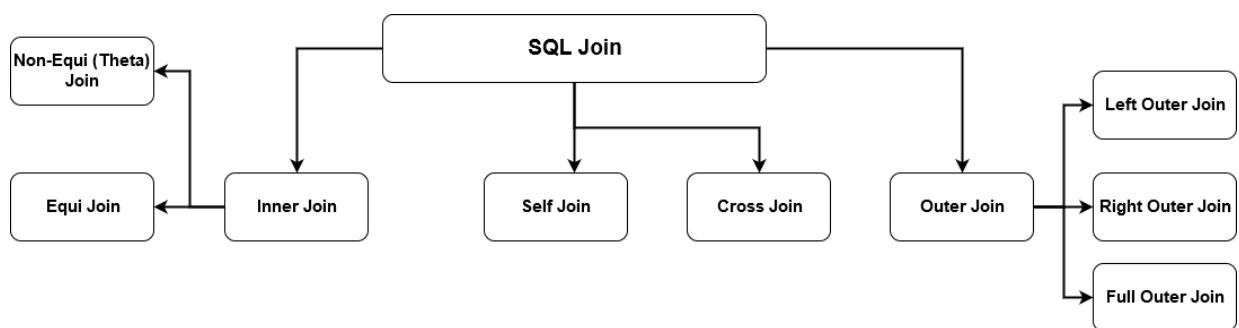
Tom

This query selects all employees from the "employees" table. With the condition that where their emp_id equals all the emp_ids in the "sales" table where the sale amount is greater than \$1000. Since all employees have made a sale greater than \$1000, all employee names are returned.

SQL JOINS

SQL JOIN is a clause that is used to combine multiple tables and retrieve data based on a common field in relational databases. Database professionals use normalizations for ensuring and improving data integrity. In the various normalization forms, data is distributed into multiple logical tables. These tables use referential constraints – primary key and foreign keys – to enforce data integrity in SQL Server tables. In the below image, we get a glimpse of the database normalization process.

SQL JOIN generates meaningful data by combining multiple relational tables. These tables are related using a key and have one-to-one or one-to-many relationships. To retrieve the correct data, you must know the data requirements and correct join mechanisms. SQL Server supports multiple joins and each method has a specific way to retrieve data from multiple tables. The below image specifies the supported SQL Server joins.



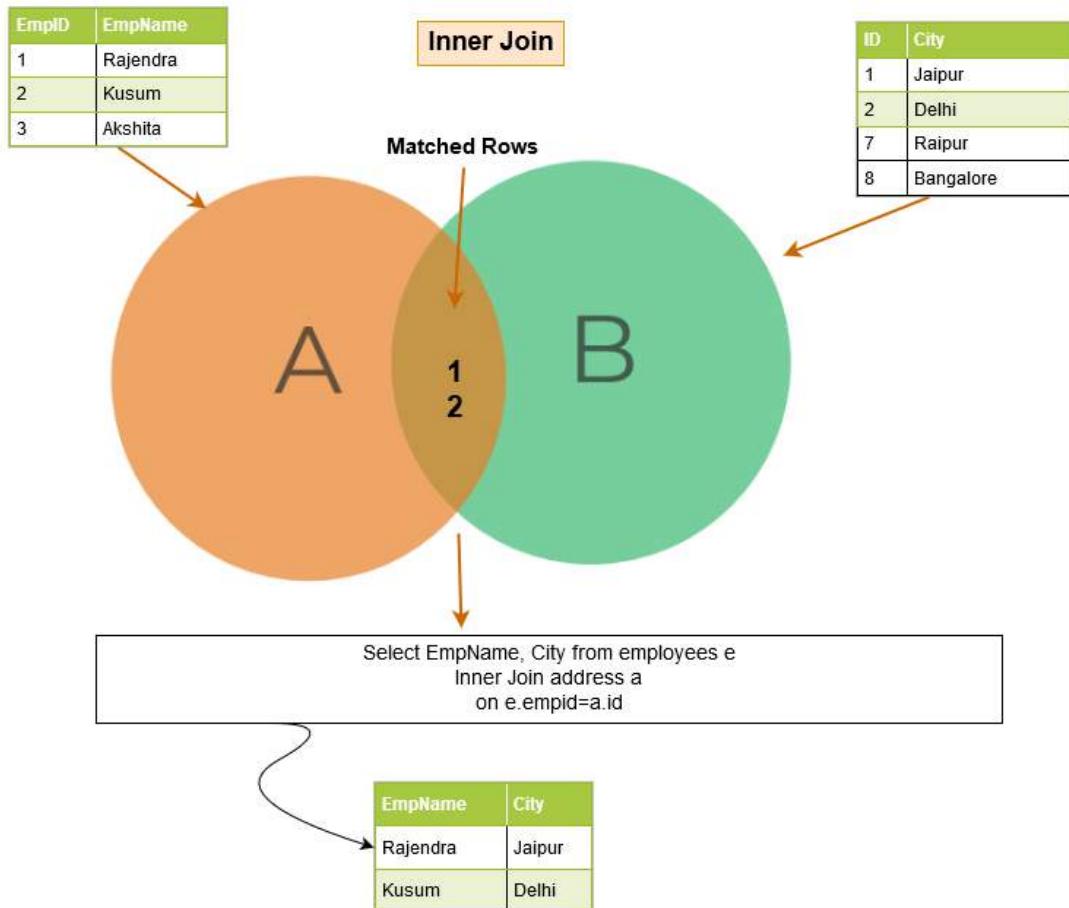
SQL inner join

The SQL inner join includes rows from the tables where the join conditions are satisfied. For example, in the below Venn diagram, inner join returns the matching rows from Table A and Table B.

In the below example, notice the following things:

- We have two tables – [Employees] and [Address].
- The SQL query is joined on the [Employees].[EmpID] and [Address].[ID] column.

The query output returns the employee records for EmpID that exists in both tables.



The inner join returns matching rows from both tables; therefore, it is also known as Equi join. If we don't specify the inner keyword, SQL Server performs the inner join operation.

```
198 % 
Select EmpName, City
from employees e
join address a
on e.empid=a.id
```

Results

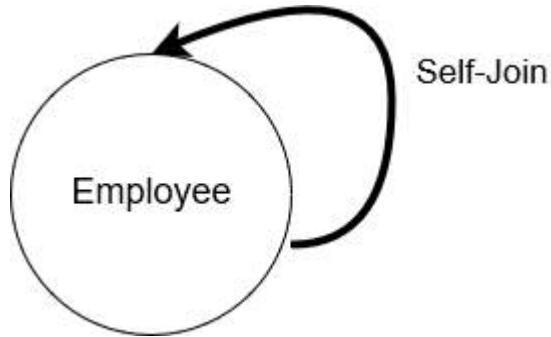
	EmpName	City
1	Rajendra	Jaipur
2	Kusum	Delhi

In another type of inner join, a theta join, we do not use the equality operator (=) in the ON clause. Instead, we use non-equality operators such as < and >.

```
SELECT * FROM Table1 T1, Table2 T2 WHERE T1.Price < T2.price
```

SQL self-join

In a self-join, SQL Server joins the table with itself. This means the table name appears twice in the from clause.



Below, we have a table [Emp] that has employees as well as their managers' data. The self-join is useful for querying hierarchical data. For example, in the employee table, we can use self-join to learn each employee and their reporting manager's name.

EmpID	Name	EmpMgrid
1	Rajendra	1
2	Mohan	1
3	Amit	1
4	Manoj	1
5	Manish	2
6	Kapil	2



```
SELECT e.EmpID, e.Name, m.Name as Manager FROM Emp e Inner Join Emp m
On e.EmpMgrid=m.EmpID;
```

```
SELECT e.EmpID, e.Name,
m.Name as Manager
FROM Emp e Inner Join Emp m
On e.EmpMgrid=m.EmpID;
```

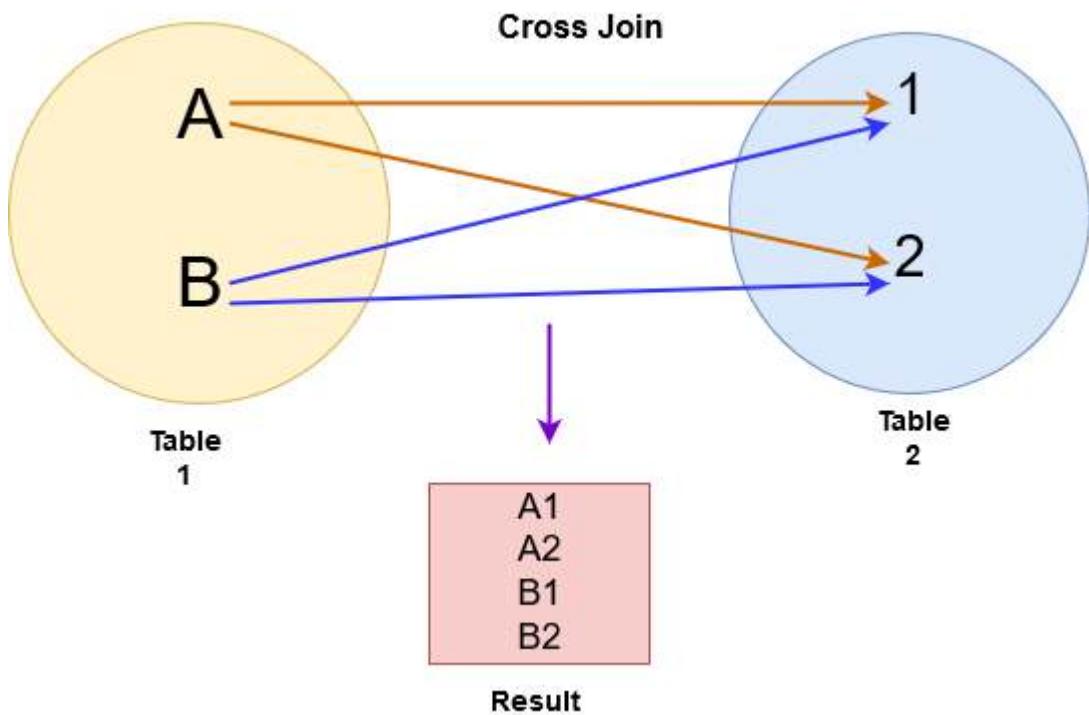
Results Messages

EmplD	Name	Manager
1	Rajendra	Rajendra
2	Mohan	Rajendra
3	Amit	Rajendra
4	Manoj	Rajendra
5	Manish	Mohan
6	Kapil	Mohan

The above query puts a self-join on [Emp] table. It joins the EmpMgrid column with the EmpID column and returns the matching rows.

SQL cross join

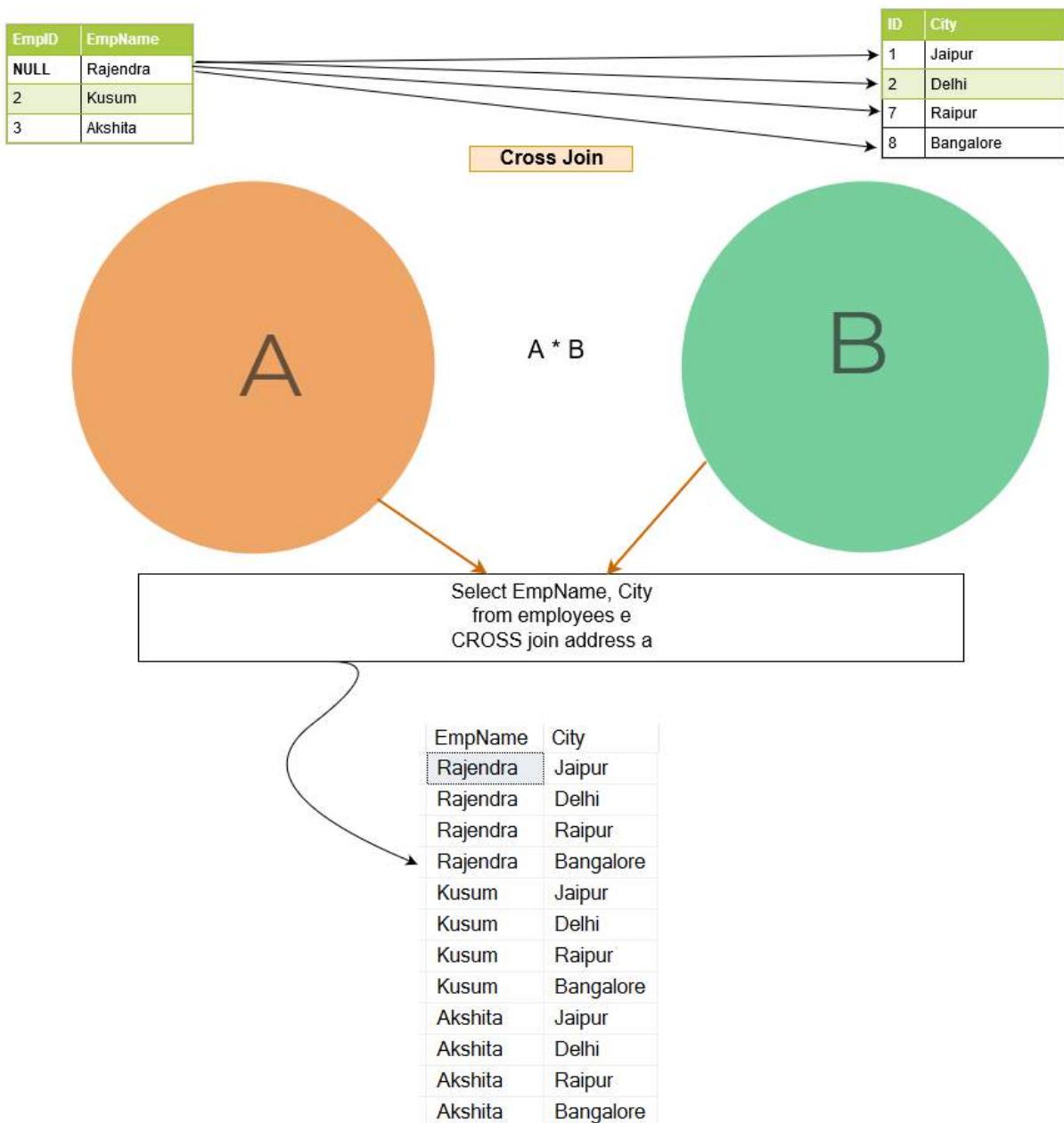
In the cross join, SQL Server returns a Cartesian product from both tables. For example, in the below image, we performed a cross-join for table A and B.



The cross join joins each row from table A to every row available in table B. Therefore, the output is also known as a Cartesian product of both tables. In the below image, note the following:

- Table [Employee] has three rows for Emp ID 1,2 and 3.
- Table [Address] has records for Emp ID 1,2,7 and 8.

In the cross-join output, row 1 of [Employee] table joins with all rows of [Address] table and follows the same pattern for the remaining rows.

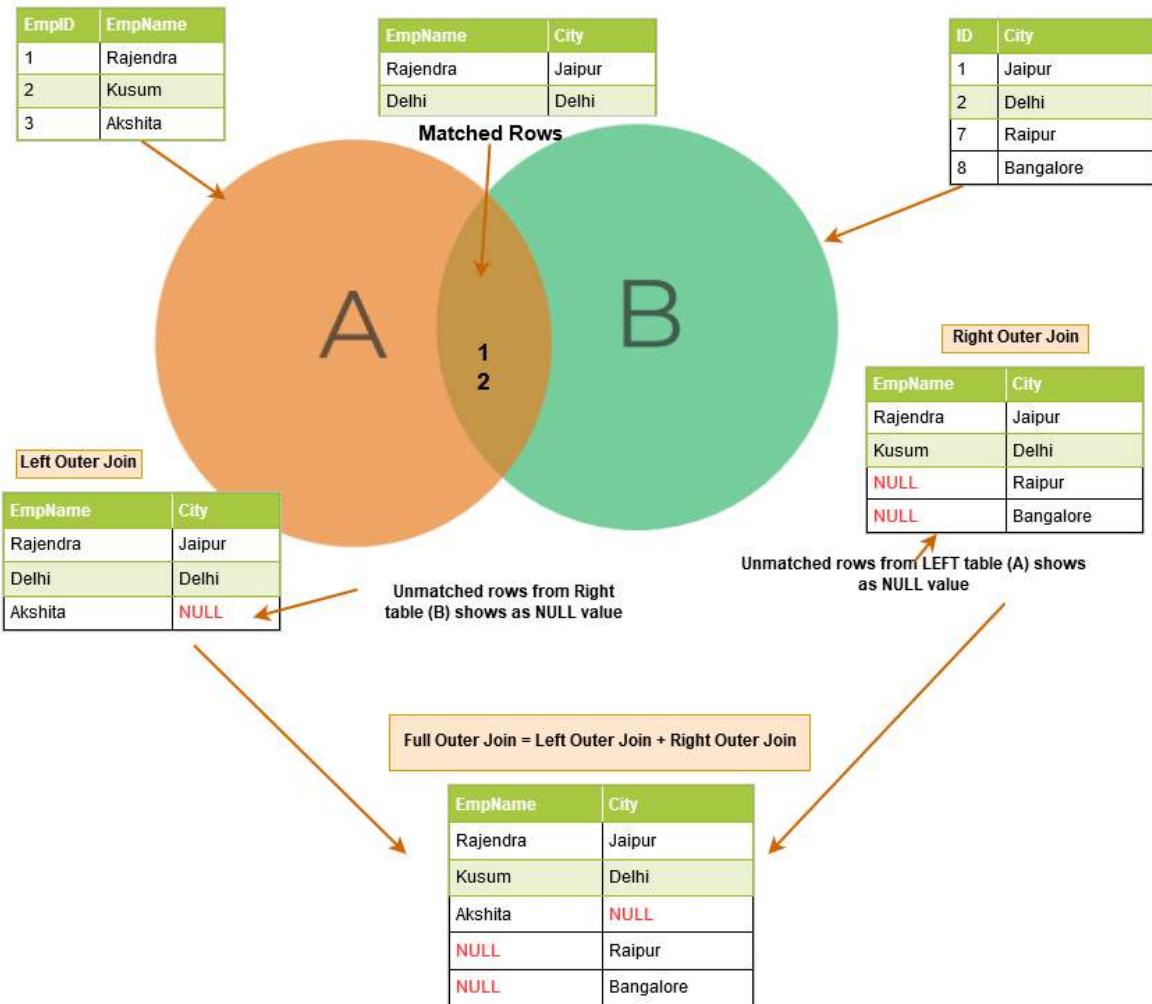


If the first table has x number of rows and the second table has n number of rows, cross join gives $x * n$ number of rows in the output. You should avoid cross join on larger tables because it might return a vast number of records and SQL Server requires a lot of computing power (CPU, memory and IO) for handling such extensive data.

SQL outer join

As we explained earlier, the inner join returns the matching rows from both of the tables. When using a SQL outer join, it not only lists the matching rows, but it also returns the unmatched rows from the other tables. The unmatched row depends on the left, right or full keywords.

The below image describes at a high-level the left, right and full outer join.

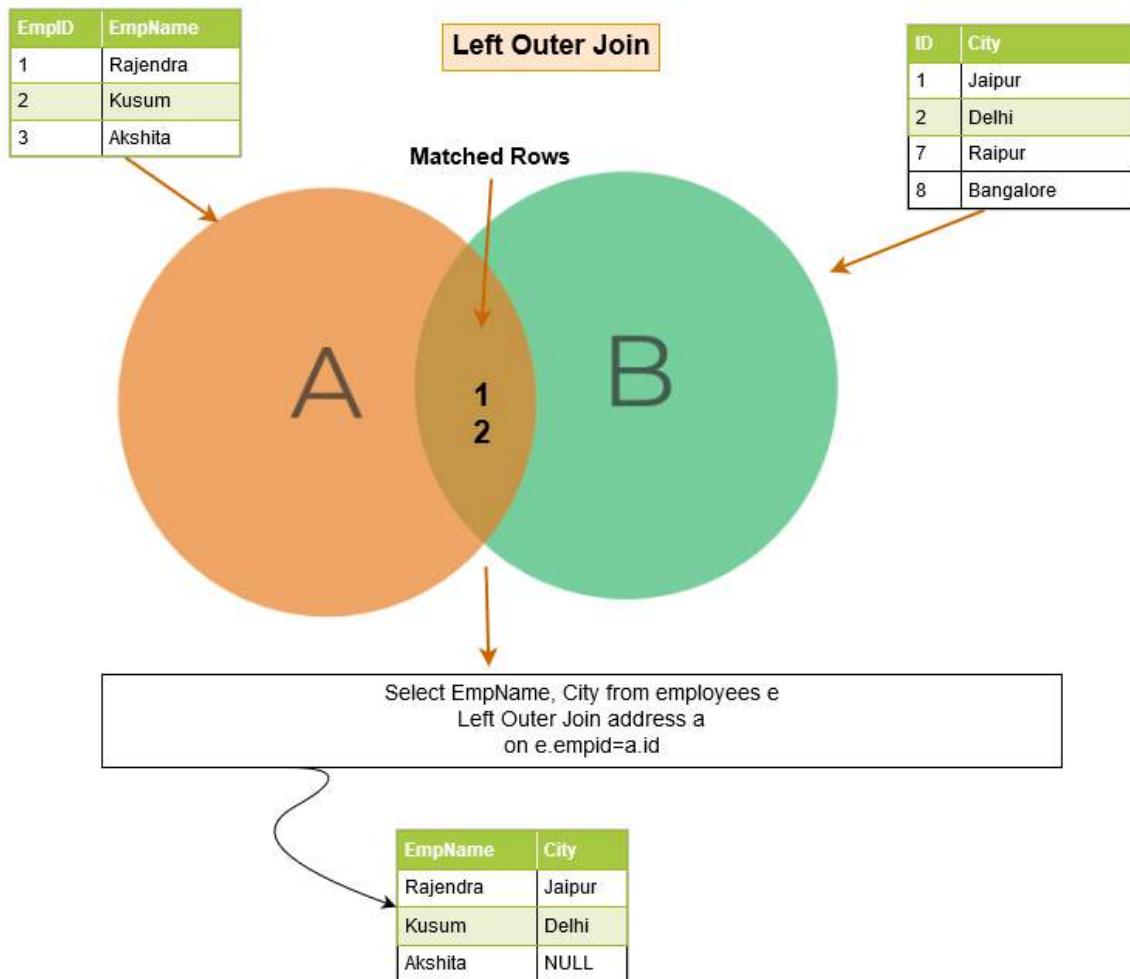


Left outer join

SQL left outer join returns the matching rows of both tables along with the unmatched rows from the left table. If a record from the left table doesn't have any matched rows in the right table, it displays the record with NULL values.

In the below example, the left outer join returns the following rows:

- Matched rows: Emp ID 1 and 2 exists in both the left and right tables.
- Unmatched row: Emp ID 3 doesn't exist on the right table. Therefore, we have a NULL value in the query output.

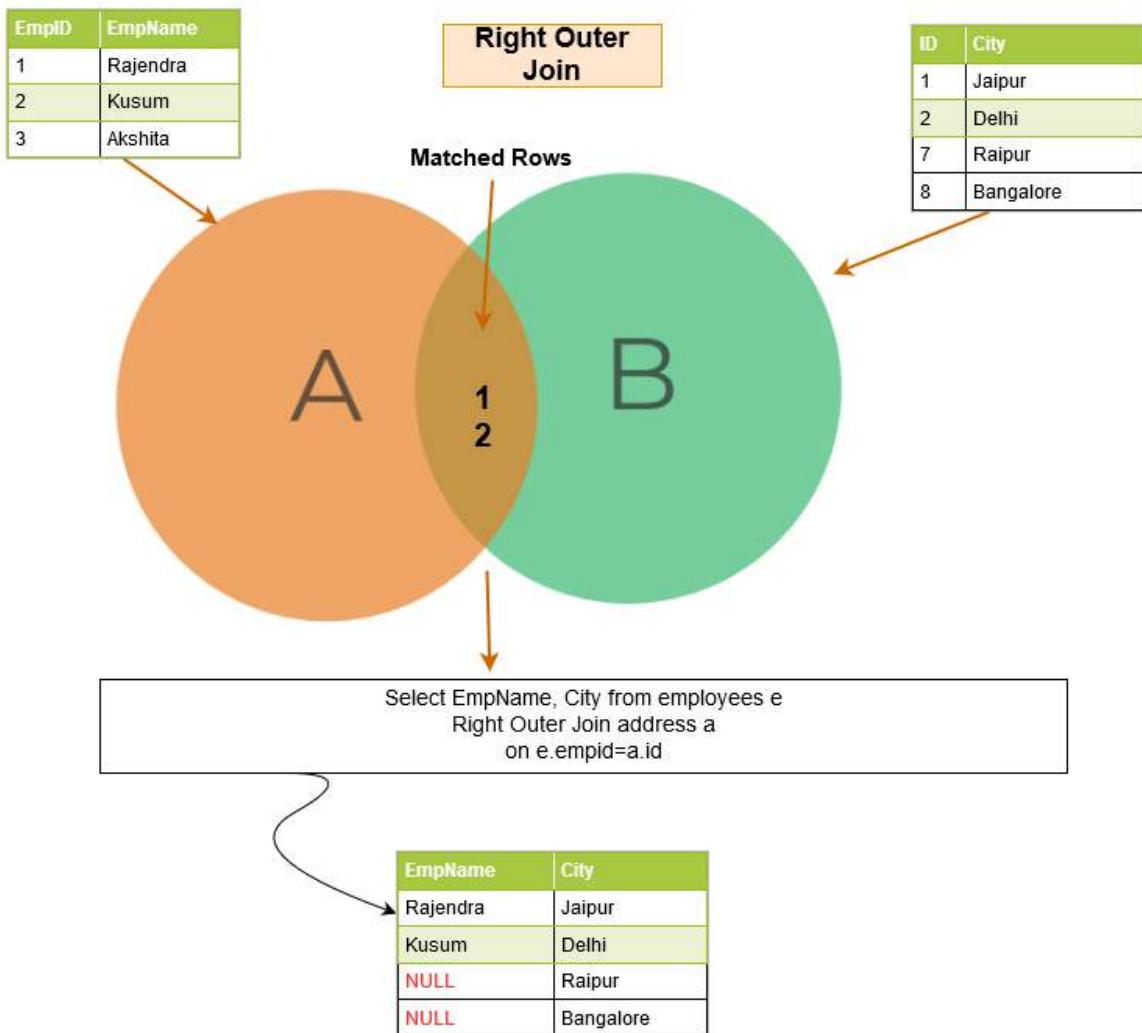


Right outer join

SQL right outer join returns the matching rows of both tables along with the unmatched rows from the right table. If a record from the right table does not have any matched rows in the left table, it displays the record with NULL values.

In the below example, we have the following output rows:

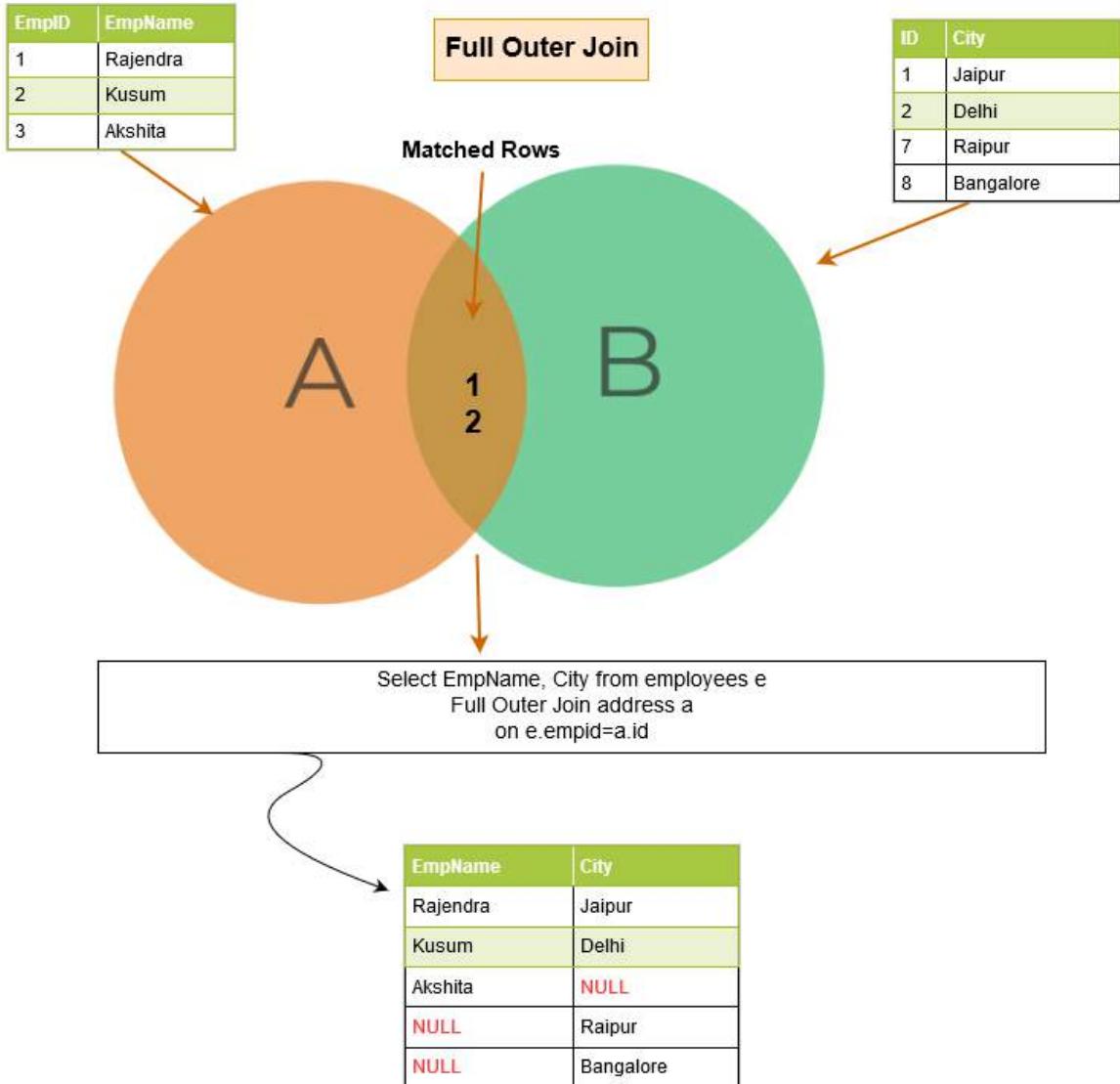
- Matching rows: Emp ID 1 and 2 exists in both tables; therefore, these rows are matched rows.
 - Unmatched rows: In the right table, we have additional rows for Emp ID 7 and 8, but these rows are not available in the left table. Therefore, we get NULL value in the right outer join for these rows.



Full outer join

A full outer join returns the following rows in the output:

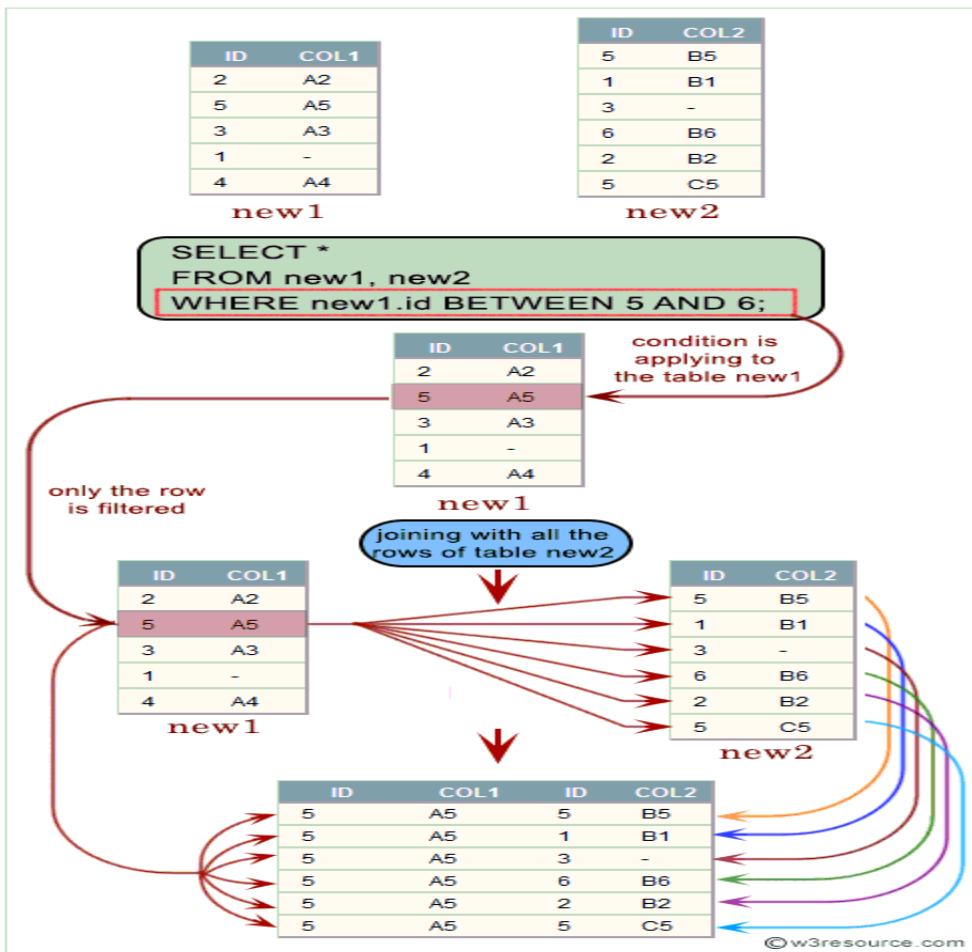
- Matching rows between two tables.
- Unmatched rows similar to left outer join: NULL values for unmatched rows from the right table.
- Unmatched rows similar to right outer join: Null values for unmatched rows from the left table.



NON EQUI JOIN

The SQL NON EQUI JOIN uses comparison operator instead of the equal sign like $>$, $<$, $>=$, $<=$ along with conditions.

Visual presentation of SQL Non Equi Join:



Syntax:

```
SELECT *
FROM table_name1, table_name2
WHERE table_name1.column [> | < | >= | <= ] table_name2.column;
```

Example:

Here is an example of non equi join in SQL between two tables

Sample table: orders

ORD_NUM	ORD_AMOUNT	ADVANCE_AMOUNT	ORD_DATE	CUST_CODE	AGENT_CODE
ORD_DESCRIPTION					
200114	3500	2000	15-AUG-08	C00002	A008
200122	2500	400	16-SEP-08	C00003	A004
200118	500	100	20-JUL-08	C00023	A006
200119	4000	700	16-SEP-08	C00007	A010
200121	1500	600	23-SEP-08	C00008	A004
200130	2500	400	30-JUL-08	C00025	A011
200134	4200	1800	25-SEP-08	C00004	A005
200108	4000	600	15-FEB-08	C00008	A004
200103	1500	700	15-MAY-08	C00021	A005
200105	2500	500	18-JUL-08	C00025	A011
200109	3500	800	30-JUL-08	C00011	A010
200101	3000	1000	15-JUL-08	C00001	A008
200111	1000	300	10-JUL-08	C00020	A008

200104	1500	500	13-MAR-08	C00006	A004
200106	2500	700	20-APR-08	C00005	A002
200125	2000	600	10-OCT-08	C00018	A005
200117	800	200	20-OCT-08	C00014	A001
200123	500	100	16-SEP-08	C00022	A002
200120	500	100	20-JUL-08	C00009	A002
200116	500	100	13-JUL-08	C00010	A009
200124	500	100	20-JUN-08	C00017	A007
200126	500	100	24-JUN-08	C00022	A002
200129	2500	500	20-JUL-08	C00024	A006
200127	2500	400	20-JUL-08	C00015	A003
200128	3500	1500	20-JUL-08	C00009	A002
200135	2000	800	16-SEP-08	C00007	A010
200131	900	150	26-AUG-08	C00012	A012
200133	1200	400	29-JUN-08	C00009	A002
200100	1000	600	08-JAN-08	C00015	A003
200110	3000	500	15-APR-08	C00019	A010
200107	4500	900	30-AUG-08	C00007	A010
200112	2000	400	30-MAY-08	C00016	A007
200113	4000	600	10-JUN-08	C00022	A002
200102	2000	300	25-MAY-08	C00012	A012

Sample table : customer

CUST_CODE	CUST_NAME	CUST_CITY	WORKING_AREA	CUST_COUNTRY	GRADE	OPENING_AMT	RECEIVE_AMT	PAYMENT_AMT	OUTSTANDING_AMT	PHONE_NO	AGENT_CODE
C00013	Holmes	London	London	UK		6000.00	5000.00	7000.00	4000.00	BBBBBBBB	A003
C00001	Micheal	New York	New York	USA		3000.00	5000.00	2000.00	6000.00	CCCCCC	A008
C00020	Albert	New York	New York	USA		5000.00	7000.00	6000.00	6000.00	BBBBSBB	A008
C00025	Ravindran	Bangalore	Bangalore	India		5000.00	7000.00	4000.00	8000.00	AVAVAVA	A011
C00024	Cook	London	London	UK		4000.00	9000.00	7000.00	6000.00	FSDDDSDF	A006
C00015	Stuart	London	London	UK		6000.00	8000.00	3000.00	11000.00	GFSGERS	A003
C00002	Bolt	New York	New York	USA		5000.00	7000.00	9000.00	3000.00	DDNRDRH	A008
C00018	Fleming	Brisban	Brisban	Australia		7000.00	7000.00	9000.00	5000.00	NHBGVFC	A005
C00021	Jacks	Brisban	Brisban	Australia		7000.00	7000.00	7000.00	7000.00	WERTGDF	A005

	C00019		Yearannaidu		Chennai		Chennai		India	
1	8000.00		7000.00		7000.00		8000.00		ZZZZBFV	
	A010									
	C00005		Sasikant		Mumbai		Mumbai		India	
1	7000.00		11000.00		7000.00		11000.00		147-25896312	
	A002									
	C00007		Ramanathan		Chennai		Chennai		India	
1	7000.00		11000.00		9000.00		9000.00		GHRDWSD	
	A010									
	C00022		Avinash		Mumbai		Mumbai		India	
2	7000.00		11000.00		9000.00		9000.00		113-12345678	
	A002									
	C00004		Winston		Brisban		Brisban		Australia	
1	5000.00		8000.00		7000.00		6000.00		AAAAAAA	
	A005									
	C00023		Karl		London		London		UK	
0	4000.00		6000.00		7000.00		3000.00		AAAABAA	
	A006									
	C00006		Shilton		Torento		Torento		Canada	
1	10000.00		7000.00		6000.00		11000.00		DDDDDDD	
	A004									
	C00010		Charles		Hampshair		Hampshair		UK	
3	6000.00		4000.00		5000.00		5000.00		MMMMMM	
	A009									
	C00017		Srinivas		Bangalore		Bangalore		India	
2	8000.00		4000.00		3000.00		9000.00		AAAAAAB	
	A007									
	C00012		Steven		San Jose		San Jose		USA	
1	5000.00		7000.00		9000.00		3000.00		KRFYGJK	
	A012									
	C00008		Karolina		Torento		Torento		Canada	
1	7000.00		7000.00		9000.00		5000.00		HJKORED	
	A004									
	C00003		Martin		Torento		Torento		Canada	
2	8000.00		7000.00		7000.00		8000.00		MJYURFD	
	A004									
	C00009		Ramesh		Mumbai		Mumbai		India	
3	8000.00		7000.00		3000.00		12000.00		Phone No	
	A002									
	C00014		Rangarappa		Bangalore		Bangalore		India	
2	8000.00		11000.00		7000.00		12000.00		AAAATGF	
	A001									
	C00016		Venkatpatti		Bangalore		Bangalore		India	
2	8000.00		11000.00		7000.00		12000.00		JRTVFDD	
	A007									
	C00011		Sundariya		Chennai		Chennai		India	
3	7000.00		11000.00		7000.00		11000.00		PPHGRTS	
	A010									
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
-+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
-+-----+-----+										

To get order number and order amount columns from orders table aliased as 'a' and customer name and working area columns from customer table aliased as 'b' after joining said two tables with the following condition -

1. order amount of orders table matches any of the opening amounts of customer table, the following SQL statement can be used:

SQL Code:

```
-- Selecting columns ord_num, ord_amount, cust_name, and  
working_area from tables orders and customer  
  
SELECT a.ord_num, a.ord_amount, b.cust_name, b.working_area  
  
-- Specifying the tables involved in the query and their aliases  
  
FROM orders a, customer b  
  
-- Filtering the rows where the ord_amount is between the  
opening_amt and opening_amt (redundant condition)  
  
WHERE a.ord_amount BETWEEN b.opening_amt AND b.opening_amt;
```

Copy

Explanation:

- This SQL query retrieves specific information from the tables orders and customer.
- It selects four columns: ord_num and ord_amount from the orders table (aliased as a), and cust_name and working_area from the customer table (aliased as b).
- The query performs an implicit join between the orders and customer tables, combining all rows from both tables.
- It then applies a condition using the WHERE clause, specifying that the ord_amount from the orders table must be between the opening_amt and opening_amt from the customer table.
- However, this condition is redundant as it always evaluates to true (any value is between itself and itself).
- The result will likely include all orders along with the names of customers and their working areas, but the condition doesn't filter the data as intended.

Output:

ORD_NUM	ORD_AMOUNT	CUST_NAME	WORKING_AREA
200110	3000	Micheal	New York
200101	3000	Micheal	New York
200108	4000	Cook	London
200119	4000	Cook	London
200113	4000	Cook	London
200108	4000	Karl	London
200119	4000	Karl	London
200113	4000	Karl	London

SQL NULL Values

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

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

Demo Database

Below is a selection from the [Customers](#) table used in the examples:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The IS NULL Operator

The IS NULL operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

The IS NOT NULL Operator

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

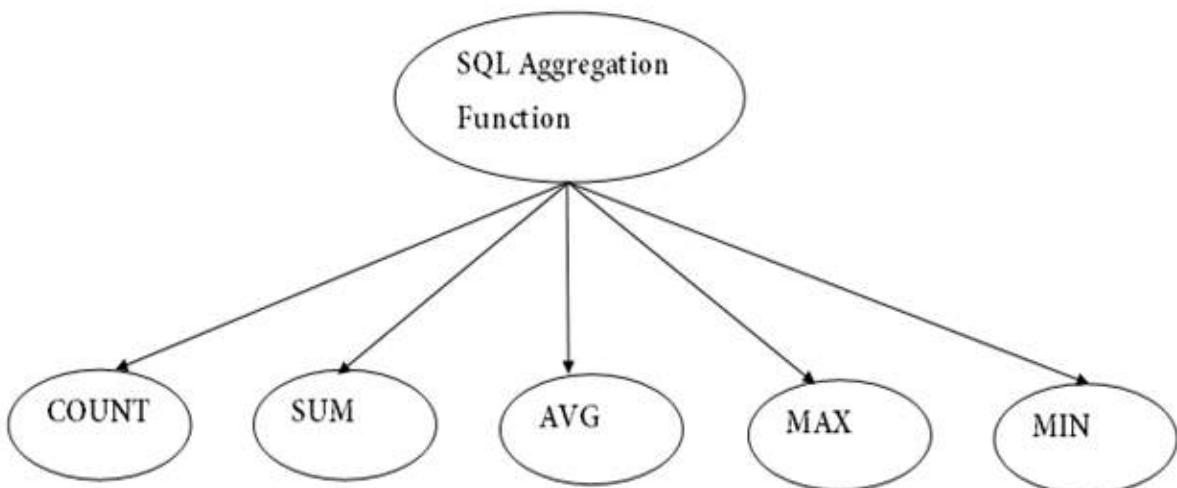
Example

```
SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;
```

SQL Aggregate Functions

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Types of SQL Aggregation Function



1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

1. COUNT(*)
2. or
3. COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;

Output:

10

Example: COUNT with WHERE

1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;
3. WHERE RATE>=20;

Output:

7

Example: COUNT() with DISTINCT

1. SELECT COUNT(DISTINCT COMPANY)
2. FROM PRODUCT_MAST;

Output:

3

Example: COUNT() with GROUP BY

1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY;

Output:

Com1	5
Com2	3
Com3	2

Example: COUNT() with HAVING

1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(*)>2;

Output:

Com1	5
Com2	3

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

1. SUM()
2. or
3. SUM([ALL|DISTINCT] expression)

Example: SUM()

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST;

Output:

Example: SUM() with WHERE

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>**3**;

Output:

320

Example: SUM() with GROUP BY

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>**3**
4. GROUP BY COMPANY;

Output:

Com1	150
Com2	170

Example: SUM() with HAVING

1. SELECT COMPANY, SUM(COST)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING SUM(COST)>=**170**;

Output:

Com1	335
Com3	170

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

1. AVG()
2. or

3. AVG([ALL|DISTINCT] expression)

Example:

1. SELECT AVG(COST)
2. FROM PRODUCT_MAST;

Output:

67.00

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

1. MAX()
2. or
3. MAX([ALL|DISTINCT] expression)

Example:

1. SELECT MAX(RATE)
2. FROM PRODUCT_MAST;

30

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

1. MIN()
2. or
3. MIN([ALL|DISTINCT] expression)

Example:

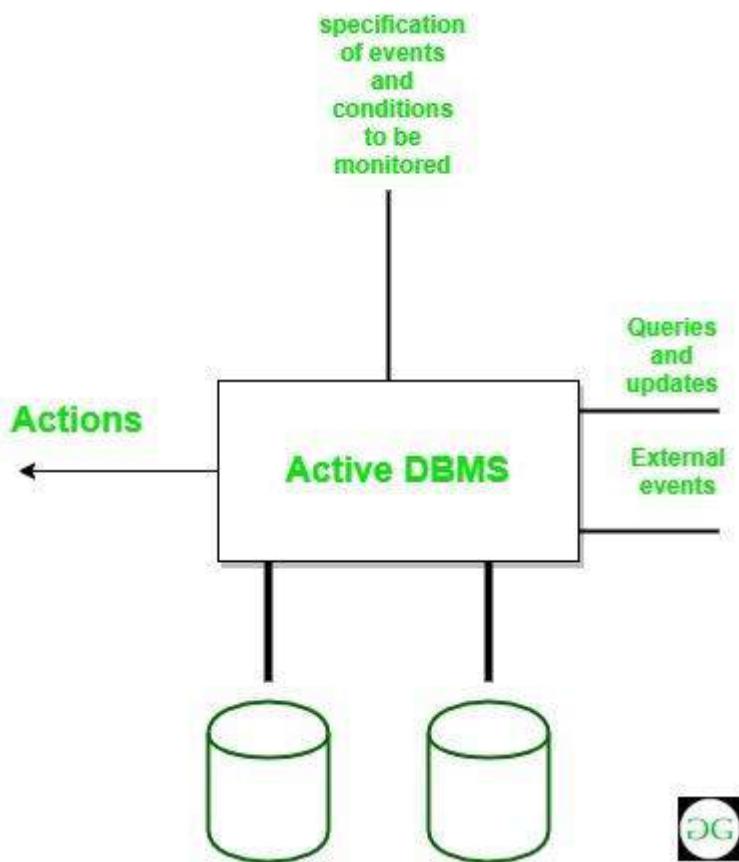
1. SELECT MIN(RATE)
2. FROM PRODUCT_MAST;

Output:

10

ACTIVE DATABASE

An active Database is a database consisting of a set of triggers. These databases are very difficult to be maintained because of the complexity that arises in understanding the effect of these triggers. In such database, DBMS initially verifies whether the particular trigger specified in the statement that modifies the database is activated or not, prior to executing the statement. If the trigger is active then DBMS executes the condition part and then executes the action part only if the specified condition is evaluated to true. It is possible to activate more than one trigger within a single statement. In such situation, DBMS processes each of the trigger randomly. The execution of an action part of a trigger may either activate other triggers or the same trigger that initialized this action. Such types of trigger that activates itself is called as 'recursive trigger'. The DBMS executes such chains of trigger in some pre-defined manner but it effects the concept of understanding.



Features of Active Database:

1. It possess all the concepts of a conventional database i.e. data modelling facilities, query language etc.
2. It supports all the functions of a traditional database like data definition, data manipulation, storage management etc.
3. It supports definition and management of ECA rules.
4. It detects event occurrence.
5. It must be able to evaluate conditions and to execute actions.
6. It means that it has to implement rule execution.

Examples of Active Databases:

1. Real-time Databases
2. In-Memory Databases
3. Transactional Databases
4. Time-series Databases

1. Real-time Databases:

Oracle TimesTen: A relational database that runs in memory and is intended for real-time applications that need response times of less than one millisecond.

VoltDB: A lightning-fast in-memory database for instantaneous analytics and data processing.

2. In-Memory Databases:

SAP HANA: A column-oriented, in-memory relational database management system for processing large amounts of data and real-time analytics.

MemSQL: Uses in-memory processing for real-time data insights, combining analytics and transactions on a single platform.

3. Transactional Databases:

MySQL Cluster: Offers automatic sharding and synchronous replication for high availability and real-time data access.

Microsoft SQL Server with Always On: High availability and disaster recovery are provided by Microsoft SQL Server with Always On, which enables real-time read access to replicated databases.

4. Time-series Databases:

InfluxDB: For time-stamped data, InfluxDB is designed to withstand heavy write and query loads. It is frequently utilized in IoT and monitoring applications.

Prometheus: A toolkit for alerting and monitoring that keeps track of time series data and is used to analyze and monitor systems in real time.

These databases and platforms support a variety of real-time data handling requirements, including high-throughput stream processing, low-latency transaction processing, and event-driven architectures.

Advantages :

1. Enhances traditional database functionalities with powerful rule processing capabilities.
2. Enable a uniform and centralized description of the business rules relevant to the information system.
3. Avoids redundancy of checking and repair operations.
4. Suitable platform for building large and efficient knowledge base and expert systems.

COMPLEX INTEGRITY CONSTRAINTS

In SQL, integrity constraints are rules that ensure data integrity in a database. They can be classified into two categories: simple and complex.

Simple integrity constraints include primary keys, unique constraints, and foreign keys. These constraints ensure that data in a table is unique and consistent, and they prevent certain operations that would violate these rules.

On the other hand, complex integrity constraints are more advanced rules that cannot be expressed using simple constraints. They are typically used to enforce business rules, such as limiting the range of values that can be entered into a column or ensuring that certain combinations of values are present in a table.

There are several ways to implement complex integrity constraints in SQL, including:

1. Check Constraints: Check constraints are used to restrict the range of values that can be entered into a column. For example, a check constraint can be used to ensure that a date column only contains dates in a certain range.

```
CREATE TABLE Employee (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    Salary DECIMAL(10,2),
    CONSTRAINT CHK_Age CHECK (Age >= 18 AND Age <= 65),
    CONSTRAINT CHK_Salary CHECK (Salary >= 0)
);
```

In this example, we are creating a table called "Employee" with columns for ID, Name, Age, and Salary. We have added two check constraints, one to ensure that the age is between 18 and 65, and another to ensure that the salary is greater than or equal to 0.

2. Assertions: Assertions are used to specify complex rules that cannot be expressed using simple constraints. They are typically used to enforce

business rules, such as ensuring that a customer's age is greater than a certain value.

```
CREATE TABLE Customer (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    Gender VARCHAR(10),
    CONSTRAINT CHK_Age CHECK (Age >= 18),
    CONSTRAINT CHK_Gender CHECK (Gender IN ('M', 'F')),
    CONSTRAINT CHK_Female_Age CHECK (Gender <> 'F' OR Age >= 21)
);
```

In this example, we are creating a table called "Customer" with columns for ID, Name, Age, and Gender. We have added three constraints: one to ensure that the age is greater than or equal to 18, another to ensure that the gender is either "M" or "F", and a third to ensure that if the gender is "F", the age is greater than or equal to 21.

3. Triggers:

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs.

For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated in simple words a trigger is a collection of [SQL](#) statements with particular names that are stored in system memory. It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. Every trigger has a table attached to it.

Because a trigger cannot be called directly, unlike a stored procedure, it is referred to as a special procedure. A trigger is automatically called whenever a data modification event against a table takes place, which is the main distinction between a trigger and a procedure. On the other hand, a stored procedure must be called directly.

The following are the key differences between triggers and stored procedures:

1. Triggers cannot be manually invoked or executed.
2. There is no chance that triggers will receive parameters.

3. A transaction cannot be committed or rolled back inside a trigger.

Syntax:

create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]

Explanation of Syntax

1. Create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. On [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each affected row.
6. [trigger_body]: This provides the operation to be performed as the trigger is fired

Why Do We Employ Triggers?

When we need to carry out some actions automatically in certain desirable scenarios, triggers will be useful. For instance, we need to be aware of the frequency and timing of changes to a table that is constantly changing. In such cases, we could create a trigger to insert the required data into a different table if the primary table underwent any changes.

Different Trigger Types in SQL Server

Two categories of triggers exist:

1. DDL Trigger
2. DML Trigger
3. Logon Triggers

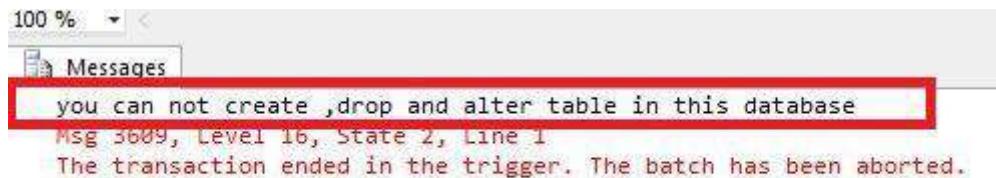
DDL Triggers

The Data Definition Language (DDL) command events such as Create_table, Create_view, drop_table, Drop_view, and Alter_table cause the DDL triggers to be activated.

SQL Server

```
create trigger safety
on database
for
create_table,alter_table,drop_table
as
print 'you can not create,drop and alter tab'
```

Output:



100 % ▾

Messages

you can not create ,drop and alter table in this database

Msg 3609, Level 16, State 2, Line 1

The transaction ended in the trigger. The batch has been aborted.

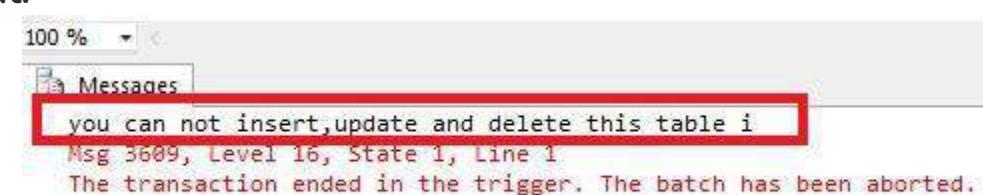
DML Triggers

The Data uses manipulation Language (DML) command events that begin with Insert, Update, and Delete set off the DML triggers. corresponding to insert_table, update_view, and delete_table.

SQL Server

```
create trigger deep
on emp
for
insert,update ,delete
as
print 'you can not insert,update and delete this table i'
rollback;
```

Output:



100 % ▾

Messages

you can not insert,update and delete this table i

Msg 3609, Level 16, State 1, Line 1

The transaction ended in the trigger. The batch has been aborted.

Logon Triggers

logon triggers are fires in response to a LOGON event. When a user session is created with a SQL Server instance after the authentication process of logging is finished but before establishing a user session, the LOGON event takes place. As a result, the PRINT statement messages and any errors generated by the trigger will all be visible in the SQL Server error log. Authentication errors prevent logon triggers from being used. These triggers can be used to track login activity or set a limit on the number of sessions that a given login can have in order to audit and manage server sessions.

How does SQL Server Show Trigger?

The show or list trigger is useful when we have many databases with many tables. This query is very useful when the table names are the same across multiple databases. We can view a list of every trigger available in the SQL Server by using the command below:

Syntax:

```
FROM    sys.triggers,    SELECT    name,    is_instead_of_trigger  
IF type = 'TR';
```

The SQL Server Management Studio makes it very simple to display or list all triggers that are available for any given table. The following steps will help us accomplish this:

Go to the **Databases** menu, select the desired database, and then expand it.

- Select the **Tables** menu and expand it.
- Select any specific table and expand it.

We will get various options here. When we choose the **Triggers** option, it displays all the triggers available in this table.

BEFORE and AFTER Trigger

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Example

Given Student Report Database, in which student marks assessment is recorded. In such a schema, create a trigger so that the total and percentage of specified marks are automatically inserted whenever a record is inserted. Here, a trigger will invoke before the record is inserted so BEFORE Tag can be used.

Suppose the Database Schema

Query

```
mysql>>desc Student;
```

Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto_increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
subj2	int(2)	YES		NULL	
subj3	int(2)	YES		NULL	
total	int(3)	YES		NULL	
per	int(3)	YES		NULL	

SQL Trigger to the problem statement.

```
CREATE TRIGGER stud_marks
BEFORE INSERT ON Student
FOR EACH ROW
SET NEW.total = NEW.subj1 + NEW.subj2 + NEW.subj3,
    NEW.per = (NEW.subj1 + NEW.subj2 + NEW.subj3) * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, the trigger will compute those two values and insert them with the entered values. i.e.

Output

Trigger	Event	Table	Timing	Created	Status
stud_marks	BEFORE	Student	BEFORE INSERT	2023-05-02 00:00:00	ACTIVE

In this way, triggers can be created and executed in the databases.

Advantage of Triggers

The benefits of using triggers in SQL Server include the following:

1. Database object rules are established by triggers, which cause changes to be undone if they are not met.
2. The trigger will examine the data and, if necessary, make changes.
3. We can enforce data integrity thanks to triggers.
4. Data is validated using triggers before being inserted or updated.
5. Triggers assist us in maintaining a records log.
6. Due to the fact that they do not need to be compiled each time they are run, triggers improve the performance of SQL queries.
7. The client-side code is reduced by triggers, saving time and labor.
8. Trigger maintenance is simple.

Disadvantage of Triggers

The drawbacks of using triggers in SQL Server include the following:

1. Only triggers permit the use of extended validations.
2. Automatic triggers are used, and the user is unaware of when they are being executed. Consequently, it is difficult to troubleshoot issues that arise in the database layer.
3. The database server's overhead may increase as a result of triggers.
4. In a single CREATE TRIGGER statement, we can specify the same trigger action for multiple user actions, such as INSERT and UPDATE.
5. Only the current database is available for creating triggers, but they can still make references to objects outside the database.

Triggers are special procedures that are executed automatically in response to certain events, such as an insert or update operation on a table. Triggers can be used to implement complex business rules that cannot be expressed using simple constraints.

CREATE TABLE Order (

 ID INT PRIMARY KEY,

 CustomerID INT,

 OrderDate DATE,

 Amount DECIMAL(10,2),

 CONSTRAINT FK_Customer_Order FOREIGN KEY (CustomerID)

```

REFERENCES Customer(ID)
);

CREATE TRIGGER TR_Order_Check_Amount
BEFORE INSERT ON Order
FOR EACH ROW
BEGIN
    IF NEW.Amount <= 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Amount must be
greater than zero.';
    END IF;
END;

```

In this example, we are creating a table called "Order" with columns for ID, CustomerID, OrderDate, and Amount. We have added a foreign key constraint to ensure that the CustomerID in the Order table matches the ID in the Customer table. We have also created a trigger called "TR_Order_Check_Amount" that will be executed before an insert operation on the Order table. The trigger checks if the amount being inserted is greater than zero. If it is not, an error message will be generated and the insert operation will be cancelled.

These are just a few examples of how complex integrity constraints can be implemented in SQL to ensure the accuracy and consistency of data in a database.

Overall, complex integrity constraints are essential for ensuring the integrity and consistency of data in a database. They allow developers to enforce complex business rules and prevent data inconsistencies, ensuring that the data in the database remains accurate and trustworthy.

4. Stored procedure:

A stored procedure in SQL is a group of SQL queries that can be saved and reused multiple times. It is very useful as it reduces the need for rewriting SQL queries. It enhances efficiency, reusability, and security in database management.

Users can also pass parameters to stored procedures so that the stored procedure can act on the passed parameter values.

Stored Procedures are created to perform one or more [DML](#) operations on the Database. It is nothing but a group of SQL statements that accepts some input in the form of parameters, performs some task, and may or may not return a value.

Syntax

Two important syntaxes for using stored procedures in SQL are:

Syntax to Create a Stored Procedure

```
CREATE PROCEDURE procedure_name  
(parameter1 data_type, parameter2 data_type, ...)  
AS  
BEGIN  
    — SQL statements to be executed  
END
```

Syntax to Execute the Stored Procedure

```
EXEC procedure_name parameter1_value, parameter2_value,  
..
```

Parameter

The most important part is the parameters. Parameters are used to pass values to the Procedure. There are different types of parameters, which are as follows:

1. BEGIN: This is what directly executes or we can say that it is an executable part.
2. END: Up to this, the code will get executed.

SQL Stored Procedure Example

Let's look at an example of Stored Procedure in SQL to understand it better.

Imagine a database named "SampleDB", a table named "Customers" with some sample data, and a stored procedure named "GetCustomersByCountry".

The stored procedure takes the parameter "Country" and returns a list of customers from the "Customers" table that matches the specified country. Finally, the stored procedure is executed with the parameter "Sri Lanka" to retrieve the list of customers from Sri Lanka.

Query:

```
-- Create a new database named "SampleDB"  
CREATE DATABASE SampleDB;
```

```
-- Switch to the new database  
USE SampleDB;
```

```
-- Create a new table named "Customers"  
CREATE TABLE Customers (  
    CustomerID INT PRIMARY KEY,  
    CustomerName VARCHAR(50),  
    ContactName VARCHAR(50),  
    Country VARCHAR(50)
```

);

```
-- Insert some sample data into the Customers table
INSERT INTO Customers (CustomerID, CustomerName,
ContactName, Country)
VALUES (1, 'Shubham', 'Thakur', 'India'),
(2, 'Aman ', 'Chopra', 'Australia'),
(3, 'Naveen', 'Tulasi', 'Sri lanka'),
(4, 'Aditya', 'Arpan', 'Austria'),
(5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain');
```

```
-- Create a stored procedure named
"GetCustomersByCountry"
CREATE PROCEDURE GetCustomersByCountry
    @Country VARCHAR(50)
AS
BEGIN
    SELECT CustomerName, ContactName
    FROM Customers
    WHERE Country = @Country;
END;
```

```
-- Execute the stored procedure with parameter "Sri lanka"
EXEC GetCustomersByCountry @Country = 'Sri lanka';
```

Note: You will need to make sure that the user account has the necessary privileges to create a database. You can try logging in as a different user with administrative privileges or contact the database administrator to grant the necessary privileges to your user account. If you are using a cloud-based

database service, make sure that you have correctly configured the user account and its permissions.

Output:

CustomerName	Contact Name
Naveen	Tulasi

Important Points About SQL Stored Procedures

- *A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.*
- *Stored procedures allow for code that is used repeatedly to be saved on the database and run from there, rather than from the client. This provides a more modular approach to database design.*
- *Since stored procedures are compiled and stored in the database, they are highly efficient. SQL Server compiles each stored procedure once and then reutilizes the execution plan. This leads to tremendous performance boosts when stored procedures are called repeatedly.*
- *Stored procedures provide better security to your data. Users can execute a stored procedure without needing to execute any of the statements directly. Therefore, a user can be granted permission to execute a stored procedure without having any permissions on the underlying tables.*

- *Stored procedures can reduce network traffic and latency, boosting application performance. A single call to a stored procedure can execute many statements.*
- *Stored procedures have better support for error handling.*
- *Stored procedures can be used to provide advanced database functionality, such as modifying data in tables, and encapsulating these changes within database transactions.*

Unit-V

Advanced Databases

Parallel Databases

Companies need to handle huge amount of data with high data transfer rate. The client server and centralized system is not much efficient. The need to improve the efficiency gave birth to the concept of Parallel Databases.

Parallel database system improves performance of data processing using multiple resources in parallel, like multiple CPU and disks are used parallelly.

It also performs many parallelization operations like, data loading and query processing.

Goals of Parallel Databases

The concept of Parallel Database was built with a goal to:

Improve performance:

The performance of the system can be improved by connecting multiple CPU and disks in parallel. Many small processors can also be connected in parallel.

Improve availability of data:

Data can be copied to multiple locations to improve the availability of data.

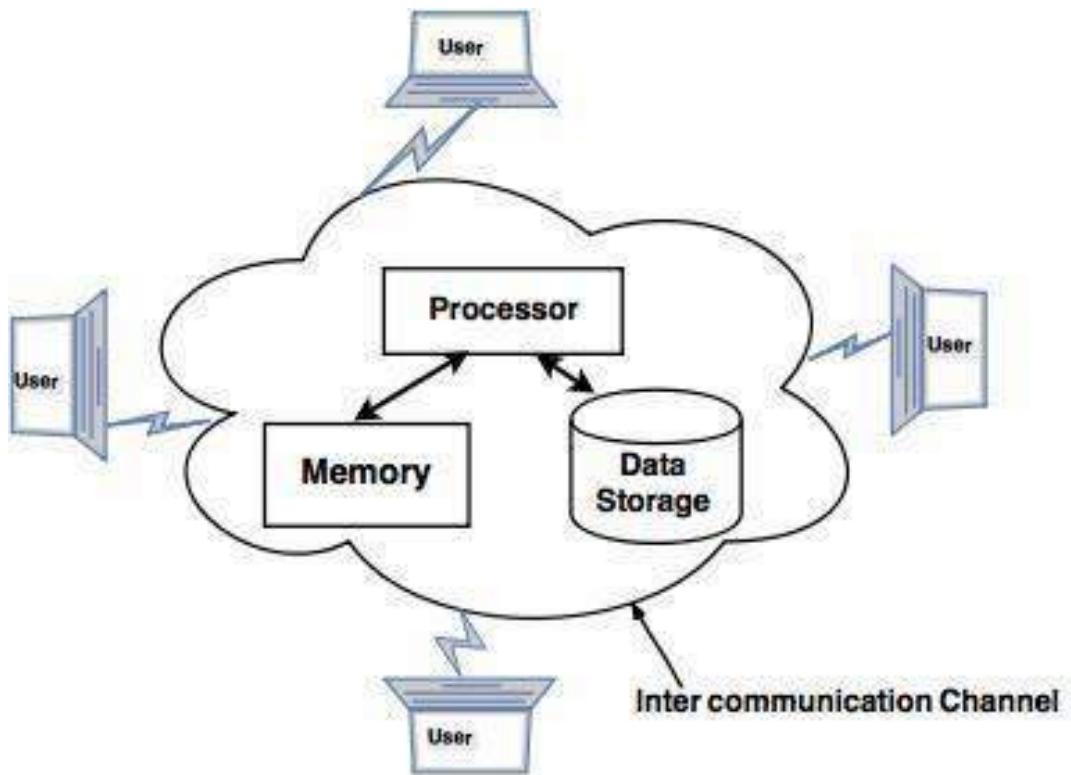
For example: if a module contains a relation (table in database) which is unavailable then it is important to make it available from another module.

Improve reliability:

Reliability of system is improved with completeness, accuracy and availability of data.

Provide distributed access of data:

Companies having many branches in multiple cities can access data with the help of parallel database system.



Parallel database system

Multimedia Database

Multimedia database is the collection of interrelated multimedia data that includes text, graphics (sketches, drawings), images, animations, video, audio etc and have vast amounts of

multisource multimedia data. The framework that manages different types of multimedia data which can be stored, delivered and utilized in different ways is known as multimedia database management system. There are three classes of the multimedia database which includes static media, dynamic media and dimensional media.

Content of Multimedia Database management system:

1. **Media data** – The actual data representing an object.
2. **Media format data** – Information such as sampling rate, resolution, encoding scheme etc. about the format of the media data after it goes through the acquisition, processing and encoding phase.
3. **Media keyword data** – Keywords description relating to the generation of data. It is also known as content descriptive data. Example: date, time and place of recording.
4. **Media feature data** – Content dependent data such as the distribution of colors, kinds of texture and different shapes present in data.

Types of multimedia applications based on data management characteristic are :

1. **Repository applications** – A Large amount of multimedia data as well as meta-data(Media format date, Media keyword data, Media feature data) that is stored for retrieval purpose, e.g., Repository of satellite images, engineering drawings, radiology scanned pictures.
2. **Presentation applications** – They involve delivery of multimedia data subject to temporal constraint. Optimal viewing or listening requires DBMS to deliver data at certain rate offering the quality of service above a certain threshold. Here data is processed as it is delivered. Example: Annotating of video and audio data, real-time editing analysis.
3. **Collaborative work using multimedia information** – It involves executing a complex task by merging drawings, changing notifications. Example: Intelligent healthcare network.

There are still many challenges to multimedia databases, some of which are :

1. **Modelling** – Working in this area can improve database versus information retrieval techniques thus, documents constitute a specialized area and deserve special consideration.
2. **Design** – The conceptual, logical and physical design of multimedia databases has not yet been addressed fully as performance and tuning issues at each level are far more complex as they consist of a variety of formats like JPEG, GIF, PNG, MPEG which is not easy to convert from one form to another.
3. **Storage** – Storage of multimedia database on any standard disk presents the problem of representation, compression, mapping to device hierarchies, archiving and buffering during input-output operation. In DBMS, a "BLOB"(Binary Large Object) facility allows untyped bitmaps to be stored and retrieved.
4. **Performance** – For an application involving video playback or audio-video synchronization, physical limitations dominate. The use of parallel processing may alleviate some problems but such techniques are not yet fully developed. Apart from this multimedia database consume a lot of processing time as well as bandwidth.
5. **Queries and retrieval** –For multimedia data like images, video, audio accessing data through query opens up many issues like efficient query formulation, query execution and optimization which need to be worked upon.

Areas where multimedia database is applied are :

- **Documents and record management :** Industries and businesses that keep detailed records and variety of documents. Example: Insurance claim record.
- **Knowledge dissemination :** Multimedia database is a very effective tool for knowledge dissemination in terms of providing several resources. Example: Electronic books.
- **Education and training :** Computer-aided learning materials can be designed using multimedia sources which are nowadays very popular sources of learning. Example: Digital libraries.
- Marketing, advertising, retailing, entertainment and travel. Example: a virtual tour of cities.
- **Real-time control and monitoring :** Coupled with active database technology, multimedia presentation of information can be very effective means for monitoring and controlling complex tasks Example: Manufacturing operation control.

Mobile Databases

Mobile databases are separate from the main database and can easily be transported to various places. Even though they are not connected to the main database, they can still communicate with the database to share and exchange data.

The mobile database includes the following components:

1. The main system database that stores all the data and is linked to the mobile database.
2. The mobile database that allows users to view information even while on the move. It shares information with the main database.
3. The device that uses the mobile database to access data. This device can be a mobile phone, laptop etc.
4. A communication link that allows the transfer of data between the mobile database and the main database.

Advantages of Mobile Database

Some advantages of mobile databases are:

1. The data in a database can be accessed from anywhere using a mobile database. It provides wireless database access.
2. The database systems are synchronized using mobile databases and multiple users can access the data with seamless delivery process.
3. Mobile databases require very little support and maintenance.
4. The mobile database can be synchronized with multiple devices such as mobiles, computer devices, laptops etc.

Disadvantages of Mobile Databases

Some disadvantages of mobile databases are:

1. The mobile data is less secure than data that is stored in a conventional stationary database. This presents a security hazard.

2. The mobile unit that houses a mobile database may frequently lose power because of limited battery. This should not lead to loss of data in database.

Web-Database

The Web-based database management system is one of the essential parts of DBMS and is used to store web application data. A web-based Database management system is used to handle those databases that are having data regarding E-commerce, E-business, blogs, e-mail, and other online applications.

While many DBMS sellers are working for providing a proprietary database for connectivity solutions with the Web, the majority of the organizations necessitate a more general way out to prevent them from being tied into a single technology. Here are the lists of some of the most significant necessities for the database integration applications within the Web. These requirements are standards and not fully attainable at present. There is no ranking of orders, and so the requirements are as follows:

- The ability and right to use valuable corporate data in a fully secured manner.
- Provides data and vendor's autonomous connectivity that allows freedom of choice in selecting the DBMS for present and future use.
- The capability to interface to the database, independent of any proprietary Web browser and/or Web server.
- A connectivity solution that takes benefit of all the features of an organization's DBMS.
- An open-architectural structure that allows interoperability with a variety of systems and technologies; such as:
 - Different types of Web servers
 - Microsoft's Distributed Common Object Model (DCOM) / Common Object Model (COM)
 - CORBA / IIOP
 - Java / RMI which is Remote Method Invocation
 - XML (Extensible Markup Language)
 - Various Web services (SOAP, UDDI, etc.)
- A cost-reducing way which allows for scalability, development, and changes in strategic directions and helps lessen the costs of developing and maintaining those applications
- Provides support for transactions that span multiple HTTP requests.
- Gives minimal administration overhead.

Multimedia Database:

The multimedia databases are used to store multimedia data such as images, animation, audio, video along with text. This data is stored in the form of multiple file types like .txt(text), .jpg(images), .swf(videos), .mp3(audio) etc.

Contents of the Multimedia Database

The multimedia database stored the multimedia data and information related to it. This is given in detail as follows:

Media data

This is the multimedia data that is stored in the database such as images, videos, audios, animation etc.

Media format data

The Media format data contains the formatting information related to the media data such as sampling rate, frame rate, encoding scheme etc.

Media keyword data

This contains the keyword data related to the media in the database. For an image the keyword data can be date and time of the image, description of the image etc.

Media feature data

The Media feature data describes the features of the media data. For an image, feature data can be colours of the image, textures in the image etc.

Challenges of Multimedia Database

There are many challenges to implement a multimedia database. Some of these are:

1. Multimedia databases contains data in a large type of formats such as .txt(text), .jpg(images), .swf(videos), .mp3(audio) etc. It is difficult to convert one type of data format to another.
2. The multimedia database requires a large size as the multimedia data is quite large and needs to be stored successfully in the database.
3. It takes a lot of time to process multimedia data so multimedia database is slow.

OLTP versus OLAP

Definition of OLTP

OLTP is an **Online Transaction Processing system**. The main focus of OLTP system is to record the current **Update, Insertion and Deletion** while transaction. The OLTP queries are **simpler** and **short** and hence require **less time in processing**, and also requires **less space**.

OLTP database gets **updated frequently**. It may happen that a transaction in OLTP fails in middle, which may effect **data integrity**. So, it has to take special care of data integrity. OLTP database has **normalized tables (3NF)**.

The best example for OLTP system is an **ATM**, in which using short transactions we modify the status of our account. OLTP system becomes the source of data for OLAP.

Definition of OLAP

OLAP is an **Online Analytical Processing system**. OLAP database stores historical data that has been inputted by OLTP. It allows a user to view different summaries of multi-dimensional data. Using OLAP, you can extract information from a large database and analyze it for decision making.

OLAP also allow a user to execute **complex queries** to extract multidimensional data. In OLTP even if the transaction fails in middle it will not harm data integrity as the user use OLAP system to retrieve data from a large database to analyze. Simply the user can fire the query again and extract the data for analysis.

OLTP and OLAP both are the online processing systems. OLTP is a transactional processing while OLAP is an analytical processing system.

OLTP is a system that manages transaction-oriented applications on the internet for example, ATM. OLAP is an online system that reports to multidimensional analytical queries like financial reporting, forecasting, etc.

The basic difference between OLTP and OLAP is that OLTP is an online database modifying system, whereas, OLAP is an online database query answering system.

There are some other differences between OLTP and OLAP which I have explained using the comparison chart shown below.

Comparison Chart

Basis for Comparison	OLTP	OLAP
Basic	It is an online transactional system and manages database modification.	It is an online data retrieving and data analysis system.
Focus	Insert, Update, Delete information from the database.	Extract data for analyzing that helps in decision making.
Data	OLTP and its transactions are the original source of data.	Different OLTPs database becomes the source of data for OLAP.
Transaction	OLTP has short transactions.	OLAP has long transactions.
Time	The processing time of a transaction is comparatively less in OLTP.	The processing time of a transaction is comparatively more in OLAP.
Queries	Simpler queries.	Complex queries.

Normalization	Tables in OLTP database are normalized (3NF).	Tables in OLAP database are not normalized.
Integrity	OLTP database must maintain data integrity constraint.	OLAP database does not get frequently modified. Hence, data integrity is not affected.

NoSQL database

NoSQL is a non-relational DBMS, that does not require a fixed schema, avoids joins, and is easy to scale. The purpose of using a No SQL database is for distributed data stores with humongous data storage needs. No SQL is used for big data and real-time web apps. For example, companies like Twitter, Face book, Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.

Advantages of NoSQL:

There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

1. High scalability –

NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.

2. High availability –

Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.

Disadvantages of NoSQL:

NoSQL has the following disadvantages.

1. Narrow focus –

NoSQL databases have very narrow focus as it is mainly designed for storage but it

provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.

2. **Open-source –**
NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words two database systems are likely to be unequal.
3. **Management challenge –**
The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
4. **GUI is not available –**
GUI mode tools to access the database is not flexibly available in the market.
5. **Backup –**
Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
6. **Large document size –**
Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

