Database concepts

- Concept of DBMS, RDBMS.
- Data Models, Concept of DBA, Database Users.
- ER Model & Diagram, Database Schema.
- Designing Database using Normalization Rules.
- Various data types Data integrity, DDL DML and DCL statements.
- Enforcing Primary key and foreign key.
- Adding Indices.

Concept of DBMS, RDBMS.

What is Database?

A database is a compilation of interconnected data that enables efficient operations such as data retrieval, insertion, and deletion.

It serves as a means to systematically arrange information, often in the format of tables, schemas, views, and reports.

For instance, consider a college database. This database arranges information concerning administrators, staff members, students, and faculty in a coherent manner. By utilizing this database, you can seamlessly retrieve, add, and remove information as needed.

Concept of DBMS and RDBMS

Database Management System

A Database Management System (DBMS) is software designed for the administration of databases. Widely used commercial databases like MySQL and Oracle serve various applications. The DBMS offers an interface to execute diverse tasks, such as crafting databases, storing data, data updates, and table creation.

This system ensures database security and protection, along with maintaining data consistency when multiple users are involved.

The DBMS empowers users to undertake the following tasks:

1. **Database Creation:** DBMS enables the creation of new databases to organize and store data efficiently.

2. **Data Storage:** It provides a platform for storing a wide range of data types, ensuring data integrity and accessibility.

3. **Data Retrieval:** Users can effortlessly retrieve specific information from the database using queries and search operations.

4. **Data Update and Modification:** DBMS facilitates the modification and updating of existing data, ensuring accuracy and relevance.

5. **Table Creation:** Users can define and create tables with specific structures to store data in a structured manner.

6. **Data Integrity:** It enforces rules and constraints to maintain the integrity and consistency of the data.

7. **Data Security:** DBMS offers security features to control access to the data, protecting it from unauthorized users.

8. **Data Backup and Recovery:** Users can perform data backups regularly and recover data in case of system failures or errors.

9. **Indexing:** DBMS allows the creation of indexes, enhancing data retrieval performance.

10. **Query Optimization:** It optimizes queries to improve the efficiency of data retrieval operations.

11. **Concurrency Control:** In multi-user environments, DBMS ensures that multiple users can access and modify data simultaneously without conflicts.

12. **Transaction Management:** DBMS supports transactions, ensuring that a series of operations are completed successfully or not at all.

13. **Data Reporting:** Users can generate reports and analyze data to extract valuable insights.

14. **User Management:** DBMS enables the management of user roles, permissions, and access levels to maintain data security.

15. **Data Relationships:** It allows users to establish relationships between different tables, facilitating complex data retrieval and analysis.

16. **Data Validation:** DBMS enforces data validation rules to ensure that entered data meets specified criteria.

17. **Data Sharing:** It enables data sharing across different applications and users while maintaining data integrity.

18. **Data Migration:** Users can transfer data between different databases or systems using DBMS tools.

19. **Data Archiving:** Older or less frequently used data can be archived to free up space while retaining accessibility.

20. **Data Auditing:** DBMS tracks changes and activities related to the database for auditing and compliance purposes.

Characteristics of DBMS

A Database Management System (DBMS) is a software application that facilitates the creation, maintenance, and manipulation of databases. It acts as an intermediary between users or applications and the actual physical database, providing an organized and controlled environment for storing and retrieving data. Here are some key characteristics of a DBMS:

1. **Data Abstraction:** A DBMS provides a level of abstraction that hides the complex underlying details of how data is stored and managed. This allows users and applications to interact with the database without needing to understand its internal workings.

2. **Data Integrity:** DBMS systems enforce data integrity rules to ensure that the data stored in the database remains accurate and consistent. This involves maintaining constraints, such as uniqueness, referential integrity, and data validation, to prevent incorrect or conflicting data from being stored.

3. **Data Security:** DBMS systems offer various security mechanisms to control access to the database. This includes user authentication, authorization, and access control, ensuring that only authorized users can perform specific actions on the data.

4. **Data Independence:** DBMS systems provide a separation between the logical structure of the database and its physical storage. This means that changes to the physical storage (such as moving to a different storage device) can be made without affecting the way users and applications access the data.

5. **Data Consistency:** A DBMS ensures that data remains consistent even when multiple users or applications are accessing and modifying it simultaneously. Techniques like transactions and locking mechanisms are used to maintain data consistency.

6. **Query Language:** DBMS systems provide a structured query language (SQL) that allows users to interact with the database using standardized commands. SQL enables users to retrieve, manipulate, and manage data stored in the database.

7. **Data Redundancy Elimination:** DBMS systems help in reducing data redundancy by providing features like normalization. This minimizes data duplication and improves data consistency.

8. **Concurrent Access and Transaction Management:** DBMS systems handle concurrent access to the database by multiple users or applications. Transaction management ensures that a series of database operations are treated as a single, indivisible unit, ensuring data integrity and consistency.

9. **Backup and Recovery:** DBMS systems offer mechanisms for data backup and recovery. Regular backups help prevent data loss in case of hardware failures, software errors, or other unexpected events.

10. **Scalability:** A well-designed DBMS can be scaled to accommodate increasing amounts of data and growing user loads. This can involve adding more hardware resources or optimizing the database structure.

11. **Data Relationships:** DBMS systems support the establishment and management of relationships between different sets of data, allowing for the creation of complex data structures and efficient retrieval of related information.

12. **Performance Optimization:** DBMS systems often include query optimization techniques to improve the performance of database operations. This involves choosing the most efficient execution plans for complex queries.

13. **Data Dictionary:** A data dictionary is a part of the DBMS that stores metadata, which includes information about the structure of the database, data types, relationships, and constraints. This helps users and applications understand the database's schema.

Advantages of DBMS

Manages Data Redundancy: By centralizing data within a single database file, DBMS effectively controls redundancy in recorded information.

Facilitates Data Sharing: Authorized users within an organization can seamlessly share data among multiple individuals through the DBMS.

Simplified Maintenance: The centralized nature of the database system makes maintenance notably simpler and more manageable.

Time Efficiency: DBMS expedites development processes and diminishes maintenance demands, leading to time savings.

Enables Backup: Incorporating backup and recovery subsystems, DBMS automatically backs up data in the event of hardware or software failures and facilitates data restoration as needed.

Supports Multiple User Interfaces: DBMS offers various user interfaces, including graphical user interfaces and application program interfaces, catering to diverse user preferences.

Disadvantages of DBMS

Expense for Hardware and Software: Running DBMS software demands robust data processing capabilities and substantial memory capacity, incurring costs for hardware upgrades.

Space Consumption: DBMS occupies considerable disk space and memory to ensure efficient operations.

Increased Complexity: Implementing a database system introduces added complexity and prerequisites.

Heightened Vulnerability to Failure: Database failures have a substantial impact, particularly in organizations where all data resides within a single database. Incidents like power outages or database corruption could lead to permanent data loss.

Relational Database Management System

RDBMS stands for *Relational Database Management System.*

RDBMS is an abbreviation for Relational Database Management Systems.

It serves as software that enables the creation, modification, and maintenance of relational databases.

A relational database is a type of system for storing and retrieving data presented in a structured table layout composed of rows and columns.

It's a specific component of Database Management Systems (DBMS) conceptualized by E.F. Codd in the 1970s. The fundamental principles of relational DBMS form the basis for prominent database systems like SQL, MySQL, and Oracle.

How it works

- Information is portrayed in the form of rows known as tuples within RDBMS.
- The prevailing choice for databases is the relational type. It comprises numerous tables, each endowed with its unique primary key.
- The structured assembly of tables facilitates seamless data retrieval within RDBMS.

table/Relation

All contents within a relational database are organized into relations. RDBMS databases utilize tables for data storage. A table constitutes an assembly of interconnected data elements, arranged in rows and columns. These tables symbolize real-world entities, like individuals, locations, or occurrences, for which data is accumulated. The systematic arrangement of data within a relational table embodies the conceptual portrayal of the database.

Properties of a Relation:

- Every relation within the database possesses an exclusive name for distinct identification.
- Within a relation, duplication of tuples is prohibited.
- Tuples within a relation are unordered.
- All attributes contained in a relation are indivisible; each cell comprises a singular value.
- A table serves as a fundamental illustration of data storage within RDBMS

Benefits(Advantages)

1. **Ease of Management:** Independent manipulation of tables simplifies database management without impacting others.

2. **Enhanced Security:** Multiple layers of security ensure controlled data access and sharing.

3. **Flexibility:** Centralized data updating prevents the need for modifications across various files. Expanding the database to include more records is straightforward, ensuring scalability. SQL queries can be applied easily.

4. **User Support:** RDBMS accommodates multiple users through a client-side architecture.

5. **Efficient Data Handling:**

   - Rapid data retrieval due to the relational design.
   - Keys, indexes, and normalization minimize data redundancy.
   - ACID properties ensure data consistency during transactions.

6. **Large Data Storage and Retrieval:** RDBMS facilitates handling vast data volumes.

7. **Effortless Data Handling:**

   - Swifter data fetching resulting from relational structure.
   - Keys, indexes, and normalization principles avert data redundancy.
   - Data consistency maintained through ACID properties in transactions.

8. **Fault Tolerance:** Database replication permits simultaneous access and aids system recovery during crises such as power outages or abrupt shutdowns.

Drawbacks(Disadvantages)

1. **High Costs and Infrastructure:** RDBMS demand substantial investments in terms of both expenses and infrastructure to establish and maintain their operations.

2. **Scalability Challenges:** Expanding data necessitates additional servers, increased power, and memory resources, which can be complex and costly.

3. **Complexity:** Large datasets can result in intricate relationships that might impede comprehension and even decrease performance.

4. **Structured Limits:** Relational databases have predefined limits on fields or columns, which could lead to data truncation or loss.

Data Models, Concept of DBA, Database Users.

Data Types

Data types define the kind of data that can be stored in a column of a database table. Different database management systems (DBMS) might support slightly different sets of data types, but here are some common categories:

1. Numeric Types

- ***Integer*:** Whole numbers (e.g., 1, -5, 100).
- ***Float/Double:*** Approximate decimal numbers with a specified precision (e.g., 3.14, -0.001).

2. Character Strings:

- ***Char:*** Fixed-length character string (e.g., 'Hello').
- ***Varchar:*** Variable-length character string (e.g., 'StudentName').

3.Date and Time Types

- ***Date:*** Represents a date (e.g., '2023-08-31').
- ***Time:*** Represents a time (e.g., '15:30:00').
- ***Timestamp:*** Represents a date and time (e.g., '2023-08-31 15:30:00').

4. Boolean Type

- ***Boolean:*** Represents true or false values.

5.Binary Types

- ***Blob (Binary Large Object):*** Stores binary data, such as images or files.

6. **Enumeration Types:**

- ***Enum:*** Represents a predefined set of values (e.g., 'red', 'green', 'blue').

7. **Composite Types:**

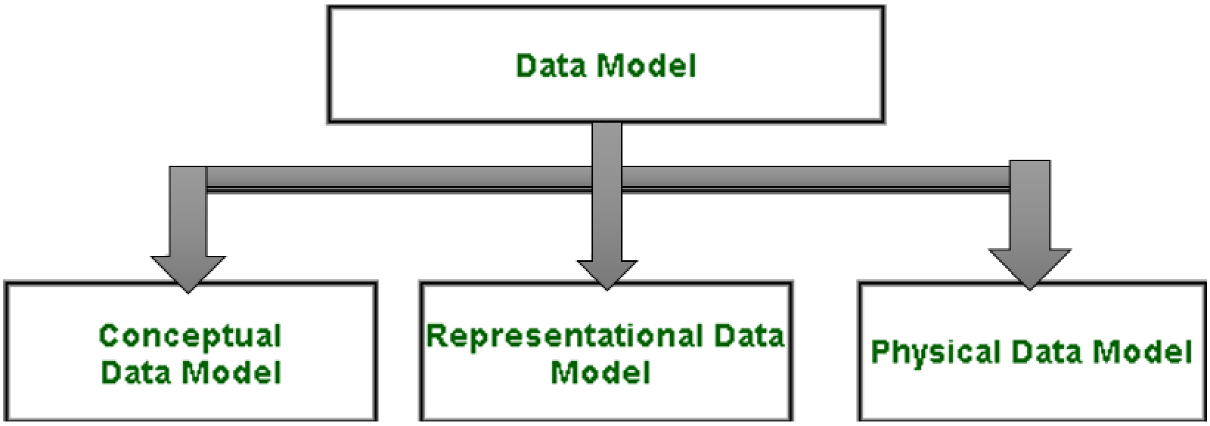
- ***JSON or XML:*** Stores structured data in JSON or XML format.

Data Models

In the realm of Database Management Systems (DBMS), a data model refers to a set of tools designed to condense the database's description. Data models offer a clear representation of data, aiding in the creation of an effective database. They guide us from conceptualizing data design to its accurate implementation.

Types of Relational Models

1. Conceptual Data Model
2. Representational Data Model
3. Physical Data Model



1. Conceptual Data Model:

The conceptual data model presents a broad overview of the database, focusing on high-level aspects. It serves to comprehend the database's necessities and demands. This model is pivotal during requirement gathering, preceding the actual database design by Database Designers. A prominent example of this approach is the entity/relationship (ER) model. The ER model delves into entities, relationships, and attributes, providing a foundation for database designers. Moreover, this concept facilitates discussions with non-technical stakeholders, aiding in understanding and addressing their requirements.

1. Entity-Relationship Model (ER Model): The ER Model stands as a top-tier data model employed to outline data and their interconnections. Essentially, it serves as a conceptual blueprint for databases, enabling an uncomplicated depiction of data perspectives.

Components of ER Model:

1. **Entity:** An entity represents a real-world object, whether it's a person, place, concept, or item. In an ER Diagram, entities are depicted using rectangles.

2. **Attributes:** Attributes provide descriptions or characteristics of an entity. These can include properties like name, age, roll number, or any other relevant information. In an ER Diagram, attributes are symbolized using ovals or ellipses.

3. **Relationship:** Relationships define connections between various entities. They represent how different entities are related to each other in the real world. In an ER Diagram, relationships are indicated using diamonds or rhombuses.

2. Representational Data Model

This type of data model is used to represent only the logical part of the database and does not represent the physical structure of the database. The representational data model allows us to focus primarily, on the design part of the database. A popular representational model is a Relational model. The relational Model consists of Relational Algebra and Relational Calculus. In the Relational Model, we basically use tables to represent our data and the relationships between them.

3. Physical Data Model

The Physical Data Model serves the practical implementation of the Relational Data Model. In the end, all database data is physically stored on secondary storage devices like disks and tapes. This storage occurs in the form of files, records, and other specific data structures. This model encompasses details about file formats, database structure, external data structure presence, and their interrelationships.

Advantages of Data Models:

1. **Accurate Data Representation:** Data models ensure precise and structured representation of data, enhancing its clarity and organization.

2. **Data Integrity and Minimized Redundancy:** Data models aid in identifying missing data and reducing redundancy by maintaining consistent and non-repetitive data entries.

3. **Enhanced Data Security:** Data models contribute to improved data security measures, safeguarding information against unauthorized access or tampering.

4. **Effective Physical Database Creation:** A well-designed data model should offer sufficient detail to serve as a foundation for constructing the physical database, ensuring alignment between the conceptual and physical levels.

5. **Relationship Definition:** Data models support the delineation of relationships between tables, as well as the establishment of primary and foreign keys. This is crucial for maintaining data consistency and integrity.

6. **Stored Procedure Definition:** The information encapsulated in a data model can be utilized to define stored procedures, enabling efficient data manipulation and retrieval.

Disadvantages of Data Models:

1. **Complexity with Large Databases:** In the context of extensive databases, comprehending the intricacies of the data model can become challenging and overwhelming.

2. **SQL Proficiency Required:** Proper knowledge of SQL is essential to effectively utilize and manipulate physical models. This requirement can be a barrier for users lacking SQL expertise.

3. **Impact of Structural Changes:** Even minor alterations to the data model's structure can necessitate significant modifications throughout the associated application, potentially causing disruptions and increased maintenance efforts.

4. **Lack of Standard Data Manipulation Language:** Unlike SQL, which is a standardized language for database manipulation, data models don't inherently provide a universally accepted data manipulation language.

5. **Requirement for Knowledge of Physical Data Storage:** Creating a data model necessitates a comprehensive understanding of the characteristics of how data is physically stored, which can present a learning curve and barrier for some users.

DBA(Database Administrator)

A Database Administrator (DBA) is an individual responsible for overseeing, maintaining, coordinating, and operating a database management system. Their primary role involves managing, securing, and ensuring the smooth functioning of database systems. They hold the authority to grant database access, coordinate tasks, plan for capacity, install and monitor software, and procure hardware resources as required. Their responsibilities encompass a wide range of tasks, including configuration, database design, migration, security implementation, troubleshooting, backup management, and data recovery. Database administration plays a vital and central role within any organization relying on one or multiple databases. DBAs serve as the overall leaders and supervisors of the database system.

Types of Database Administrator (DBA) :
There are several types of Database Administrators (DBAs), each specializing in different aspects of managing and maintaining databases. Here are some common types:

1. **Administrative DBA:** Responsible for overall management of the database server, including tasks such as backup and recovery, security management, performance monitoring, and system maintenance.

2. **Development DBA:** Focuses on designing, developing, and implementing the database structure based on the requirements of applications. They create database objects, optimize queries, and ensure efficient data access.

3. **Data Warehouse DBA:** Specializes in managing data warehouses and data marts. They design and implement large-scale data repositories used for business intelligence and reporting.

4. **Cloud DBA:** Manages databases hosted on cloud platforms. They ensure data availability, performance, and security in cloud environments.

5. **Application DBA:** Works closely with application developers to optimize database performance for specific applications. They handle tasks like query tuning, indexing, and database design for application requirements.

6. **Backup and Recovery DBA:** Focuses on creating and implementing strategies for data backup, recovery, and disaster planning to ensure data availability and minimize downtime.

7. **Performance Tuning DBA:** Specializes in optimizing database performance. They monitor and analyze performance metrics, identifying and resolving performance bottlenecks.

8. **Security DBA:** Concentrates on database security, implementing access controls, encryption, and auditing mechanisms to protect sensitive data from unauthorized access and breaches.

9. **Replication DBA:** Manages database replication processes to ensure data consistency and availability across multiple database instances.

10. **Database Architect:** Designs and plans the overall structure of the database system, including schema design, table relationships, and data integrity. They provide a blueprint for the database implementation.

11. **Disaster Recovery DBA:** Focuses on creating and testing disaster recovery plans to ensure business continuity in case of data loss or system failures.

12. **Migration DBA:** Specializes in migrating data from one database platform to another. They ensure data accuracy, consistency, and minimal downtime during migrations.

13. **Database Compliance DBA:** Ensures that the database system adheres to industry regulations and compliance standards, such as GDPR or HIPAA.

14. **Big Data DBA:** Specializes in managing and optimizing large-scale databases used for big data analytics. They handle distributed databases, NoSQL databases, and data processing frameworks.

15. **NoSQL DBA:** Manages NoSQL databases, which are used for handling unstructured or semi-structured data. They specialize in platforms like MongoDB, Cassandra, and Redis.

Different types of Database Users

Database Administrator (DBA):

- Defines database schema and controls the three levels of the database.
- Creates new user accounts and manages access.
- Ensures database security and authorizes user access.
- Monitors performance, recovery, backup, and provides technical support.
- Responsible for resolving security breaches and performance issues.

- Performs Data Control Language (DCL) operations.
- Has a system or superuser account in the DBMS.
- Handles hardware and software failures and repairs damage.
- Manages privileges and access permissions.

Naive / Parametric End Users:
- Unsophisticated users who frequently use database applications.
- Lack in-depth DBMS knowledge.
- Commonly interact with databases to perform specific tasks.
- Examples: Railway ticket booking users, bank clerks.

System Analyst:
- Analyzes requirements of parametric end users.
- Ensures end users' needs are met.
- Acts as an intermediary between end users and the DBMS.

Sophisticated Users:
- Familiar with databases.
- Can develop own database applications.
- Write SQL queries directly through the query processor.
- Often engineers, scientists, business analysts.

Database Designers:
- Design database structures including tables, indexes, views, triggers, etc.
- Enforce constraints and relationships in the design.
- Understand requirements of different user groups.
- Create designs that satisfy diverse user needs.

Application Programmers:
- Write code for application programs.
- Back-end programmers who develop software.
- Use programming languages like Visual Basic, C, etc.
- Design, debug, test, and maintain programs for users' interaction with databases.

Casual Users / Temporary Users:
- Occasional database users seeking new information each time.

- Examples include middle or higher-level managers.

Specialized Users:

- Sophisticated users who create specialized database applications.
- Applications might not fit traditional data-processing frameworks.
- Examples include computer-aided design applications.

ER Model & Diagram, Database Schema

The term "ER model" refers to the Entity-Relationship model, which serves as a data model at an elevated level. Its purpose is to establish the definition of data components and their interconnections within a designated system.

It generates a conceptual blueprint for the database, creating a straightforward and easily manageable representation of data.

In the practice of ER modeling, the arrangement of the database structure is visually represented through a diagram termed as an entity-relationship diagram.

Why Use ER Diagrams In DBMS?

ER diagrams play a pivotal role in portraying the E-R model within a database, simplifying the process of transforming them into relational structures (tables).

The significance of ER diagrams lies in their ability to model real-world objects, adding a layer of practicality to their utility.

With no demand for technical prowess or hardware assistance, ER diagrams offer a user-friendly approach.

Even for individuals unfamiliar with complex concepts, these diagrams are highly accessible and uncomplicated to generate.

ER diagrams offer a uniform approach to logically visualizing data, providing a standardized solution.

Symbols Used in ER Model

The ER Model is employed to conceptualize the system's logical aspect from a data stance, encompassing these symbols

- **Rectangles: T**hese rectangles are utilized to symbolize Entities within the ER Model.

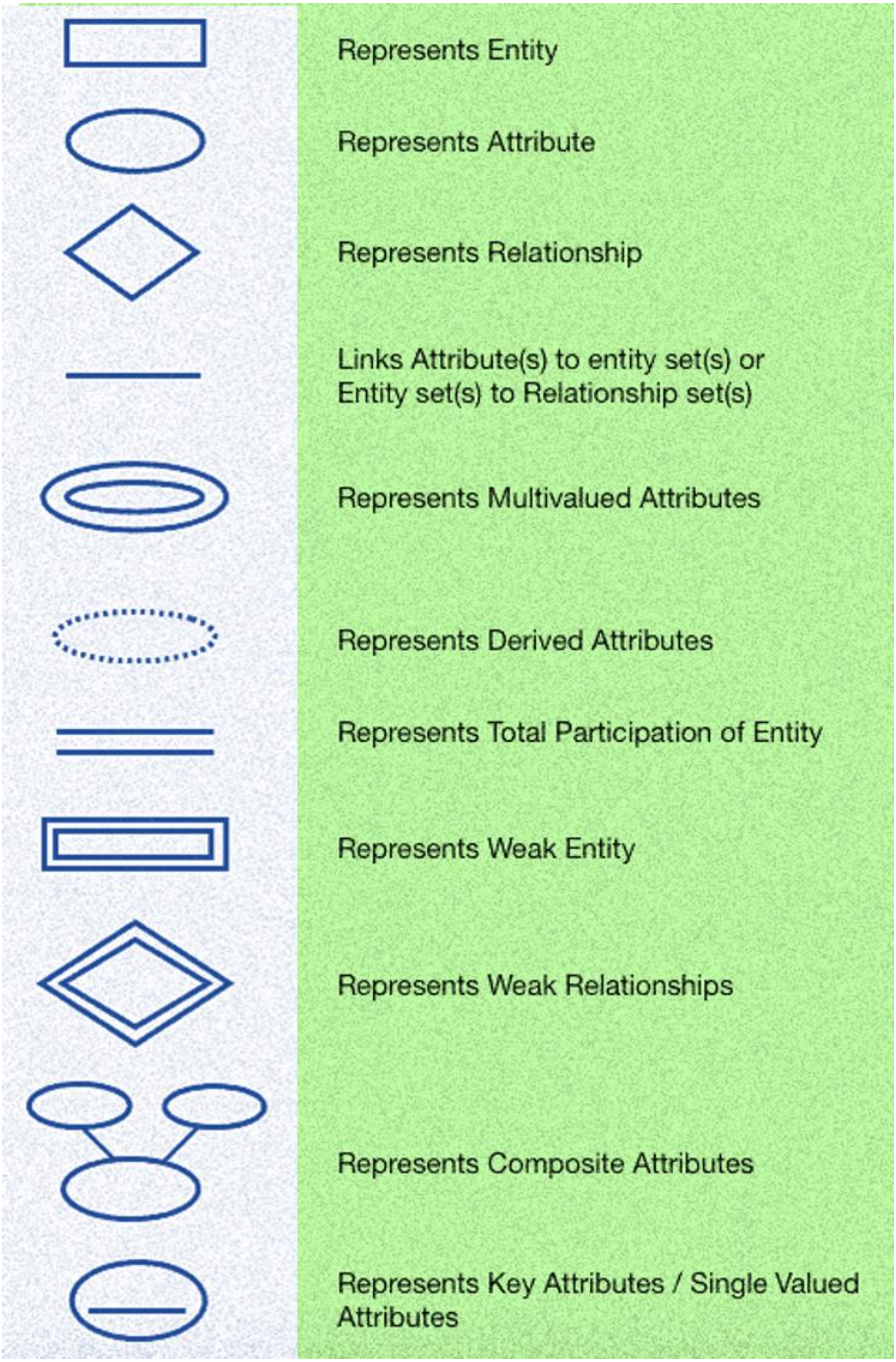
- **Ellipses:** Ellipses are employed to represent Attributes in the ER Model.

- **Diamonds:** The diamond shape is used to portray Relationships among different Entities.

- **Lines:** Lines are utilized to depict associations between attributes and entities, as well as entity sets with different types of relationships.

- **Double Ellipses**: Double ellipses are used to signify Multi-Valued Attributes.

- **Double Rectangles:** Double rectangles are employed to indicate a Weak Entity.



Represents Entity

Represents Attribute

Represents Relationship

Links Attribute(s) to entity set(s) or
Entity set(s) to Relationship set(s)

Represents Multivalued Attributes

Represents Derived Attributes

Represents Total Participation of Entity

Represents Weak Entity

Represents Weak Relationships

Represents Composite Attributes

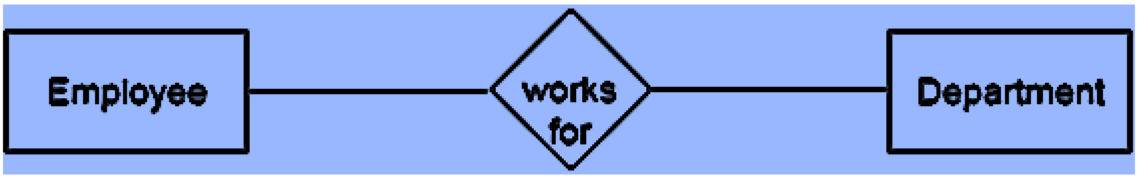
Represents Key Attributes / Single Valued
Attributes

Components of ER Diagram

Entity

An entity can encompass various elements such as objects, classes, individuals, or locations. Within an ER diagram, entities are symbolized using rectangular shapes.

Take into account an instance of an organization - a manager, product, staff member, department, and so forth could be regarded as entities in this context.



1. Weak Entity

A weak entity is an entity that relies on another entity. Unlike a strong entity, a weak entity lacks its own key attribute. It is depicted using a double rectangle in diagrams.

Example:-A corporation has the ability to retain data regarding the family members (parents, children, spouse) of an employee. However, these family members only hold significance due to their connection with the employee. Consequently, family members are categorized as a vulnerable entity type, while employees are recognized as the pivotal entity type for the family members, indicating that they are..

2. Strong Entity

A Strong Entity refers to an entity possessing a primary Attribute. It maintains independence from other entities within the Schema, boasting a primary key that facilitates its distinct identification. This is symbolized by a rectangle and is termed as a Strong Entity Type.

Attributes

Attributes are the characteristics that establish the nature of an entity type. For instance, attributes like Roll Number, Name, Date of Birth, Age, Address, and Mobile Number serve to characterize the entity type of Student. Within an Entity-Relationship (ER) diagram, attributes are depicted using oval shapes.

1. Key Attribute-

The key attribute is the distinctive characteristic that identifies each individual entity within the set of entities. For instance, the Roll_No serves as a unique identifier for every student. In an Entity-Relationship (ER) diagram, this key attribute is depicted as an oval shape containing underlying lines.

2. Composite Attribute

A composite attribute is formed by combining multiple individual attributes. An illustration of this is the student Entity type's Address attribute, which encompasses Street, City, State, and Country. Within an ER diagram, a composite attribute is symbolized by an oval containing smaller ovals

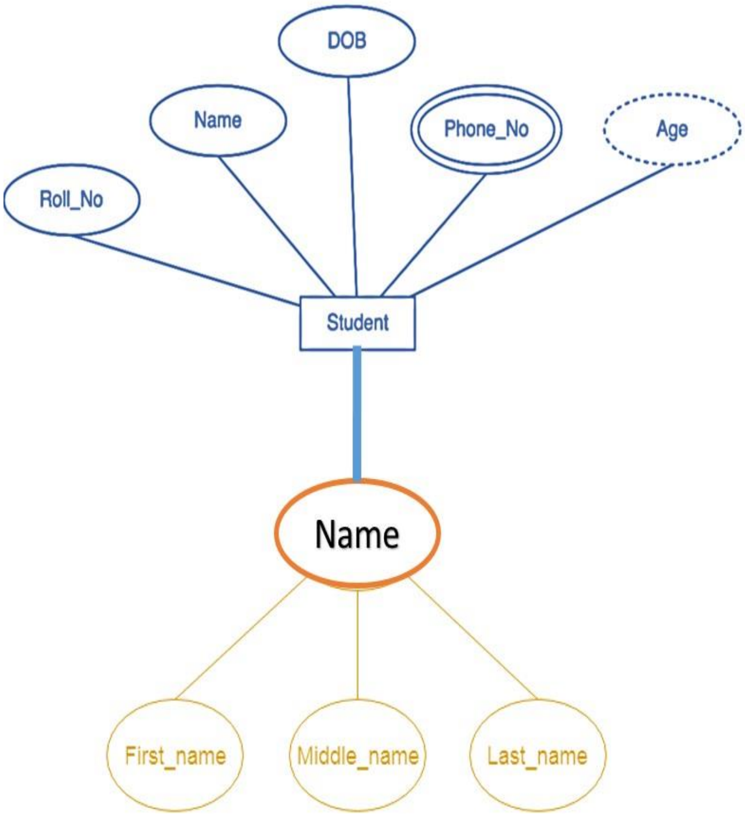
3. Multivalued Attribute

A characteristic that holds multiple values for a specific entity. For instance, Phone_Number (which can have multiple entries for a single student). In an Entity-Relationship (ER) diagram, a multivalued attribute is depicted using a pair of connected ovals.

4. Derived Attribute

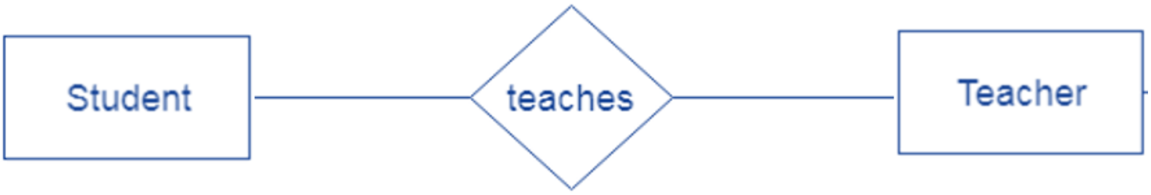
An attribute that is obtained based on other attributes within the entity type is referred to as a derived attribute. For example, Age (calculated from Date of Birth). In an ER diagram, a derived attribute is depicted using a dashed oval.

The Student entity type along with its attributes can be illustrated as follows:



Relationship

A relationship is utilized to define the connection between entities. In graphical representations, such as in an ER diagram, this relationship is symbolized using a diamond or rhombus shape.

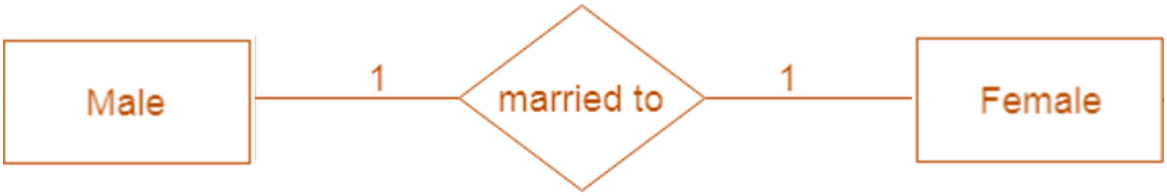


Types of relationship are as follows:

1. One-to-One Relationship

If a relationship is such that it links a singular instance of an entity, it is referred to as a one-to-one relationship.

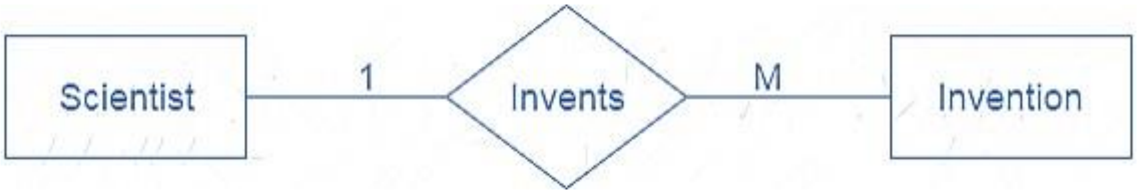
Example-A woman can be married to a single man, and conversely, a man can be married to a single woman.



2.One-to-many relationship

When only one instance of the entity on the left is connected to multiple instances of the entity on the right within a relationship, it is referred to as a one-to-many relationship.

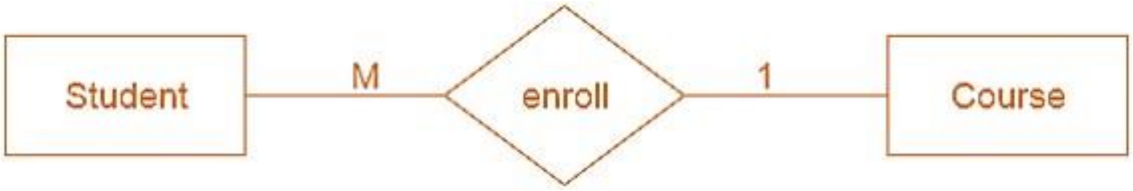
Example--Scientists have the capability to create numerous inventions, while the act of inventing a particular invention is attributed to a specific scientist.



3. Many-to-one relationship

When more than one instances of the entity on the left are linked to a single instance of the entity on the right through a relationship, it is referred to as a many-to-one relationship.

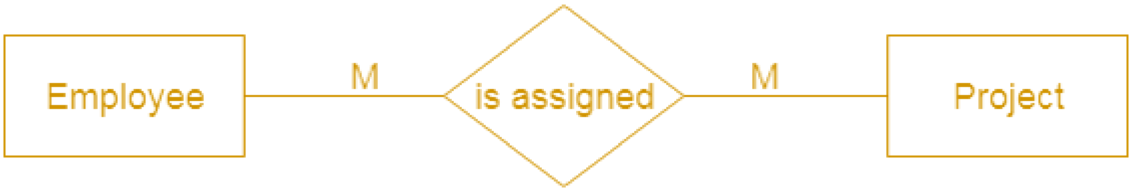
Example-A student is enrolled in a single course, whereas a course can accommodate multiple students.



4.Many-to-many relationship

A many-to-many relationship arises when multiple instances of the entity on the left are connected with multiple instances of the entity on the right through the relationship.

Example -An employee can be assigned to multiple projects, and likewise, a project can involve multiple employees.



Advantages of ER Model

Simplicity: The conceptual ER Model construction is straightforward. If we grasp the connections between attributes and entities, developing the ER Diagram for the model becomes effortless.

Efficient Communication Tool: Database designers extensively rely on this model to effectively convey their concepts.

Smooth Transition to Various Models: This model harmonizes effectively with the relational model, seamlessly transforming the ER model into tables. Furthermore, it can be adapted into other models like the network model, hierarchical model, and more.

Disadvantages of ER Model

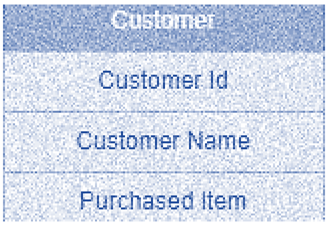
Lack of Notation Consistency: An established industry standard for creating an ER model is absent. Consequently, different developers might employ notations that are unclear to their peers.

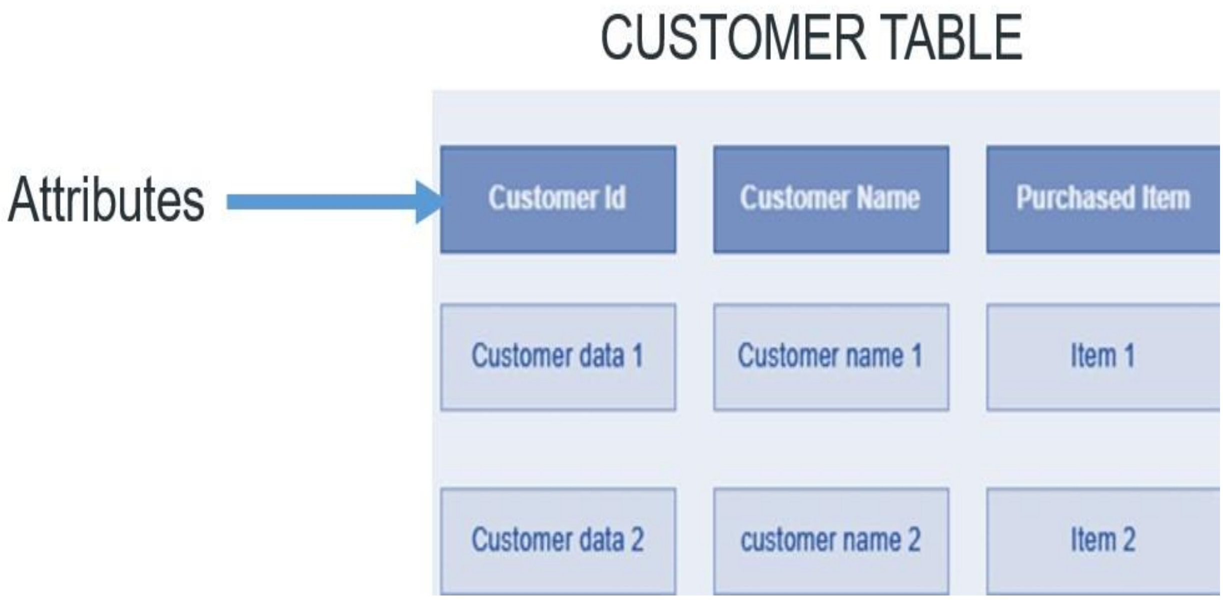
Concealed Information: The ER model's high-level perspective can lead to the omission or concealment of certain details, potentially resulting in the loss of some information.

Schema

What is Schema?

- The framework of the database takes form through the attributes, and this framework is identified as the schema.
- A schema outlines the logical limitations, such as tables and primary keys, that define the structure of a database.
- The schema doesn't encompass the data attribute types.





Customer Id	Customer Name	Purchased Item
Customer data 1	Customer name 1	Item 1
Customer data 2	customer name 2	Item 2

Database Schema

A ***database schema*** is a logical and structured representation of the organization, arrangement, and relationships among the data stored in a database. It defines the design, format, and constraints of the data stored in the database tables, along with the interconnections between these tables. In essence, a database schema outlines the blueprint for how data is organized, stored, and accessed within a database management system. It includes information about tables, fields, data types, relationships, constraints, and other elements that define the structure and integrity of the database.

Types of Database Schema

The database schema is categorized into three types, namely:

1. *Logical Schema*
2. *Physical Schema*
3. *View Schema*

1 .Physical Database Schema

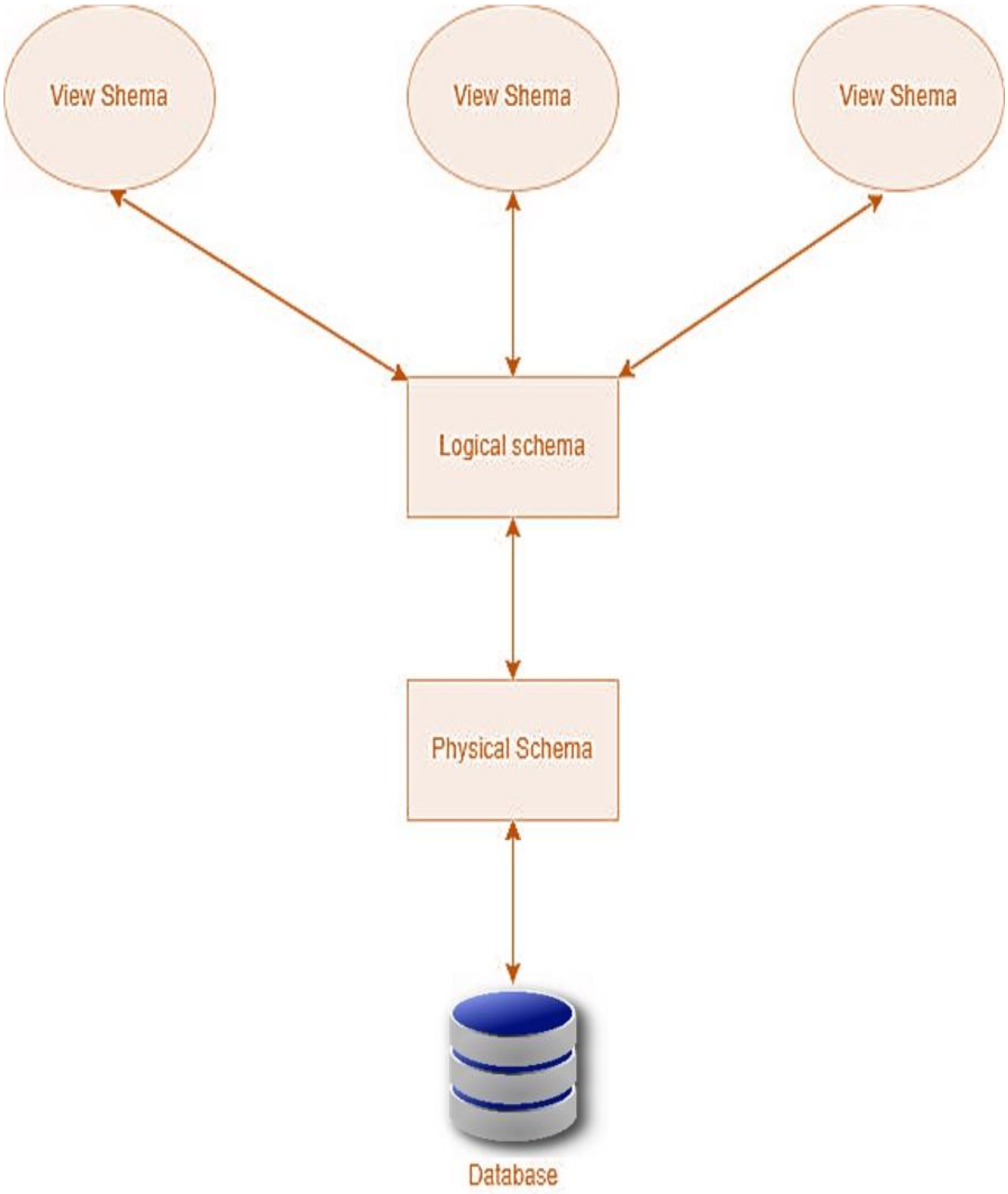
- The physical schema outlines how data is physically stored within storage systems as files and indices. It involves the concrete code or syntax required to establish the database's structure. When crafting a database structure on the physical level, it is referred to as the physical schema.
- The choice of data storage locations and methods within various storage blocks is made by the Database Administrator.

2. Logical Database Schema

- The logical database schema encompasses all the rational restrictions to be enforced on the stored data, as well as outlines the tables, perspectives, entity connections, and integrity constraints.
- The logical schema elucidates the manner in which data is stored, comprising tables and their interconnected attributes.
- Through the utilization of ER modeling, the connections among data elements are upheld.
- Within the logical schema, diverse integrity constraints are outlined to ensure the accuracy of data insertion and updates.

3. View Schema

- This refers to a view-level design that outlines how interactions between end-users and the database are defined.
- Users can interact with the database through an interface without requiring extensive knowledge about the underlying data storage methods employed within the database.



Three Layer Schema Design

Creating Database Schema

To create a schema, the "CREATE SCHEMA" statement is employed in various databases. However, the interpretation of this statement can differ across different database systems. Let's explore some examples of statements used for creating a database schema in various database systems:

1. MySQL: In MySQL, the "**CREATE SCHEMA**" statement is utilized to create a database. This is because, in MySQL, both "CREATE SCHEMA" and "CREATE DATABASE" statements serve the same purpose.

2. SQL Server:-Within SQL Server, the "CREATE SCHEMA" statement is employed to generate a new schema.

3. Oracle Database:-In Oracle Database, the "CREATE USER" statement is used to create a new schema. In Oracle, a schema is automatically generated with every database user. The "CREATE SCHEMA" statement, however, doesn't create a new schema. Instead, it populates the existing schema with tables and views, facilitating access to these objects without necessitating multiple SQL statements for separate transactions.

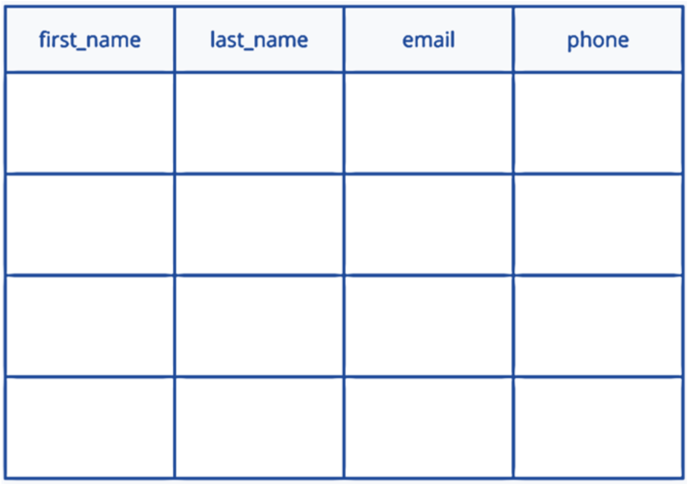
Database Schema Designs

Creating a schema design constitutes the initial phase in establishing a solid groundwork for data administration. Inefficient schema designs prove challenging to oversee and result in heightened memory consumption and resource utilization. The logical course of action is contingent upon the demands of the business. Opting for the appropriate database schema design is essential to simplify the project lifecycle. Presented herewith is an assortment of well-known database schema designs.

1. Flat Model
2. Hierarchical Model
3. Network Model
4. Relational Model
5. Star Schema
6. Snowflake Schema

1.Flat Model

A flat model schema embodies a 2-dimensional array where each column holds identical data types, and items within a row possess interrelatedness. It can be likened to a solitary spreadsheet or a database table devoid of interconnections. This schema structure proves optimal for modest applications devoid of intricate data structures.



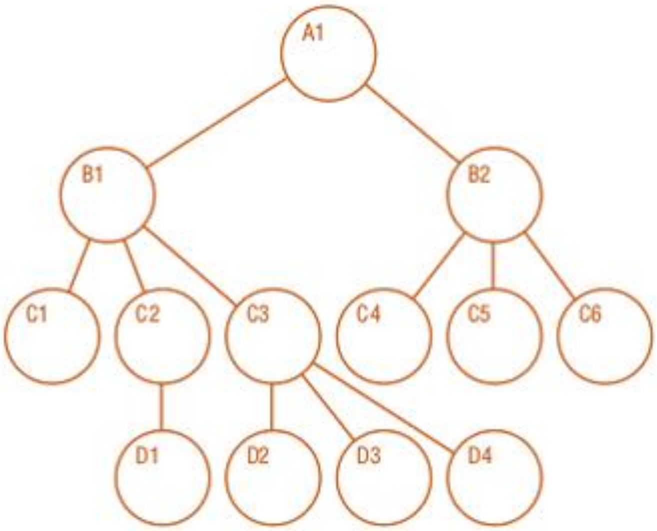
first_name	last_name	email	phone

Designing Flat model

2.Hierarchical Model

The Hierarchical model design features a configuration resembling a tree. Within this tree structure, there exists a root data node and subsequent child nodes. A one-to-many relationship is established between each parent node and its child node. These types of database schemas find representation in formats like XML or JSON files, which are capable of encompassing entities along with their respective sub-entities.

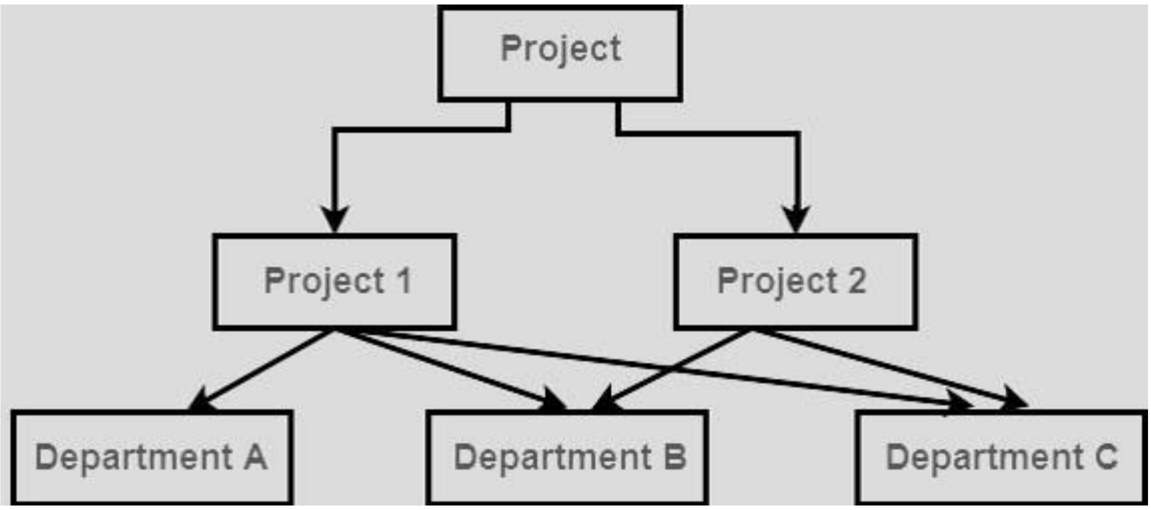
Hierarchical schema models excel in housing nested data, effectively representing instances like Hominoid classification.



Designing Hierarchical Model

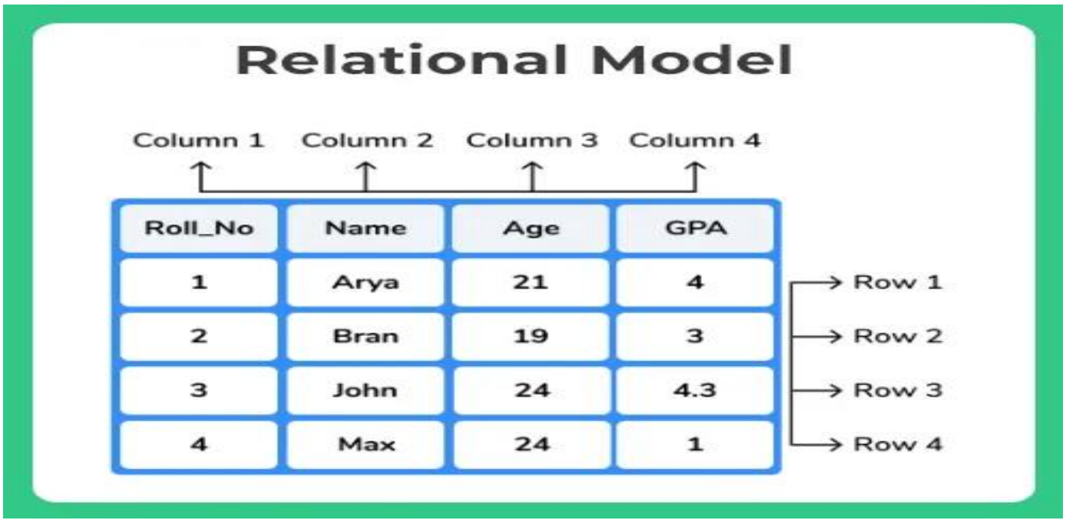
3.Network Model

The network model and the hierarchical model bear striking resemblances, but a significant disparity lies in their handling of data relationships. The network model permits the existence of many-to-many relationships, in contrast to the hierarchical models, which solely accommodate one-to-many relationships.



4.Relational Model

Relational models serve as the foundation for relational databases, where data is stored in the form of tables or relations. Within this context, relational operators are employed to manipulate and compute various values from the data.


Relational Model

Roll_No	Name	Age	GPA
1	Arya	21	4
2	Bran	19	3
3	John	24	4.3
4	Max	24	1

Column 1
Column 2
Column 3
Column 4
Row 1
Row 2
Row 3
Row 4

5.Star Schema

The star schema represents an alternative schema design approach for data organization. It excels in managing and analyzing vast volumes of data and operates based on the concepts of "Facts" and "Dimensions". In this context, a fact corresponds to a numerical data point that drives business processes, while a dimension provides context and description to the facts. The Star Schema is particularly effective for structuring data within Relational Database Management Systems (RDBMS).

6.Snowflake Schema

The snowflake schema is a modification of the star schema. In the star schema, there exists a central "Fact" table housing primary data points with references to its associated dimension tables. However, in the snowflake schema, dimension tables have the potential to branch out into their own additional dimension tables, creating a more intricate hierarchical structure.

Designing Database using Normalization Rules

Normalization

Normalization is a database design process that involves organizing and structuring a relational database to reduce data redundancy and dependency, thereby improving data integrity and overall efficiency. The primary goal of normalization is to eliminate anomalies that can occur when data is stored in a non-optimal structure. It helps ensure that the data is stored in a way that minimizes redundancy while preserving the relationships between different pieces of information.

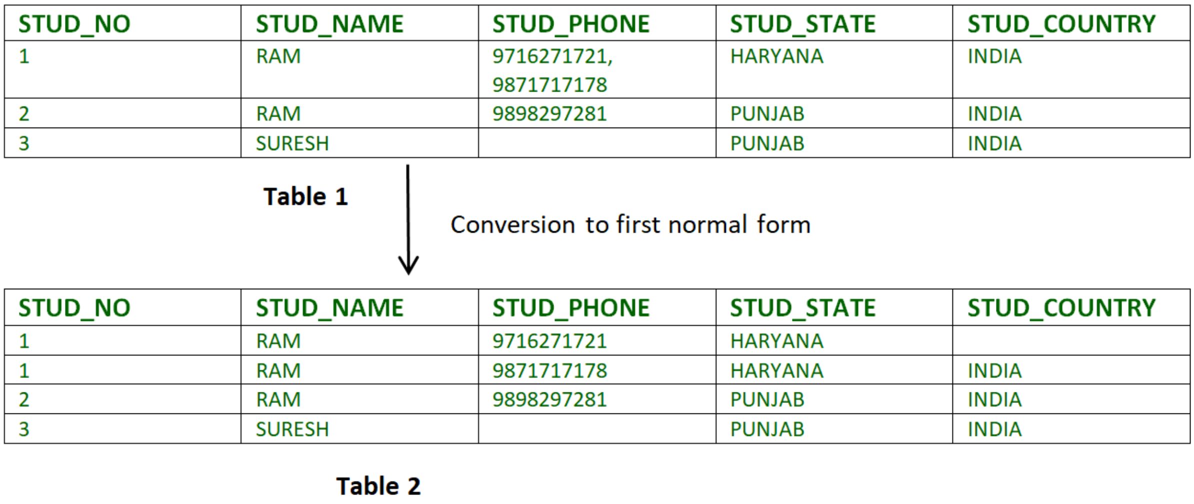
The process of normalization is typically carried out through a series of steps or "normal forms," each addressing specific types of data anomalies. The most commonly discussed normal forms are the first normal form (1NF), second normal form (2NF), third normal form (3NF), and beyond.

Here's a brief overview of these normal forms:

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

In 1NF, each column in a table must hold only atomic (indivisible) values, and each row should be uniquely identifiable. This eliminates the possibility of storing multiple values within a single cell and ensures that the data is organized in a tabular format.

EXAMPLE- Table 1's "STUDENT" relation violates the first normal form (1NF) due to the presence of the multi-valued attribute "STUD_PHONE". The decomposition of this relation into 1NF is demonstrated in table 2



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

Table 1

Conversion to first normal form

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

Table 2

Example 2 -

ID Name Courses

----

1 A m1, m2

2 B m3

3 c m2, m3

- The previous table exhibits a multi-valued attribute "Course," which renders it non-compliant with the first normal form (1NF). The subsequent table, on the other hand, adheres to 1NF principles, as it lacks any multi-valued attributes.

ID	**Name**	**Course**
1	**A**	**m1**
1	**A**	**m2**
2	**B**	**m3**
3	**C**	**m2**
3	**C**	**m3**

2. **Second Normal Form (2NF):**

In 2NF, the table must be in 1NF, and all non-key attributes (attributes that are not part of the primary key) must be fully functionally dependent on the entire primary key. This eliminates partial dependencies, where non-key attributes are dependent on only a portion of the primary key.

Example-

ProductID	**product**	**Brand**
1	Monitor	Dell
2	Monitor	Apple
3	Scanner	HP
4	Head phone	JBL

Product table following 2NF:

Products Category table:

ProductID	**product**
1	Monitor
2	Monitor
3	Scanner
4	Head phone

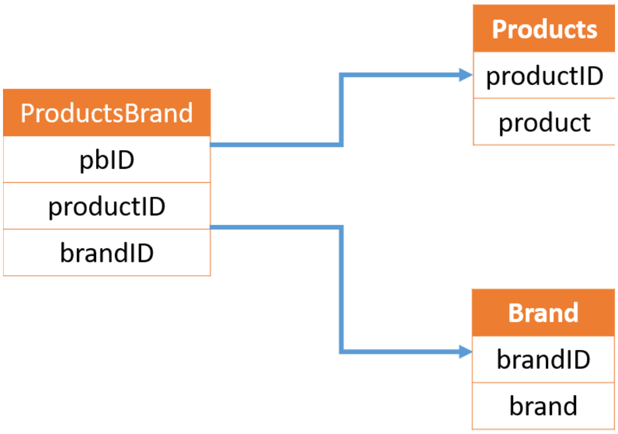
Brand table:

ProductID	**Brand**
1	Dell

2	Apple
3	HP
4	JBL

Products Brand table:

pbID	**productID**	**brandID**
1	1	1
2	1	2
3	2	3
4	3	4



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

In 3NF, the table must be in 2NF, and all non-key attributes must be directly dependent on the primary key. This eliminates transitive dependencies, where non-key attributes are dependent on other non-key attributes.

These three normal forms are the most commonly used and usually suffice for many database designs. However, there are additional normal forms like Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF) that address more complex scenarios of data dependencies and anomalies.

Normalization helps improve database design by:

- Reducing data redundancy, which in turn conserves storage space and ensures consistency.

- Preventing update anomalies (problems that arise when changes to data aren't properly reflected).

- Enhancing data integrity by maintaining consistent relationships between entities.

Benefits of Normalization

1. Reduces data duplication.
2. Enhances database organization.

3. Ensures uniform data across the database.

4. Enables increased flexibility in database design.

5. Upholds relational integrity principles.

Disadvantages of Normalization

1. Requires clear user requirements before database construction.

2. Performance deterioration occurs with higher-level normalization, like 4NF and 5NF.

3. Normalizing relations of higher degrees is time-consuming and complex.

4. Inadequate decomposition can lead to suboptimal database design and subsequent issues.

Various data types Data integrity, DDL DML and DCL statements

Data integrity refers to the accuracy, consistency, and reliability of data in any given context. It is a fundamental aspect of data management and is essential for ensuring that data remains intact and trustworthy throughout its lifecycle. Data integrity is particularly important in various domains, including databases, information systems, and data storage, as well as in compliance with regulations and data security.

1.Accuracy: Data should be free from errors and represent the true and correct values or information. Inaccurate data can lead to flawed decision-making and operational problems.

2. Consistency: Data should remain uniform and coherent across various data sources, databases, or data sets. Inconsistent data can lead to confusion and hinder data analysis.

3. Completeness: Data must be complete, meaning that all expected data elements or fields are present and appropriately filled in. Incomplete data can result in incomplete analysis or reports.

4. Reliability: Data should be reliable and available when needed. Unreliable data or data that is frequently unavailable can disrupt business processes.

5. Security: Data integrity also involves ensuring that data is protected from unauthorized access, tampering, or corruption. Security measures like encryption and access controls are essential to maintaining data integrity.

Methods and practices for ensuring data integrity include:

1.Validation Rule: Implement validation rules and constraints to ensure that data entered into a system adheres to predefined criteria, such as data types, ranges, and formats.

2. Data Validation: Employ data validation techniques to check data for accuracy and consistency. This may involve data cleansing, deduplication, and normalization.

3. Backup and Recovery: Regularly back up data and have robust disaster recovery plans in place to ensure data can be restored in case of corruption or loss.

4. Access Control: Implement access controls and authentication mechanisms to prevent unauthorized access and modification of data.

5. Audit Trails: Maintain audit trails to record and track changes made to data, including who made the changes and when they occurred. This helps identify and rectify unauthorized alterations.

6. Hash Functions: Use cryptographic hash functions to create checksums or hashes of data. Comparing these hashes can reveal any unauthorized changes to the data.

7. Data Verification: Periodically verify the accuracy and consistency of data through reconciliation and verification processes.

8. Data Governance: Establish data governance policies and procedures to enforce data integrity standards across the organization.

Types of data integrity

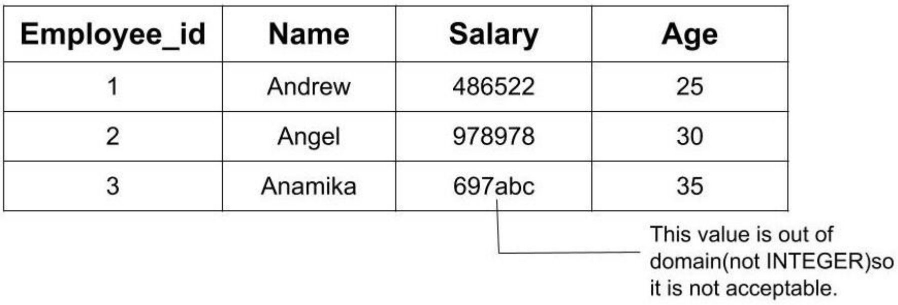
There are mainly four types of Data Integrity:

1. Domain Integrity
2. Entity Integrity
3. Referential Integrity
4. User-Defined Integrity

Domain Integrity

Domain, in this context, pertains to the acceptable values within a specified range. It denotes the scope of values that can be utilized and stored in a specific database column. The available data types primarily include integers, text, and dates, among others. It's important that any input entered into a column falls within the permissible range of the associated data type.

Example-To store employee salaries in the 'employee_table,' it's possible to implement constraints that permit only integer values. Any input that deviates from this requirement, like text or character-based data, would be declined, and the Database Management System (DBMS) would generate error messages to indicate the violation of the defined constraint. This helps ensure that only valid integer values are accepted for salary entries in the database, maintaining data consistency and accuracy.



Employee_id	Name	Salary	Age
1	Andrew	486522	25
2	Angel	978978	30
3	Anamika	697abc	35

This value is out of domain(not INTEGER)so it is not acceptable.

DOMAIN INTEGRITY

Entity Integrity

Every row representing an entity in a database table must have a distinct means of identification.

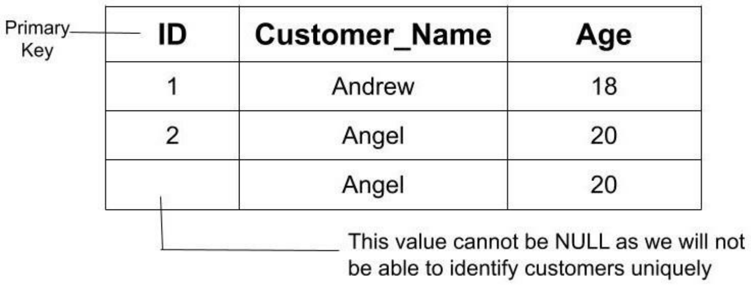
This is typically achieved using primary keys, which serve as unique identifiers for each record.

It's essential to enforce the entity constraint, which specifies that the primary key value must not be NULL.

This requirement ensures that every record in the database has a specific and non-null primary key value.

When the primary key value is not NULL, it becomes possible to distinguish records from one another, even if all other field values are identical. In essence, primary keys enable the unequivocal identification of each individual record in the database**.**

Example:-In a customer database with a 'customer_table' containing attributes like age and name, it's crucial to ensure that each customer can be uniquely identified. Sometimes, there might be two customers with identical names and ages, leading to confusion when retrieving data. To address this challenge, primary keys are assigned to each table entry. These primary keys serve the purpose of uniquely identifying each record in the table, even in cases where other attributes like name and age might not be sufficient for differentiation.



Primary Key

ID	Customer_Name	Age
1	Andrew	18
2	Angel	20
	Angel	20

This value cannot be NULL as we will not be able to identify customers uniquely

ENTITY INTEGRITY

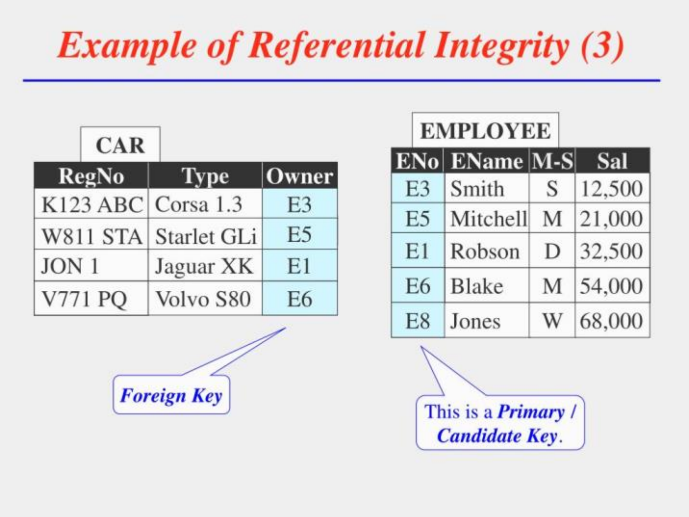
Referential Integrity

Referential Integrity is a crucial concept employed to uphold data consistency when managing two interconnected tables within a database. It involves establishing specific rules within the database structure to ensure that modifications, insertions, and deletions in the database do not compromise data integrity. These constraints for referential integrity dictate that when a foreign key in one table references the primary key of another table, every value of that foreign key in the first table must either be null or correspond to a valid entry in the second table.

Example:-Imagine we have two tables: "*table 1*" (with columns student_id, name, age, and course_id) and "*table 2*" (with columns course_id, course_name, and duration).

In the context of referential integrity, it means that if any "course_id" exists in the "*table 1*" table, it must also exist in the "*table 2*" table; otherwise, this scenario is not permitted.

In other words, the "course_id" in the "*table 1*" table should either be null or, if a "course_id" is present, it must be a valid entry in the *"table 2*" table. This way, referential integrity is maintained to ensure the consistency and accuracy of data between these two tables.



Example of Referential Integrity (3)

CAR

RegNo	Type	Owner
K123 ABC	Corsa 1.3	E3
W811 STA	Starlet GLi	E5
JON 1	Jaguar XK	E1
V771 PQ	Volvo S80	E6

EMPLOYEE

ENo	EName	M-S	Sal
E3	Smith	S	12,500
E5	Mitchell	M	21,000
E1	Robson	D	32,500
E6	Blake	M	54,000
E8	Jones	W	68,000

Foreign Key

This is a *Primary / Candidate Key*.

User-Defined Integrity

On occasion, domain, referential, and entity integrity alone may fall short in preserving data integrity. In such cases, additional measures are often employed, typically involving the use of triggers and stored procedures. Triggers are essentially sets of statements that automatically execute in response to predefined events, providing a means to enforce more intricate data integrity rules when necessary.

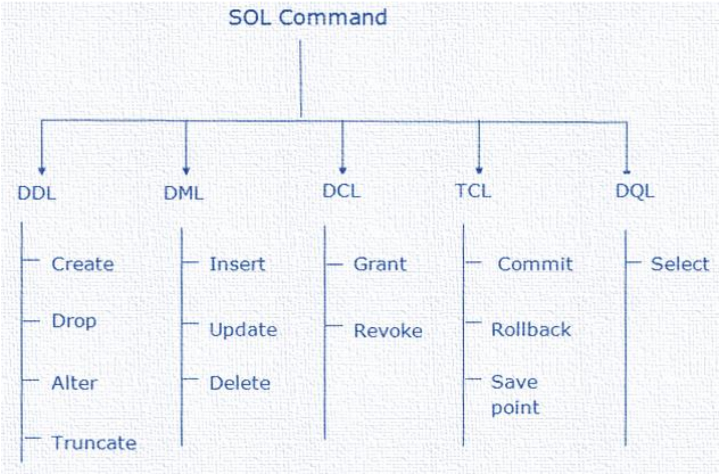
Example:-When a new row containing marks for various subjects of students is added to the student_table, an automatic calculation of the new average is performed and stored.

SQL commands

- SQL commands are directives employed to interact with a database, facilitating the execution of particular actions, operations, and inquiries on data within the database.
- SQL has the capability to execute a range of functions, such as generating tables, inserting data into tables, deleting tables, altering table structures, and defining user permissions.

Types of SQL Commands

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



DDL (Data Definition Language)

DDL, which stands for Data Definition Language, is a subset of SQL (Structured Query Language) used for defining and managing the structure of a database. DDL commands allow you to create, modify, and delete database objects like tables, indexes, and constraints. DDL is primarily concerned with defining the schema or structure of the database. Common DDL commands include:

1.CREATE: Used to create new database objects such as tables, indexes, and views.

```
CREATE TABLE table_name (
            column1 datatype,
```

2.ALTER: Used to modify the structure of an existing database object, like adding or dropping columns from a table.

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

3.DROP: Used to delete a database object like a table, index, or view.

```
DROP TABLE table_name;
```

4.TRUNCATE: Used to remove all rows from a table but keep the table structure intact.

```
TRUNCATE TABLE table_name;
```

5.COMMENT: Used to add comments or descriptions to database objects for documentation purposes.

```
ON TABLE table_name IS 'This is a table comment.';
```

6.CREATE INDEX: Used to create an index on one or more columns of a table for faster data retrieval.

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

7.CREATE VIEW: Used to create a virtual table based on the result of a query.

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

DDL commands are typically used by database administrators and developers to design and manage the database's structure. They are essential for defining how data should be organized and ensuring data integrity within a database system.

DML(Data Manipulation Language)

DML, which stands for Data Manipulation Language, is a subset of SQL (Structured Query Language) that is used for managing and manipulating data stored in a database. DML commands allow you to perform operations on the data itself, such as inserting, updating, and deleting records in database tables. The primary DML commands are:

1.INSERT: Used to add new rows (records) of data into a database table.

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

2.UPDATE: Used to modify existing data in a database table.

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

3.DELETE: Used to remove rows (records) from a database table based on specified criteria.

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

DML commands are essential for maintaining and changing the data within a database. These commands enable users and applications to insert new data, update existing data, and remove unwanted data, ensuring that the database remains accurate and up-to-date.

DCL (Data Control Language)

DCL, which stands for Data Control Language, is a subset of SQL (Structured Query Language) used for controlling and managing permissions and access rights within a database management system (DBMS). DCL commands are essential for ensuring data security and access control by specifying which users or roles have the authority to perform certain actions on database objects. The two primary DCL commands are:

1.GRANT: The GRANT command is used to give specific privileges or permissions to users or roles. These privileges can include the ability to perform actions like SELECT, INSERT, UPDATE, DELETE, or even the ability to create or modify database objects.

```
GRANT privilege_type
ON object_name
TO user_or_role;
```

For example, to grant SELECT permission on a table to a user:

```
GRANT SELECT ON table_name TO user_name;
```

2.REVOKE: The REVOKE command is used to remove previously granted privileges from users or roles.

```
REVOKE privilege_type ON object_name FROM user_or_role;
```

For example, to revoke SELECT permission on a table from a user:

```
REVOKE SELECT ON table_name FROM user_name;
```

DCL commands play a crucial role in controlling who can access and manipulate data within a database, ensuring data integrity and security. Database administrators use these commands to define and enforce access policies, restrict unauthorized access, and manage the permissions of users and roles in the database system. Properly configured DCL commands help protect sensitive data and maintain the integrity of the database.

TCL (Transaction Control Language)

TCL, which stands for Transaction Control Language, is a subset of SQL (Structured Query Language) used for managing database transactions. Transactions in a database are sequences of one or more SQL statements that are treated as a single, indivisible unit of work. TCL commands are used to control the beginning and ending of transactions, ensuring data consistency and integrity. The primary TCL commands are:

1.COMMIT: The COMMIT command is used to permanently save the changes made during the current transaction. Once a COMMIT is issued, all changes are made permanent and cannot be rolled back.

```
COMMIT;
```

2.ROLLBACK: The ROLLBACK command is used to undo changes made during the current transaction and restore the database to its previous state. It cancels all the changes made since the last COMMIT or SAVEPOINT.

```
ROLLBACK;
```

3.SAVEPOINT: The SAVEPOINT command is used to set a point within a transaction to which you can later roll back if needed. It allows you to create intermediate savepoints within a transaction.

```
SAVEPOINT savepoint_name;
```

TCL commands are critical for maintaining data consistency and ensuring that a series of related SQL statements are executed as a single, atomic operation. Transactions help protect data integrity and ensure that the database remains in a consistent state even in the presence of errors or interruptions. The use of COMMIT and ROLLBACK commands is essential for managing the success or failure of database operations within a transaction.

DQL(Data Query Language)

Data Query Language (DQL) is a subset of SQL (Structured Query Language) specifically designed for retrieving and querying data from a relational database. DQL commands are used to interact with the data stored within database tables, allowing users to retrieve, filter, and manipulate data to extract meaningful information. The primary DQL command is:

SELECT: The SELECT command is the core of DQL and is used to retrieve data from one or more tables in a database. It enables you to specify the columns you want to retrieve, apply filtering conditions, and sort the result set.

```
SELECT column1, column2

FROM table_name

WHERE condition ORDER BY column1;
```

In addition to SELECT, DQL may involve using clauses like WHERE to filter data, JOIN to combine data from multiple tables, GROUP BY for grouping data, HAVING for filtering grouped data, and more.

DQL is fundamental for extracting and presenting data in a structured and meaningful way, making it one of the most commonly used components of SQL, especially for reporting and analysis purposes. It allows users to interact with and retrieve data from a database, helping them make informed decisions based on the information stored in the database.

```
);
```

Enforcing Primary key and foreign key.

What is the Key?

In the realm of database management systems, a key refers to a particular attribute or a group of attributes employed to distinguish each record, often called a tuple, within a table uniquely. Keys hold significant importance in the structure of a relational database, as they serve to organize data and create connections between tables. Within a database, various types of keys exist, each serving a distinct function. Keys are pivotal components in relational databases, as they ensure record uniqueness, facilitate table relationships, and enhance overall database performance.

1. Enforcing a Primary Key

A primary key is a unique identifier for each record in a table. To enforce a primary key constraint, follow these steps:

- Define a primary key when creating a table using the '**PRIMARY KEY'** constraint.

```
CREATE TABLE employees ( employee_id INT PRIMARY KEY,
first_name VARCHAR(50), last_name VARCHAR(50) );
```

- If you're altering an existing table to add a primary key, use the ALTER TABLE statement.

```
ALTER TABLE employees ADD PRIMARY KEY (employee_id);
```

The primary key constraint enforces uniqueness, meaning that no two rows in the table can have the same value for the primary key column(s).

2. Enforcing a Foreign Key

A foreign key is a field in one table that refers to the primary key in another table, establishing a relationship between the tables. To enforce a foreign key constraint, follow these steps:

- Define a foreign key when creating a table using the '**FOREIGN KEY'** constraint.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
        customer_id INT, order_date DATE,
        FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
 );
```

- The '**REFERENCES'** clause specifies the table and column that the foreign key references. In this example, it references the '**customer_id'** column in the **'customers'** table.

- Ensure that the referenced column in the referenced table (in this case, **'customers'** in the **'customers'** table) has a primary key constraint.

Foreign key constraints enforce referential integrity, ensuring that data in the referencing table (in this case, the **'orders'** table) is consistent with data in the referenced table (in this case, the '**customers'** table).

Enforcing these constraints helps maintain data quality and consistency in your database, preventing the insertion of invalid or inconsistent data and facilitating data relationships between tables.

Difference between Primary Key and Foreign Key

Primary keys and foreign keys are both fundamental components of relational database design, but they serve different purposes and have distinct characteristics. Here are the key differences between primary keys and foreign keys:

Primary Key

1. **Uniqueness:** A primary key is a column or a set of columns in a table that uniquely identifies each row. Every value in the primary key column(s) must be unique within the table.

2. **Required:** A primary key is required for every table. It ensures that each row in the table can be uniquely identified.

3. **Constraints:** Primary keys enforce data integrity by preventing duplicate and null values in the primary key column(s).

4. **Indexed:** By default, primary key columns are automatically indexed, which can improve query performance.

5. **One per Table:** Each table can have only one primary key.

6. **Used for Joins:** Primary keys are often used as reference points for creating relationships with other tables through foreign keys.

Foreign Key:

1. **Relationships:** A foreign key is a column or a set of columns in one table that establishes a link or relationship between data in two tables. It references the primary key of another table.

2. **Referential Integrity:** Foreign keys enforce referential integrity by ensuring that data in the referencing table (child table) corresponds to data in the referenced table (parent table).

3. **Optional:** While foreign keys are commonly used to create relationships, they are not required in every table. Tables can exist without foreign keys.

4. **Values Must Match:** Values in the foreign key column(s) must match values in the primary key column(s) of the referenced table.

5. **Can Have Multiple:** A table can have multiple foreign keys, each referencing a different table and primary key.

6. **Used for Joins:** Foreign keys are used to join related tables, allowing you to retrieve data from multiple tables based on their relationships.

Adding Indices

Adding indices to databases is a fundamental part of database optimization. Indices improve query performance by allowing the database management system (DBMS) to quickly locate rows that meet specific criteria. Here's how you can add indices to a database, typically using SQL:

1. Determine Which Columns to Index:
Start by identifying the columns that are frequently used in WHERE clauses of your SELECT queries. These are good candidates for indexing, as they can significantly speed up data retrieval. Consider indexing columns used in JOIN conditions as well.

2. Choose the Appropriate Index Type:
There are different types of indexes, and the choice depends on your specific use case and DBMS:

- **Single-Column Index**: This indexes a single column.
- **Composite Index**: Indexes multiple columns together, useful for queries involving multiple criteria.
- **Unique Index**: Ensures uniqueness on indexed columns.
- **Clustered Index (in some DBMS):** Determines the physical order of data rows in the table.
- **Non-Clustered Index:** Contains a copy of the indexed columns and a pointer to the actual data row.

3. Create an Index:
To create an index, you use the '**CREATE INDEX**' statement in SQL. Here's a basic syntax example for creating a single-column index:

```
CREATE INDEX index_name

ON table_name (column_name);
```

*For example, to create an index called '**idx_last_name**' on the '**last_name**'column of a table named '**employees**' :-*

```
CREATE INDEX idx_last_name

ON employees (last_name);
```

If you want to create a composite index on multiple columns, you can specify them within the parentheses:

```
CREATE INDEX idx_name_age

ON employees (last_name, age);
```

4. Monitor and Maintain Indexes:
After adding indexes, it's essential to monitor their performance over time. Regularly analyze query execution plans to ensure that the indexes are being used as expected. Additionally, you may need to perform index maintenance tasks like rebuilding or reorganizing indexes to optimize performance as data changes.

5. Consider Index Size:
Keep in mind that indexes consume storage space. Too many indexes or indexing large columns can significantly increase storage requirements. Balance the benefits of improved query performance against the storage costs.

6. Drop or Modify Indexes:
If you find that an index is not being used or is no longer necessary, you can drop it using the '**DROP INDEX**' statement:

```
DROP INDEX index_name;
```

You can also modify existing indexes, such as adding or removing columns from composite indexes, depending on your DBMS.

7. Test Query Performance:
Before and after adding indexes, test the performance of your queries to ensure that they have indeed improved. Profiling tools and query execution plans can help you evaluate the effectiveness of your indexing strategy.

Queries

- Concepts of Transactions
- ACID Property of Transaction Constraints.

Concepts of Transactions

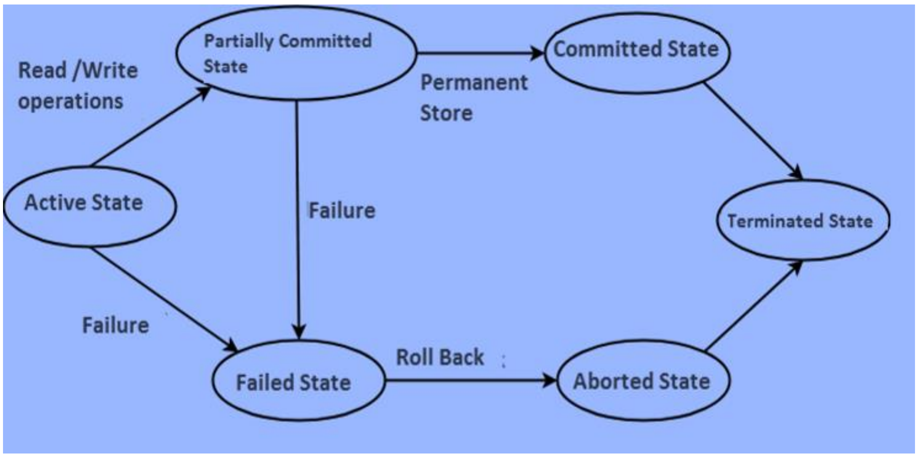
A transaction is a fundamental concept in DBMS that represents a unit of work involving one or more database operations (e.g., INSERT, UPDATE, DELETE, SELECT). Here are some key concepts associated with transactions:

Properties of Transaction (ACID Properties)

To carry out a transaction within a Database Management System (DBMS), it must exhibit a set of characteristics commonly referred to as the ACID properties.

- A – Atomicity
- C – Consistency
- I – Isolation
- D – Durability

Transaction States



Transactions in database management systems (DBMS) have several key properties that define their behavior and ensure data integrity. These properties are often referred to as the ACID properties:

1. **Atomicity (A):** Atomicity ensures that a transaction is treated as a single, indivisible unit of work.

It follows the "all or nothing" principle, meaning that all the operations within a transaction are either fully completed or fully undone in case of an error or failure.

If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged.

Atomicity ensures that the database remains in a consistent state.

2. **Consistency (C):** Consistency guarantees that a transaction brings the database from one valid state to another valid state.

This means that a transaction must adhere to all integrity constraints, rules, and validations defined in the database schema.

If a transaction violates consistency rules, it is rolled back, and the database remains unchanged. Consistency ensures that data remains in a coherent state during and after the transaction.

3. **Isolation (I):** Isolation defines the degree to which one transaction is isolated from the effects of other concurrent transactions.

It ensures that concurrent transactions do not interfere with each other and maintains data integrity.

Different isolation levels (e.g., Read Uncommitted, Read Committed, Repeatable Read, Serializable) specify the level of isolation between concurrent transactions.

Higher isolation levels provide stronger guarantees but may impact performance due to locking and resource contention.

4. **Durability (D):** Durability guarantees that once a transaction is committed, its effects are permanent and will survive any subsequent system failures, such as power outages, crashes, or hardware failures.

This is typically achieved by writing transaction changes to non-volatile storage (e.g., disk) and maintaining a transaction log for recovery purposes. Durability ensures data persistence and reliability.

These ACID properties collectively provide a framework for ensuring the reliability, consistency, and integrity of database transactions, even in complex and concurrent environments. They are essential in applications where data accuracy is critical, such as financial systems, e-commerce platforms, and any scenario where data integrity is paramount. Transactions that adhere to the ACID properties can be trusted to maintain data integrity and consistency.

States of Transaction

Transactions in a database management system (DBMS) go through various states during their lifecycle. These states represent the progress and outcome of a transaction. The common states of a transaction include:

1. **Active (or Running):** The transaction is in the active state when it is executing its operations. During this phase, the transaction is interacting with thedatabase, reading, and writing data. It may issue one or more SQL statements.

2. **Partially Committed (or Preparing):** After the transaction has executed its operations successfully, it enters the partially committed state. In this state, the DBMS prepares to make the changes permanent but has not yet committed the transaction. The system ensures that all necessary conditions are met before moving to the committed state.

3. **Committed:** In the committed state, the transaction has been successfully completed, and all its changes have been made permanent in the database. Once a transaction is committed, its changes are considered permanent and cannot be undone.

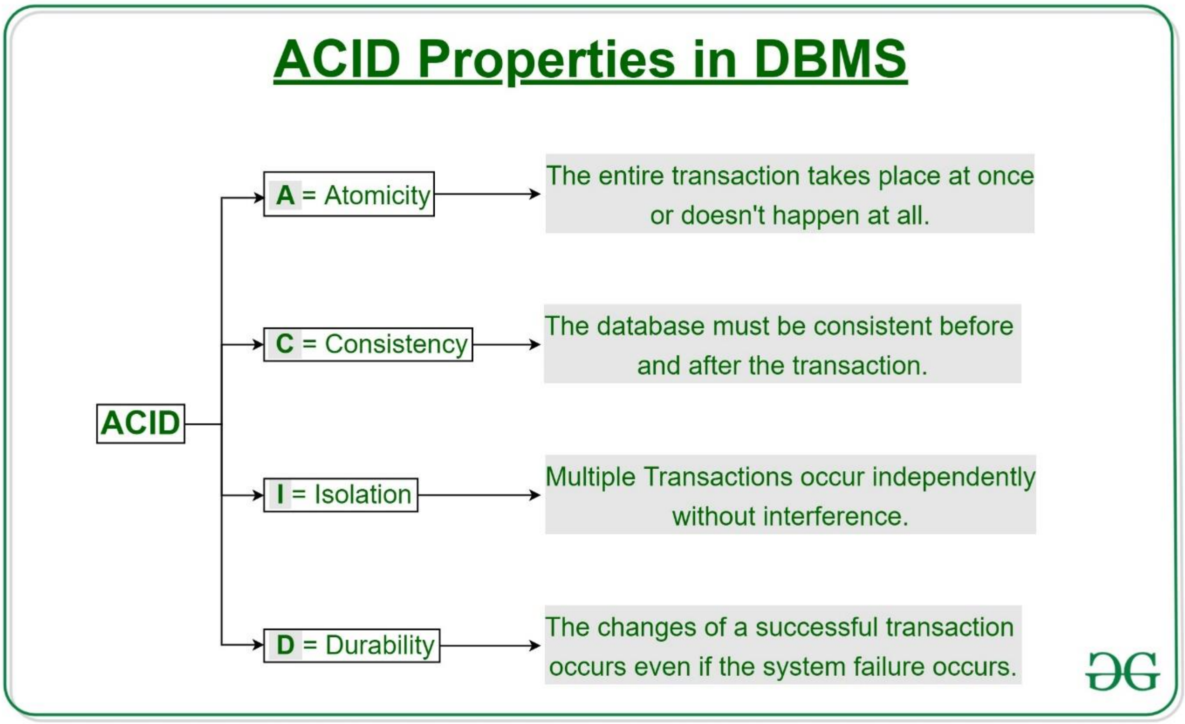
4. **Aborted (or Rolled Back):** If an error occurs during the execution of the transaction or if the transaction encounters a condition that causes it to fail, it may enter the aborted state. In this state, the transaction is rolled back, which means that any changes it made to the database are undone, and the database is restored to its previous state.

5. **Failed:** The failed state is different from the aborted state. A transaction enters the failed state when it encounters a critical error that prevents it from continuing. In this case, the DBMS automatically rolls back the transaction to maintain data integrity and consistency.

6. **Terminated (or Completed):** After a transaction has reached a terminal state (committed, aborted, or failed), it is considered terminated. The transaction can no longer be modified or interact with the database. It may still be examined for auditing or debugging purposes.

7.**Pending:** In some systems, a transaction may enter a pending state temporarily. This typically occurs when a transaction is waiting for a resource or a lock that is held by another transaction. While in the pending state, the transaction is not actively executing but is waiting for its turn to proceed.

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



The entire transaction takes place at once or doesn't happen at all.

The database must be consistent before and after the transaction.

Multiple Transactions occur independently without interference.

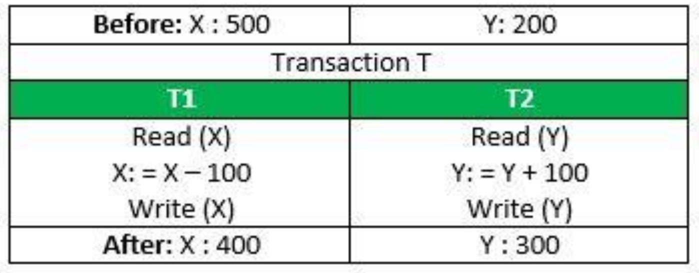
The changes of a successful transaction occurs even if the system failure occurs.

Atomicity:

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is

considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
—**Abort**: If a transaction aborts, changes made to the database are not visible.
—**Commit**: If a transaction commits, changes made are visible.
Atomicity is also known as the ‘All or nothing rule’.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.



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

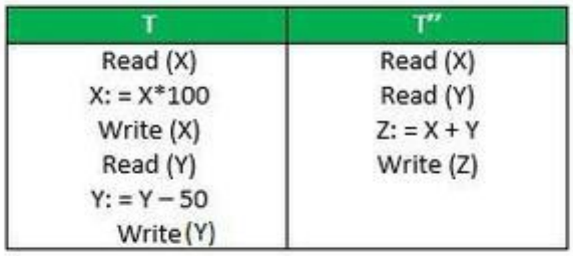
If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its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

Consistency:

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

Isolation:

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



T	T”
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

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

Durability:

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

Some important points:

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

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

ACID properties are the four key characteristics that define the reliability and consistency of a transaction in a Database Management System (DBMS). The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. Here is a brief description of each of these properties:

1. Atomicity: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its original state, ensuring data consistency and integrity.

2. Consistency: Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database is in a consistent state both before and after the transaction is executed. Constraints, such as unique keys and foreign keys, must be maintained to ensure data consistency.
3. Isolation: Isolation ensures that multiple transactions can execute concurrently without interfering with each other. Each transaction must be isolated from other transactions until it is completed. This isolation prevents dirty reads, non-repeatable reads, and phantom reads.
4. Durability: Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The transaction's changes are saved to the database permanently, and even if the system crashes, the changes remain intact and can be recovered.

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

Advantages of ACID Properties in DBMS:

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

Disadvantages of ACID Properties in DBMS:

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

Joins and Functions

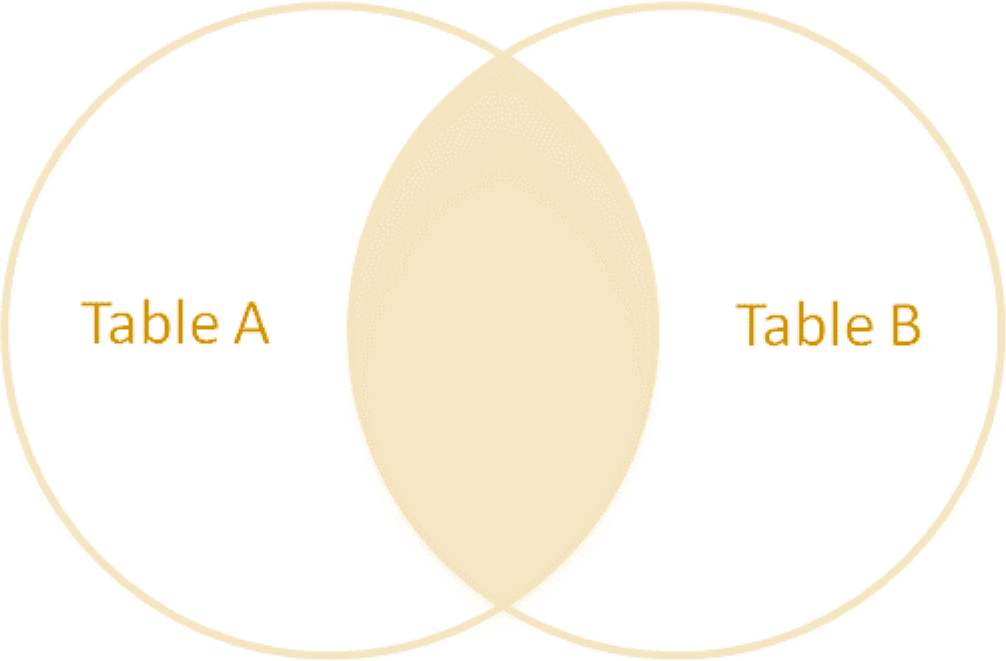
- Joining of tables
- Sub Queries
- Functions used in query like sum, average, max, min, count etc.
- Indexing and Query Optimization.

Joining of tables

Joining tables is a fundamental operation in relational databases that allows you to combine data from two or more tables based on a related column. This operation is crucial for querying and analyzing data stored in a database. There are several types of joins in SQL, the most common being INNER JOIN, LEFT JOIN (or LEFT OUTER JOIN), RIGHT JOIN (or RIGHT OUTER JOIN), and FULL JOIN (or FULL OUTER JOIN).

1.INNER JOIN:

- Returns only the rows that have matching values in both tables.
- Rows that do not have a match in the other table are excluded from the result set.



Syntax

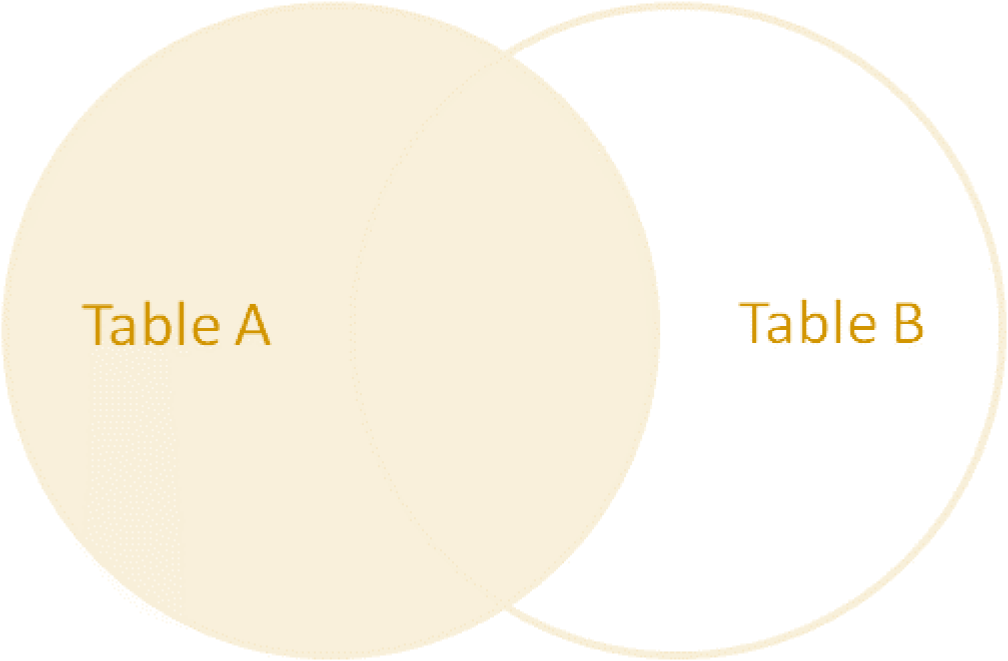
SELECT *columns*

FROM *table1*

INNER JOIN *table2* ON *table1.column = table2.colum*

2.LEFT JOIN (LEFT OUTER JOIN):

- Returns all rows from the left table and the matched rows from the right table.
- If there is no match in the right table, NULL values are included for the columns from the right table.

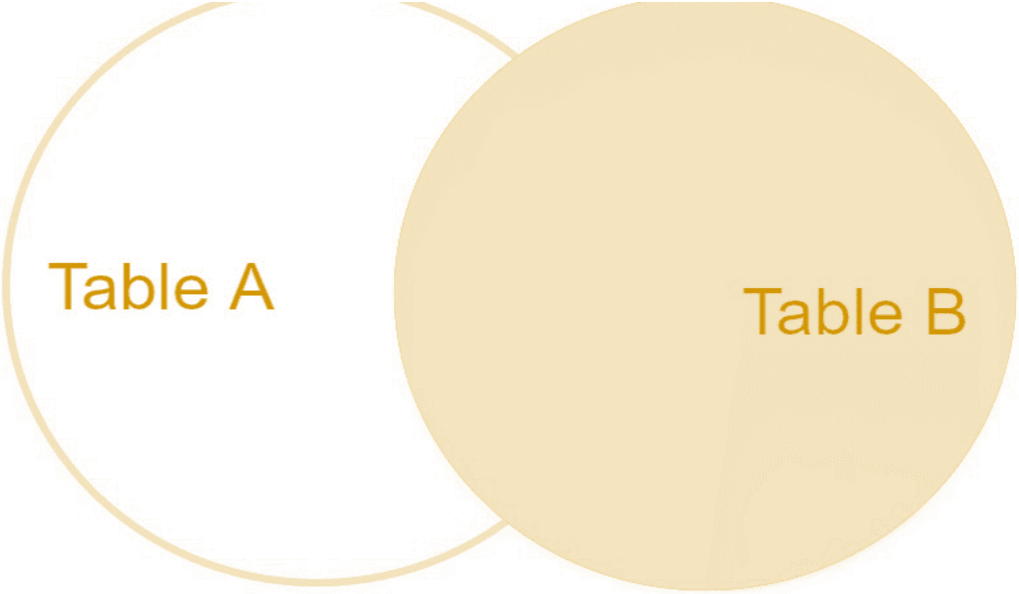


Syntax:-

```
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.column = table2.column;
```

3.RIGHT JOIN (RIGHT OUTER JOIN):

- Returns all rows from the right table and the matched rows from the left table.
- If there is no match in the left table, NULL values are included for the columns from the left table



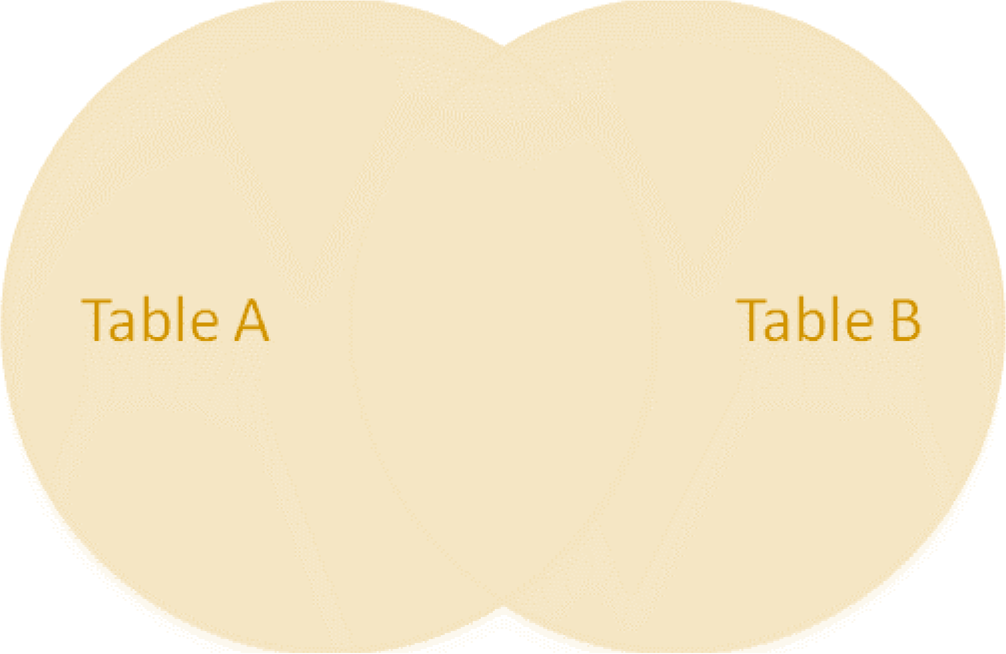
Syntax:-

```
SELECT columns
FROM table1
```

```
RIGHT JOIN table2 ON table1.column = table2.column;
```

4.FULL JOIN (FULL OUTER JOIN):

- Returns all rows when there is a match in either the left or the right table.
- If there is no match in one of the tables, NULL values are included for the columns from the table without a match.



Syntax:-

```
SELECT columns
FROM table1
FULL JOIN table2 ON table1.column_name = table2.column_name;
```

When performing a join, it's essential to specify the columns on which you want to join the tables. These columns should have compatible data types or can be explicitly converted to compatible data types.

Here's a simple example using two hypothetical tables, "orders" and "customers," to demonstrate an INNER JOIN:

```
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

This query retrieves the order IDs and customer names for orders that have matching customer IDs in both the "orders" and "customers" tables.

Sub Queries

A subquery in SQL is essentially a query nested within another query, commonly found within the WHERE clause of the main SQL query. Here are some fundamental rules and guidelines for using subqueries:

1. **Purpose of Subqueries**: Subqueries are employed to retrieve data that will be used by the main query for filtering, comparison, or as a part of calculations.

2. **Location**: Subqueries are typically placed within parentheses and inside the WHERE clause of the main query. They can also be used in other clauses like SELECT, FROM, or HAVING when necessary.

3. **Comparison Operators**: Subqueries often use comparison operators such as =, <, >, <=, >=, or IN to relate the results of the subquery with the main query.

4. **Single-Row or Multi-Row Results**: Subqueries can return either a single value or a set of values (single-row or multi-row results). The choice depends on the specific use case and the operator used.

5. **Subquery Types**: Subqueries can be categorized into various types, including scalar subqueries (returning a single value), single-row subqueries (returning one row), and multi-row subqueries (returning multiple rows).

6. **Alias and Correlation**: When using subqueries in the SELECT clause, it's often necessary to assign an alias to the subquery. Additionally, correlated subqueries refer to the main query's tables, allowing for more complex relationships between the subquery and the main query.

7. **Performance Considerations**: Be cautious when using subqueries, especially correlated subqueries, as they can impact query performance. In some cases, alternative methods like JOINs or CTEs (Common Table Expressions) might be more efficient.

8. **Subquery Limitations**: Some database systems have limitations on the complexity and nesting depth of subqueries. Always consult the documentation for your specific database system to understand its capabilities and limitations regarding subqueries.

Syntax:

While there isn't a single, universal syntax for subqueries in SQL because their usage can vary depending on the specific query and database system, subqueries are commonly used in conjunction with the SELECT statement. Here's a general template for a subquery within a SELECT statement:

```
SELECT column1, column2, ...

FROM table1

WHERE columnN operator (SELECT columnX FROM tableY WHERE condition);
```

In this template:

- **column1, column2,** etc., represent the columns you want to retrieve in your main query.
- **table1** is the main table you're querying.
- **columnN** is a column from **table1** that you want to compare using an operator.
- **operator** is a comparison operator like **=, <, >, IN**, etc.
- The subquery within parentheses **(SELECT columnX FROM tableY WHERE condition)** is embedded in the WHERE clause and retrieves data from another table **(tableY)** based on a specific condition.

Subqueries with the INSERT Statement

Subqueries can also be used with the INSERT statement in SQL to insert data into a table based on the results of a subquery. This allows you to populate a table with data derived from another table or the result of a subquery. Here's a basic syntax for using subqueries with the INSERT statement:

```
INSERT INTO target_table (column1, column2, ...)

SELECT expression1, expression2, ...

FROM source_table

WHERE condition;
```

In this syntax:

- **target_table** is the table into which you want to insert data.
- **column1, column2,** etc., are the columns in **target_table** where you want to insert data.
- **expression1, expression2,** etc., are expressions or values that you want to insert into the corresponding columns in **target_table.**
- **source_table** is the table or subquery that provides the data you want to insert.
- **condition** is an optional condition that filters the data from the **source_table** before insertion.

Here's an example to illustrate how you might use a subquery with the INSERT statement:

Let's say you have two tables, **employees** and **new_hires,** and you want to insert data into the **employees** table from the **new_hires** table based on a certain condition:

```
INSERT INTO employees (employee_id, employee_name, salary)

SELECT new_id, new_name, new_salary

FROM new_hires

WHERE new_salary > 50000;
```

In this example, the INSERT statement retrieves data from the **new_hires** table, specifically the **employee_id, employee_name**, and **salary** columns, but only for records where the **new_salary** is greater than 50,000. It then inserts this data into the **employees** table

Subqueries with the SELECT Statement

Subqueries can be used with the SELECT statement in SQL to retrieve data from one or more tables based on the results of a nested query. This allows you to create more complex and dynamic queries by incorporating the results of one query into another. Here's the basic syntax for using subqueries with the SELECT statement:

```
SELECT column1, column2, ...

FROM table1

WHERE columnN operator (SELECT columnX FROM tableY WHERE condition);
```

In this syntax:

- **column1, column2,** etc., represent the columns you want to retrieve in your main query.
- **table1** is the main table you're querying.
- **columnN** is a column from **table1** that you want to compare using an operator.
- **operator** is a comparison operator like **'=', '<',' >', 'IN'**, etc.
- The subquery within parentheses **(SELECT columnX FROM tableY WHERE condition**) is embedded in the WHERE clause and retrieves data from another table **(tableY)** based on a specific condition.

Here's an example to illustrate how you might use a subquery with the SELECT statement:

Suppose you have two tables, **employees** and **salaries,** and you want to retrieve the names of employees who earn a salary greater than the average salary in the **salaries** table:

```
ELECT employee_name

FROM employees

WHERE employee_id IN (SELECT employee_id FROM salaries WHERE salary >
(SELECT AVG(salary) FROM salaries));
```

In this example, the main SELECT statement retrieves the **employee_name** from the **employees** table for employees whose **employee_id** matches the result of the subquery. The subquery calculates the average salary from the **salaries** table and then filters employees who earn more than the calculated average.

Subqueries with the UPDATE Statement

Subqueries can also be used with the UPDATE statement in SQL to modify data in a table based on the results of a subquery. This allows you to update records in one table using information from another table or using the results of a subquery. Here's a basic syntax for using subqueries with the UPDATE statement:

```
UPDATE target_table

SET column1 = value1, column2 = value2, ...

WHERE columnN operator (SELECT columnX FROM source_table WHERE condition);
```

In this syntax:

- **target_table i**s the table you want to update.

- **column1, column2**, etc., are the columns in **target_table** that you want to update.
- **value1, value2,** etc., are the new values you want to set for the corresponding columns.
- **columnN** is a column in **target_table** that you want to compare using an operator.
- **operator** is a comparison operator like **=, <, >, IN,** etc.
- The subquery within parentheses **(SELECT columnX FROM source_table WHERE condition)** is embedded in the WHERE clause and retrieves data from another table **(source_table)** based on a specific condition.

Here's an example to illustrate how you might use a subquery with the UPDATE statement:

Suppose you have two tables, **orders** and **customers**, and you want to update the **customer_id** in the **orders** table based on a customer's name from the **customers** table

```
UPDATE orders
SET customer_id = (SELECT customer_id FROM customers WHERE customer_name = 'John Doe')
WHERE order_id = 123;
```

In this example, the UPDATE statement updates the **customer_id** column in the **orders** table with the result of the subquery. The subquery retrieves the **customer_id** from the **customers** table for the customer with the name 'John Doe,' and this value is used to update the specified order (order_id = 123) in the **orders** table.

Subqueries with the DELETE Statement

Subqueries can also be used with the DELETE statement in SQL to remove rows from a table based on the results of a subquery. This allows you to delete specific records in a table using information derived from another table or a subquery's results. Here's a basic syntax for using subqueries with the DELETE statement:

```
DELETE FROM target_table
WHERE columnN operator (SELECT columnX FROM source_table WHERE condition);
```

In this syntax:

- **target_table** is the table from which you want to delete rows.
- **columnN** is a column in **target_table** that you want to compare using an operator.
- **operator** is a comparison operator like **=, <, >, IN**, etc.
- The subquery within parentheses **(SELECT columnX FROM source_table WHERE condition)** is embedded in the WHERE clause and retrieves data from another table **(source_table)** based on a specific condition.
- **condition** is an optional condition that filters the data from the **source_table** before deletion.

Here's an example to illustrate how you might use a subquery with the DELETE statement:

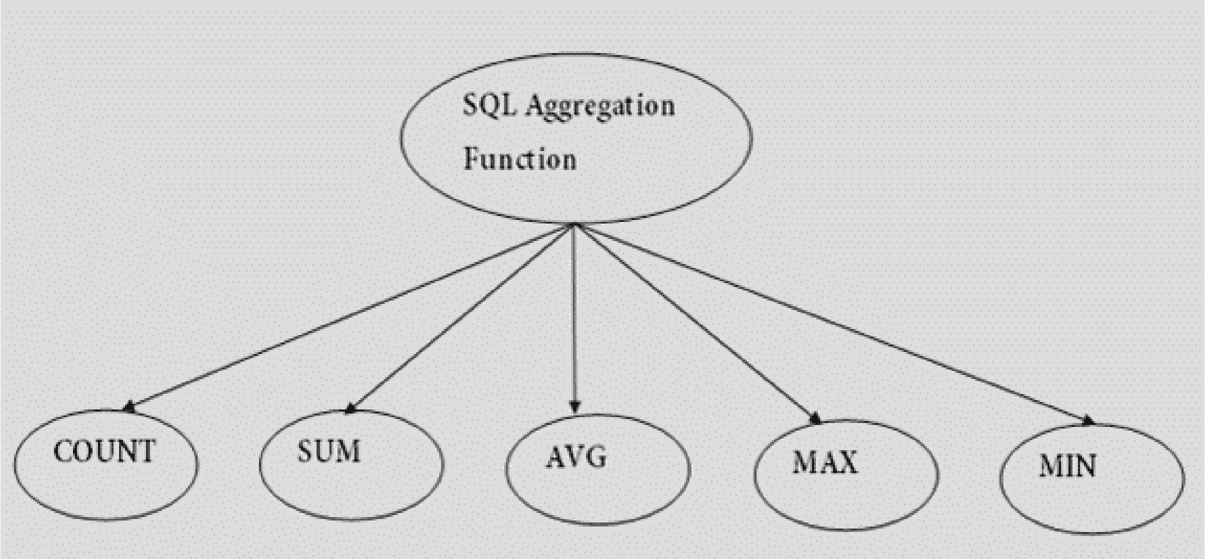
Suppose you have two tables, **employees** and **salaries,** and you want to delete employees from the **employees** table whose salary exceeds the average salary in the **salaries** table:

```
DELETE FROM employees

WHERE employee_id IN (SELECT employee_id FROM salaries WHERE salary > (SELECT AVG(salary) FROM salaries));
```

In this example, the DELETE statement removes rows from the **employees** table where the **employee_id** matches the result of the subquery. The subquery calculates the average salary from the **salaries** table and filters employees with salaries greater than the calculated average.

Functions used in query like sum, average, max, min, count etc

Functions like sum, average, max, min, count, and others are commonly used in various database query languages, such as SQL, to perform calculations and aggregations on data. Here's a brief overview of these functions:



1.SUM: The SUM function is used to calculate the total of a numeric column in a database table. For example, you can use it to find the total sales for a particular product.

```
SELECT SUM(sales_amount) FROM sales_data;
```

2.AVERAGE (AVG): The AVERAGE function, often abbreviated as AVG, calculates the average value of a numeric column.

```
SELECT AVG(temperature) FROM weather_data
```

3.MAX: The MAX function returns the maximum value in a specified column. It is useful for finding the highest value in a dataset.

```
SELECT MAX(score) FROM exam_scores;
```

4.MIN: The MIN function returns the minimum value in a specified column. It is used to find the lowest value in a dataset.

```
SELECT MIN(price) FROM product_prices;
```

5.COUNT: The COUNT function counts the number of rows that meet certain criteria. It can be used to count all rows in a table or to count rows that meet specific conditions.

```
SELECT COUNT(*) FROM employees;-- Count all employees
SELECT COUNT(*) FROM order WHERE sus = 'Shipped'; -- Count shipped orders
```

Certainly! Here are some common SQL functions used in queries like **SUM, AVERAGE, MAX, MIN,** and **COUNT,** along with examples using a hypothetical table called **sales**:

Assume the **sales** table has the following structure:

order_id	product_id	quantity	price
1	101	5	10
2	102	3	15
3	101	2	10
4	103	4	12
5	102	6	15

1.SUM: Calculates the sum of values in a column.

Example: Calculate the total revenue.

```
SELECT SUM(quantity * price) AS total_revenue FROM sales;
```

Result:

total_revenue

180.0

2.AVERAGE: Calculates the average of values in a column.

Example: Calculate the average price of products.

```
SELECT AVG(price) AS avg_price FROM sales;
```

Result:

avg_price

12.4

3.MAX: Finds the maximum value in a column.

Example: Find the maximum quantity sold.

```
SELECT MAX(quantity) AS max_quantity FROM sales;
```

Result:

max_quantity

6

4.MIN: Finds the minimum value in a column.

Example: Find the minimum price of products.

```
SELECT MIN(price) AS min_price FROM sales;
```

Result:

min_price

10.0

5.COUNT: Counts the number of rows in a table or the number of non-null values in a column.

Example: Count the number of orders.

```
SELECT COUNT(*) AS order_count FROM sales
```

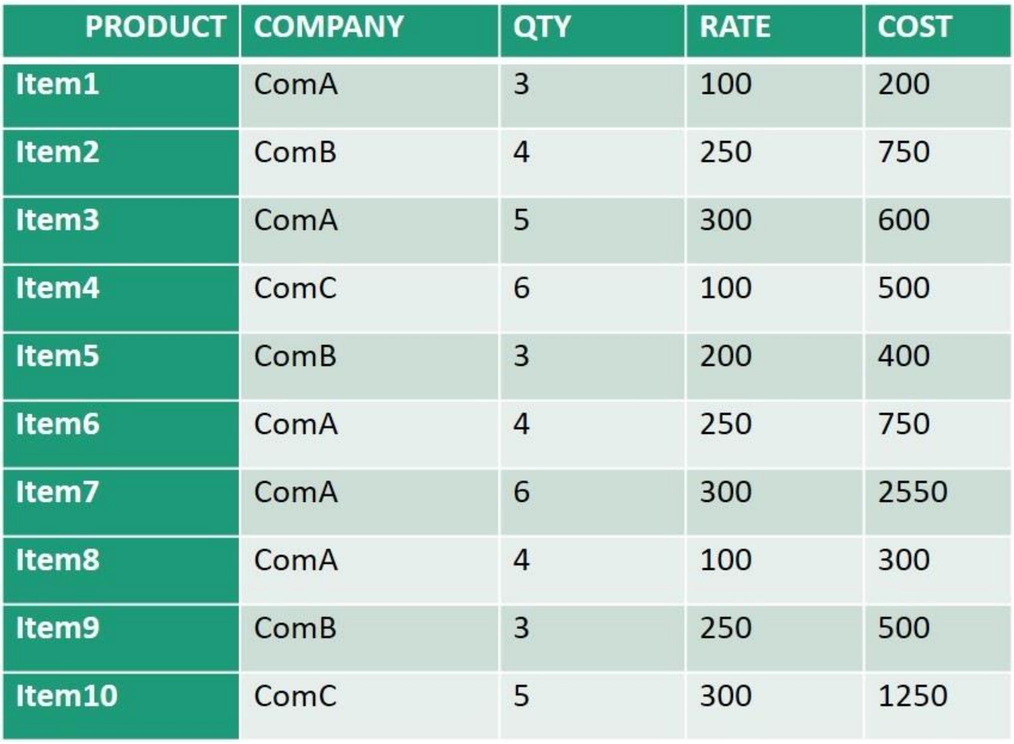
Result:

order_count

5

***Example*-2**

PRODUCT_GOOD



PRODUCT	COMPANY	QTY	RATE	COST
Item1	ComA	3	100	200
Item2	ComB	4	250	750
Item3	ComA	5	300	600
Item4	ComC	6	100	500
Item5	ComB	3	200	400
Item6	ComA	4	250	750
Item7	ComA	6	300	2550
Item8	ComA	4	100	300
Item9	ComB	3	250	500
Item10	ComC	5	300	1250

***Example:* COUNT()**

```
SELECT COUNT(*)
FROM PRODUCT_GOOD;
```

Output:

10

***Example:* COUNT with WHERE**

```
SELECT COUNT(*)
FROM  PRODUCT_GOOD;
WHERE RATE>=200;
```

Output:

7

Example: **COUNT() with DISTINCT**

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_GOOD;
```

Output:

```
3
```

Example: **COUNT() with GROUP BY**

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_GOOD
```

Output:

```
ComA    5
ComB    3
ComC    2
```

Example: **COUNT() with HAVING**

```
SELECT COMPANY, COUNT(*)
SELECT COMPANY, COUNT(*)
GRHAV2;
```

Output:

```
ComA    5
ComB    3
```

Indexing and Query Optimization

Absolutely, your analogy of indexing to the index in a book is quite apt. It provides a clear and intuitive understanding of how indexing works in databases.

In the context of databases, creating an index is akin to building a reference point for the data. Much like flipping through a book's index to quickly locate information, a database index allows for the rapid retrieval of specific data without scanning through the entire dataset. This indexing technique significantly enhances the speed and efficiency of queries, as the database can swiftly pinpoint the relevant information, leading to faster query performance.

The intricacies of how databases achieve this efficiency through indexing can be explored further in the upcoming sections of your article.

How to Create Indexes

Creating indexes in a relational database involves using the **CREATE INDEX** statement.

The syntax may vary slightly between different database management systems (DBMS)

Syntax:

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

Query:

```
CREATE INDEX product_category_index
ON product (category);
```

When you run this query, it will experience a prolonged execution time compared to a standard query. This is because the database is scanning through a substantial 12 million rows and constructing a new 'category' index from the ground up, a process that takes approximately 4 minutes.

Now, let's assess how the performance of the original query improves after the implementation of indexing.

```
SELECT COUNT(*)

FROM product

WHERE category = ‘electronics’;
```

You'll observe a significant improvement in the query's speed this time. It is likely to complete in a much shorter timeframe, perhaps around 400 milliseconds.

Moreover, the positive impact of indexing on 'category' extends beyond just queries explicitly involving this condition. To illustrate, let's consider a scenario where queries involve additional conditions beyond 'category'—even these queries will experience enhanced performance due to the indexing on 'category'.

```
SELECT COUNT(*)

FROM product

WHERE category = ‘electronics’

AND product_subcategory = ‘headphone’;
```

In this case, the query's execution time is expected to be reduced compared to its normal duration, perhaps completing in around 600 milliseconds. The database can efficiently locate all 'electronics' products using the index, resulting in a smaller set of records. Subsequently, it can then identify 'headphones' from this narrowed-down set in the usual manner.

Now, let's explore the impact of changing the order of conditions in the 'WHERE' clause.

```
SELECT COUNT(*)

FROM product

WHERE product_subcategory = ‘headphone’

AND category = ‘electronics’;
```

Types of Indexing

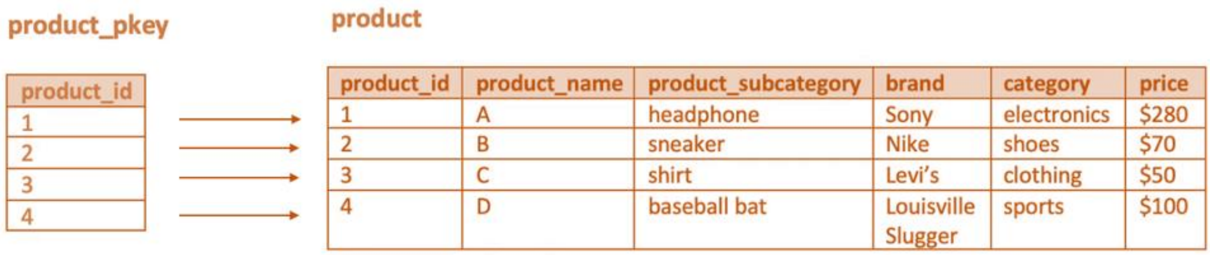
Exploring the realm of database indexing involves delving into two primary types

1.Clustered Index:

- A clustered index stands as the unique index for a table, employing the primary key to structure the data within that table. Unlike a non-clustered index, a clustered index doesn't require explicit declaration; instead, it is automatically generated when the primary key is defined. By default, the clustered index utilizes the ascending order of the primary key for organization.

These clustered indexes play a pivotal role in shaping the physical arrangement of data within the table, facilitating efficient retrieval and storage operations.

Let me demonstrate this with an easy example.



product_pkey

product_id
1
2
3
4

product

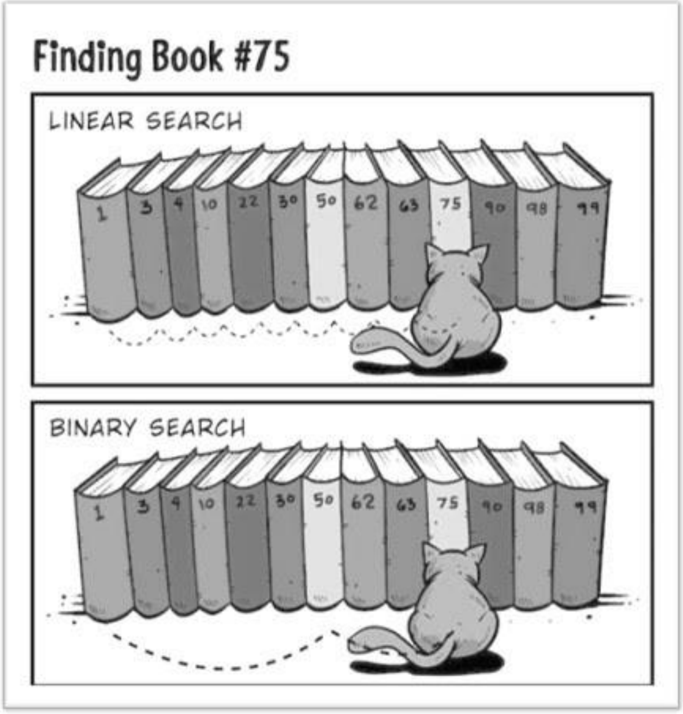
product_id	product_name	product_subcategory	brand	category	price
1	A	headphone	Sony	electronics	$280
2	B	sneaker	Nike	shoes	$70
3	C	shirt	Levi’s	clothing	$50
4	D	baseball bat	Louisville Slugger	sports	$100

The 'product' table will automatically come with a clustered index named 'product_pkey,' and this index is structured around the primary key 'product_id.'

Now, when you run a query to search the table by ‘product_id’ (like in the query below), the clustered index will help the database to perform optimal searches, and return the result faster.

```
SELECT product_name, category, price
FROM product

WHERE product_id = 3;
```

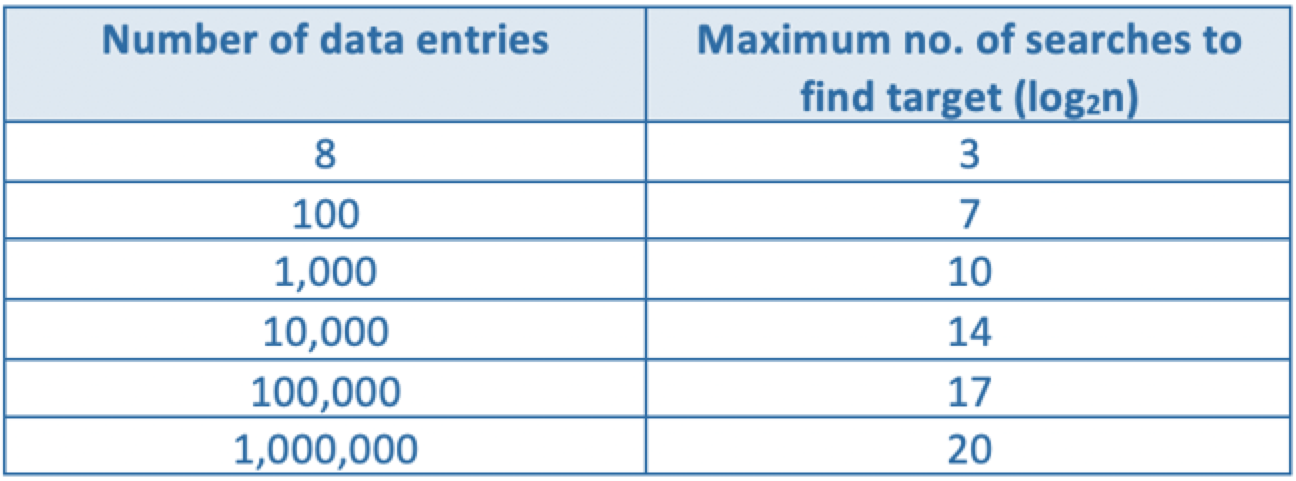
You might be curious about how it accomplishes this. Indices employ an efficient search technique called binary search.



Binary search stands out as a highly effective algorithm for locating an item within a sorted list. Its methodology involves iteratively dividing the data in half and assessing whether the target entry, sought through a query, is positioned before or after the middle entry in the dataset.

If the query value is less than the middle entry, the search narrows down to the lower half; otherwise, it focuses on the upper half.

This process continues until the desired value is located. The brilliance of binary search lies in its ability to minimize the number of searches required, resulting in faster query execution. The following table helps to understand the impact of binary search in terms of number of searches:



Number of data entries	Maximum no. of searches to find target ($\log_2 n$)
8	3
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20

Likewise, in the case of our dataset containing 12 million rows, employing a binary search would necessitate a maximum of 24 searches, as opposed to the worst-case scenario of 12 million searches.

This underscores the formidable efficiency and power of indexes in optimizing data retrieval processes.

2.Non-clustered Index

Now, the challenge is to extend the benefits of indexing beyond the primary key, and the solution lies in non-clustered indexes.

All the queries we initially explored to enhance query performance relied on non-clustered indexes—indexes that need to be explicitly defined.

A non-clustered index is distinct in that it's stored separately from the actual data in the table. It operates much like the index page of a book, as mentioned earlier. The index page is situated in one location, while the contents of the book are in another. This design permits the inclusion of more than one non-clustered index per table, as we discussed earlier.

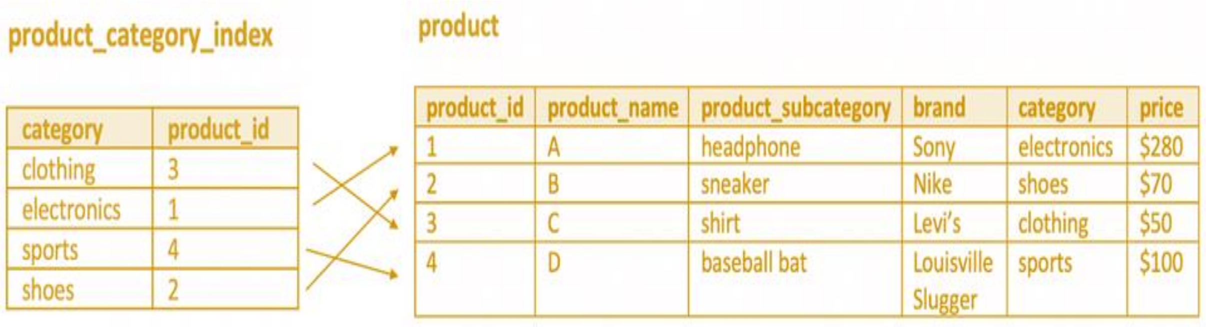
But how is this achieved?

Consider crafting a query that involves searching for an entry in a column for which you've already established a non-clustered index. This type of index inherently encompasses:

1. Column entries for which the index is created.

2. Addresses of the corresponding rows in the main table to which the column entries belong.

You can see this visually in the left mini-table in the figure:



product_category_index

category	product_id
clothing	3
electronics	1
sports	4
shoes	2

product

product_id	product_name	product_subcategory	brand	category	price
1	A	headphone	Sony	electronics	$280
2	B	sneaker	Nike	shoes	$70
3	C	shirt	Levi's	clothing	$50
4	D	baseball bat	Louisville Slugger	sports	$100

Let me explain this using a query.

```
CREATE INDEX product_category_index

ON product (category);

SELECT product_name, category, price

FROM product

WHERE category = ‘electronics’;
```

The database operates through three key steps:

1. **Firstly:** It navigates to the non-clustered index (in this case, 'product_category_index'), pinpointing the column entry you searched for (e.g., category = 'electronics') using the efficient binary search method.

2. **Secondly:** It seeks the address of the corresponding row in the main table that corresponds to the identified column entry.

3. **Finally:** It accesses that specific row in the main table, retrieving additional column values as needed for your query (e.g., product_name, price).

It's important to note that a non-clustered index involves an additional step compared to a clustered index—it requires finding the address and then going to the corresponding row in the main table. This additional step makes non-clustered indexes relatively slower than their clustered counterparts.

```
CREATE TABLE demo(
   id INT NOT NULL,
   first_name VARCHAR(25) NOT NULL,
   last_name VARCHAR(35) NOT NULL,
   age INT NOT NULL,
   PRIMARY KEY(id)
);
/* Index on First Name */
CREATE INDEX demo_fname ON demo (first_name);
/* Will tell us whether our query uses the intended index */
explain SELECT * FROM demo WHERE first_name = "Donald" \G
```

Output

```
*************************** 1. row ***************************
                       id: 1
         select_type: SIMPLE
                   table: demo
            partitions: NULL
                    type: ref
      possible_keys: demo_fname
                       key: demo_fname
        key_len: 27
               ref: const
                    rows: 1
             filtered: 100.00 Extra: NULL
```

To harness the index's benefits, it's crucial to isolate the column, ensuring it's not incorporated into a function or expression.

Stored Procedures, Triggers and Cursors

- Introduction to Stored Procedures.
- Introduction to Triggers and Cursor.
- Creating Trigger
- Creating Cursor
- Using Cursor

Introduction to Stored Procedures.

Stored procedures are a feature in SQL that allows you to define and store a set of SQL statements as a named procedure in a database. These procedures can be called and executed by applications, users, or other database objects. Stored procedures are commonly used for encapsulating business logic, improving code reusability, and enhancing database security. Here's an overview of stored procedures in SQL:

Creating a Stored Procedure:

The syntax for creating a stored procedure can vary slightly between different SQL database management systems (DBMS), but the general structure is similar. Below is a simplified example in generic SQL syntax:

```
CREATE PROCEDURE procedure_name ([parameter_list])

AS BEGIN

-- SQL statements to define the procedure's logic

END;
```

- **procedure_name:** This is the name you give to your stored procedure.
- **[parameter_list]:** You can define input parameters that the procedure can accept. These parameters are optional.

Example of Creating a Simple Stored Procedure in SQL Server:

```
CREATE PROCEDURE GetEmployee @EmployeeID INT

AS

BEGIN

    SELECT * FROM Employees WHERE EmployeeID = @EmployeeID;

END;
```

In this example, we've created a stored procedure called **GetEmployee** that takes an **EmployeeID** as an input parameter and selects the corresponding employee from the **Employees** table in SQL Server.

Executing a Stored Procedure:

To execute a stored procedure, you can use the **EXEC** or **CALL** statement, depending on the specific SQL DBMS you are using.

- In SQL Server, you use **EXEC:**

```
EXEC GetEmployee @EmployeeID = 123;
```

In MySQL, you use **CALL:**

```
CALL GetEmployee(123);
```

Modifying a Stored Procedure:

You can modify an existing stored procedure using the **ALTER PROCEDURE** statement in SQL Server, or the **ALTER PROCEDURE** command in other DBMS, which can vary slightly.

Dropping a Stored Procedure:

To remove a stored procedure, you use the **DROP PROCEDURE** statement:

```
DROP PROCEDURE IF EXISTS GetEmployee;
```

Control Flow in Stored Procedures:

Stored procedures support various control flow constructs like **IF, ELSEIF, ELSE, CASE**, loops (e.g., **WHILE, LOOP, REPEAT**), and exception handling using **BEGIN...END** blocks, depending on the specific DBMS.

Security Considerations:

You can control access to stored procedures by granting or revoking execution privileges to specific users or roles. Properly managing permissions is essential to ensure that only authorized users can execute these procedures

Stored procedures are a valuable feature in SQL that allows you to encapsulate complex database operations, enforce security, and promote code reusability. They are widely used in database-driven applications and are an important part of database management and development.

Creating Trigger, Creating Cursor, Using Cursor

Introduction to Triggers

Certainly! Triggers in the context of databases refer to special types of stored procedures that automatically execute in response to specific events on a particular table or view. These events typically involve data manipulation operations, such as INSERT, UPDATE, DELETE, or even certain schema-level events like CREATE, ALTER, or DROP.

"A trigger is an automated code segment, acting as a procedure, that is executed in response to specific events occurring in a table or view within a database. In contrast, a cursor is a control structure utilized in databases for traversing through records. It's noteworthy that a cursor can be declared and employed within the context of a trigger "

Key Concepts:

1. **Events:**
   - ***INSERT:*** Triggered after a new row is added to the table.
   - ***UPDATE:*** Triggered after one or more existing rows are modified.
   - ***DELETE:*** Triggered after one or more rows are removed from the table.
2. **Timing:**
   - ***BEFORE Triggers:*** Executed before the triggering event, allowing modification of the data before it is actually written to the database.
   - ***AFTER Triggers:*** Executed after the triggering event has occurred.
3. **Row-Level and Statement-Level Triggers:**
   - ***Row-Level Triggers:*** Executed once for each row affected by the triggering event.
   - ***Statement-Level Triggers:*** Executed once for each triggering event, regardless of the number of rows affected.

Use Cases

1. **Data Validation:**
   - Ensure that certain conditions are met before allowing data changes.
2. **Enforcing Business Rules:**
   - Implement complex business logic or rules automatically.
3. **Audit Trails:**

- Log changes made to a table for auditing purposes.

4. **Cascade Operations:**

   - Automatically perform additional operations on other tables when a specified event occurs.

5. **Synchronization:**

   - Keep multiple tables in sync by triggering actions in response to changes in one table.

Syntax (for an AFTER INSERT Trigger):

```
CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
  -- Trigger logic here
END;
```

- **trigger_name**: The name you assign to the trigger.

- **AFTER INSERT ON table_name:** Specifies the timing and event that trigger the execution.

- **FOR EACH ROW:** Indicates that the trigger will be executed once for each row affected by the triggering event.

- **BEGIN...END:** The block where you define the logic of the trigger.

Example:

Let's consider a simple example where we want to create an audit trail for an **employees** table:

```
CREATE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employee_audit (employee_id, action, timestamp)
  VALUES (NEW.employee_id, 'INSERT', NOW());
END;
```

In this example, every time a new row is inserted into the **employees** table, the trigger logs the action into an **employee_audit** table with details like the employee ID, the action performed (INSERT), and the timestamp.

Triggers are powerful tools, but they should be used with caution to avoid unintended consequences and to ensure they do not negatively impact database performance.

Introduction to Cursor

A cursor in the context of databases is a programming construct or a database object that enables the traversal and manipulation of records in a result set. It provides a mechanism for iterating over a set of rows returned by a SQL query. Cursors are often used in procedural languages, such as PL/SQL or T-SQL, to perform operations on a row-by-row basis.

There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors.

1.Implicit Cursors, often referred to as the Default Cursors of SQL Server, are automatically assigned by SQL Server when users execute Data Manipulation Language (DML) operations. These cursors are generated by the system without explicit declaration by the user.

2.explicit cursors:-Users create explicit cursors when needed. These cursors are specifically crafted by users for the purpose of retrieving data from a table in a row-by-row manner.

How To Create Explicit Cursor?

1. Declare Cursor Object

Syntax:

```
DECLARE cursor_name CURSOR FOR

SELECT * FROM table_name
```

Query:

```
DECLARE s1 CURSOR FOR

SELECT * FROM studDetails
```

2. Open Cursor Connection

Syntax:

```
OPEN cursor_connection
```

Query:

```
OPEN s1
```

3.Close cursor connection

Syntax:

```
CLOSE cursor_name
```

Query:

```
CLOSE s1
```

4.Deallocate cursor memory

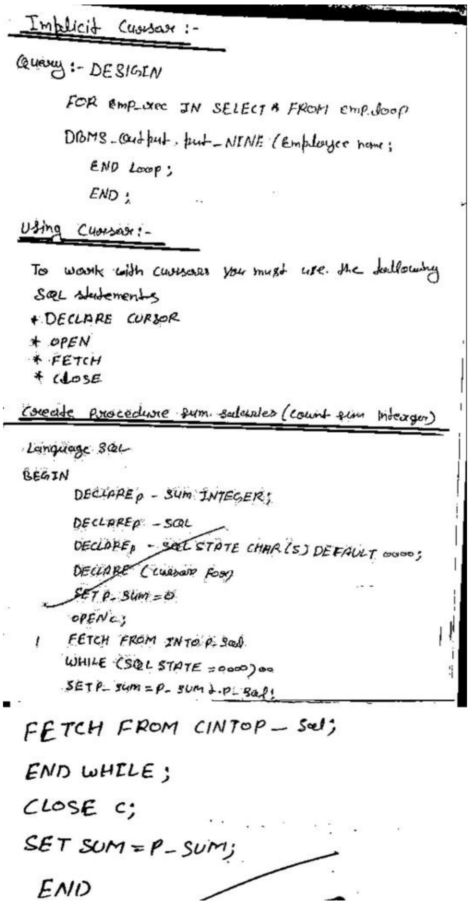
Syntax:

```
DEALLOCATE cursor_name
```

Query:

DEALLOCATE s1

How To Create an Implicit Cursor?



Implicit Cursor :-

Query :- DESIGIN

```
    FOR emp_rec IN SELECT * FROM emp.loop
    DBMS_Output . put_NINE (Employee name ;
        END Loop ;
        END ;
```

Using Cursor :-

To work with cursors you must use the following SQL statements

- * DECLARE CURSOR
- * OPEN
- * FETCH
- * CLOSE

Create procedure sum. salaries (count sum integer)

```
Language SQL
BEGIN
        DECLARE p - sum INTEGER;
        DECLAREp -SQL
        DECLAREp - SQL STATE CHAR(S) DEFAULT 0000;
        DECLARE (cursor for)
        SET P- Sum = 0
        OPEN c ;
  /     FETCH FROM INTO P- Sal
       WHILE (SQL STATE =0000) 00
       SET P- sum = P- sum + P- Sal!
FETCH FROM CINTOP - Sal;
END WHILE ;
CLOSE C;
SET SUM = P-SUM;
 END
```

Syntax (for a Simple Cursor):

-- Cursor Declaration

DECLARE *cursor_name* CURSOR FOR

SELECT *column1, column2*

FROM *table_name*

WHERE condition*;*

-- Cursor Opening

OPEN *cursor_name;*

-- Cursor Fetching and Processing

```
FETCH NEXT FROM cursor_name INTO variable1, variable2;

-- Loop through the result set
WHILE @@FETCH_STATUS = 0
BEGIN
  -- Process the current row (variable1, variable2)

  -- Fetch the next row
  FETCH NEXT FROM cursor_name INTO variable1, variable2;
END;

-- Cursor Closing
CLOSE cursor_name;
```

Example:

Suppose we want to process each employee's name and salary from an **employees** table:

```
DECLARE emp_cursor CURSOR FOR
SELECT employee_name, salary
FROM employees;

OPEN emp_cursor;

FETCH NEXT FROM emp_cursor INTO @employee_name, @salary;

WHILE @@FETCH_STATUS = 0
BEGIN
  -- Process the current employee (e.g., print or update)
  PRINT CONCAT('Employee: ', @employee_name, ', Salary: ', @salary);

  -- Fetch the next employee
  FETCH NEXT FROM emp_cursor INTO @employee_name, @salary;
END;

CLOSE emp_cursor;
```

In this example, the cursor iterates through the result set of the query, fetching the employee name and salary for each row and then processing them within a loop. Cursors provide a flexible mechanism for handling individual rows in a result set within procedural code

In this chapter, we will discuss Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

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

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

Benefits of Triggers

Triggers can be written for the following purposes −

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

Creating Triggers

The syntax for creating a trigger is −

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

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

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

Select * from customers;

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

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

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

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

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

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

```
Trigger created.
```

The following points need to be considered here −

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

Triggering a Trigger

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

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

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

```
Old salary:
New salary: 7500
Salary difference:
```

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

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

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

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

Creating Cursor

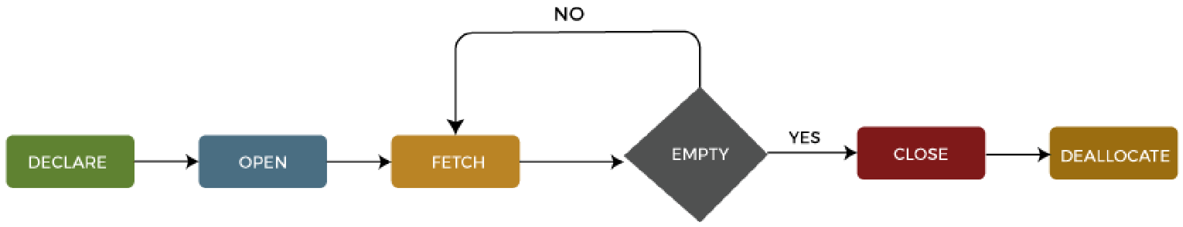
A cursor in SQL Server is a d**atabase object that allows us to retrieve each row at a time and manipulate its data**. A cursor is nothing more than a pointer to a row. It's always used in conjunction with a SELECT statement. It is usually a collection of SQL logic that loops through a predetermined number of rows one by one. A simple illustration of the cursor is when we have an extensive database of worker's records and want to calculate each worker's salary after deducting taxes and leaves.

The SQL Server **cursor's purpose is to update the data row by row, change it, or perform calculations that are not possible when we retrieve all records at once**. It's also useful for performing administrative tasks like SQL Server database backups in sequential order. Cursors are mainly used in the development, DBA, and ETL processes.

This article explains everything about SQL Server cursor, such as cursor life cycle, why and when the cursor is used, how to implement cursors, its limitations, and how we can replace a cursor.

Life Cycle of the cursor

We can describe the life cycle of a cursor into the **five different sections** as follows:



1: Declare Cursor

The first step is to declare the cursor using the below SQL statement:

```
1.   DECLARE cursor_name CURSOR
2.   FOR select_statement;
```

We can declare a cursor by specifying its name with the data type CURSOR after the DECLARE keyword. Then, we will write the SELECT statement that defines the output for the cursor.

2: Open Cursor

It's a second step in which we open the cursor to store data retrieved from the result set. We can do this by using the below SQL statement:

```
1.    OPEN cursor_name;
```

3: Fetch Cursor

It's a third step in which rows can be fetched one by one or in a block to do data manipulation like insert, update, and delete operations on the currently active row in the cursor. We can do this by using the below SQL statement:

```
FETCH NEXT FROM cursor INTO variable_list;
```

We can also use the **@@FETCHSTATUS function** in SQL Server to get the status of the most recent FETCH statement cursor that was executed against the cursor. The **FETCH** statement was successful when the @@FETCHSTATUS gives zero output. The **WHILE** statement can be used to retrieve all records from the cursor. The following code explains it more clearly:

```
WHILE @@FETCH_STATUS = 0
BEGIN
FETCH NEXT FROM cursor_name;
END;
```

4: Close Cursor

It's a fourth step in which the cursor should be closed after we finished work with a cursor. We can do this by using the below SQL statement:

```
CLOSE cursor_name;
```

5: Deallocate Cursor

It is the fifth and final step in which we will erase the cursor definition and release all the system resources associated with the cursor. We can do this by using the below SQL statement:

```
DEALLOCATE cursor_name;
```

Uses of SQL Server Cursor

We know that relational database management systems, including SQL Server, are excellent in handling data on a set of rows called result sets. **For example**, we have a table **product_table** that contains the product descriptions. If we want to update the **price** of the product, then the below '**UPDATE**' query will update all records that match the condition in the '**WHERE**' clause:

```
UPDATE product_table SET unit_price = 100 WHERE product_id = 105;
```

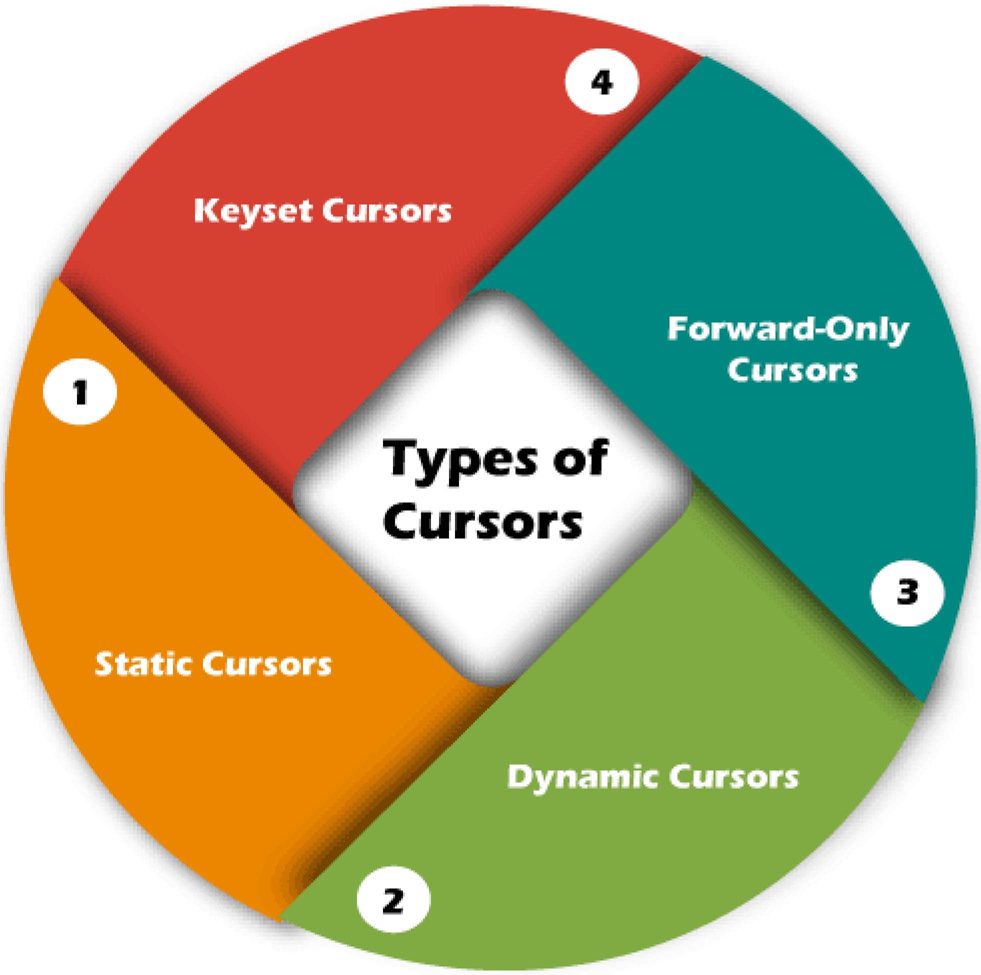
Sometimes the application needs to process the rows in a singleton fashion, i.e., on row by row basis rather than the entire result set at once. We can do this process by using cursors in SQL Server. Before using the cursor, we must know that cursors are very bad in performance, so it should always use only when there is no option except the cursor.

The cursor uses the same technique as we use loops like FOREACH, FOR, WHILE, DO WHILE to iterate one object at a time in all programming languages. Hence, it could be chosen because it applies the same logic as the programming language's looping process.

Types of Cursors in SQL Server

The following are the different types of cursors in SQL Server listed below:

- Static Cursors
- Dynamic Cursors
- Forward-Only Cursors
- Keyset Cursors



Static Cursors

The result set shown by the static cursor is always the same as when the cursor was first opened. Since the static cursor will store the result in **tempdb**, they are always **read-only**. We can use the static cursor to move both forward and backward. In contrast to other cursors, it is slower and consumes more memory. As a result, we can use it only when scrolling is necessary, and other cursors aren't suitable.

This cursor shows rows that were removed from the database after it was opened. A static cursor does not represent any INSERT, UPDATE, or DELETE operations (unless the cursor is closed and reopened).

Dynamic Cursors

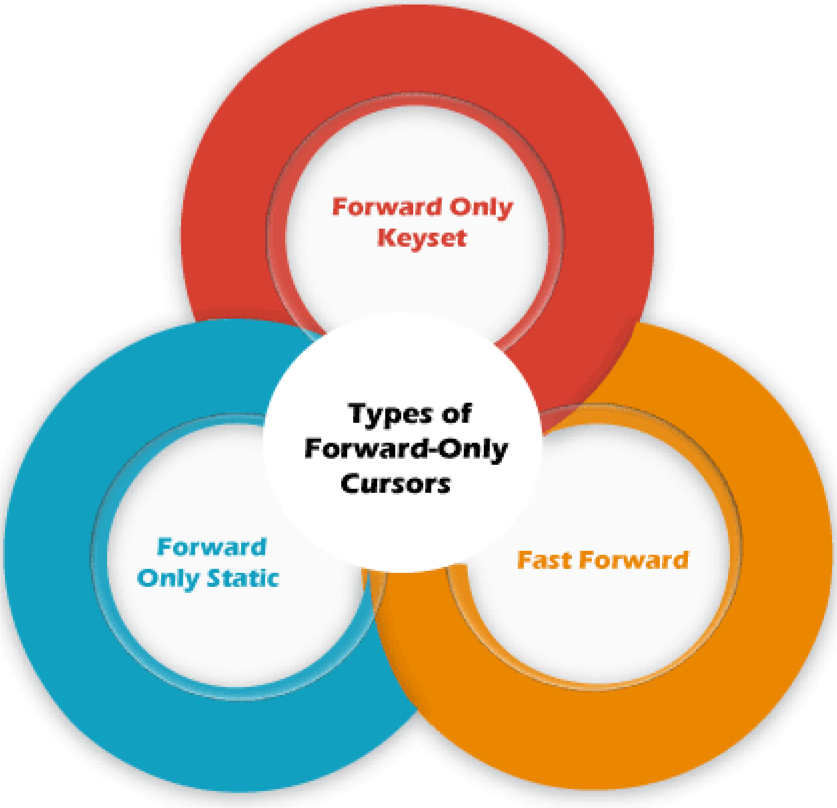
The dynamic cursors are opposite to the static cursors that allow us to perform the data updation, deletion, and insertion operations while the cursor is open. It is **scrollable by default**. It can detect all changes made to the rows, order, and values in the result set, whether the changes occur inside the cursor or outside the cursor. Outside the cursor, we cannot see the updates until they are committed.

Forward-Only Cursors

It is the default and fastest cursor type among all cursors. It is called a forward-only cursor because it **moves only forward through the result set**. This cursor doesn't support scrolling. It can only retrieve rows from the beginning to the end of the result set. It allows us to perform insert, update, and delete operations. Here, the effect of insert, update and delete operations made by the user that affect rows in the result set are visible as the rows are fetched from the cursor. When the row was fetched, we cannot see the changes made to rows through the cursor.

The Forward-Only cursors are three categorize into three types:

1. Forward_Only Keyset
2. Forward_Only Static
3. Fast_Forward



Keyset Driven Cursors

This cursor functionality **lies between a static and a dynamic cursor** regarding its ability to detect changes. It can't always detect changes in the result set's membership and order like a static cursor. It can detect changes in the result set's rows values as like a dynamic cursor. It can only **move from the first to last and last to the first row**. The order and the membership are fixed whenever this cursor is opened.

It is operated by a set of unique identifiers the same as the keys in the keyset. The keyset is determined by all rows that qualified the SELECT statement when the cursor was first opened. It can also detect any changes to the data source, which supports update and delete operations. It is scrollable by default.

Using Cursor

Implementation of Example

Let us implement the cursor example in the SQL server. We can do this by first creating a table named "**customer**" using the below statement:

```
CREATE TABLE customer (
id int PRIMARY KEY,
c_name nvarchar(45) NOT NULL,
email nvarchar(45) NOT NULL,
city nvarchar(25) NOT NULL
);
```

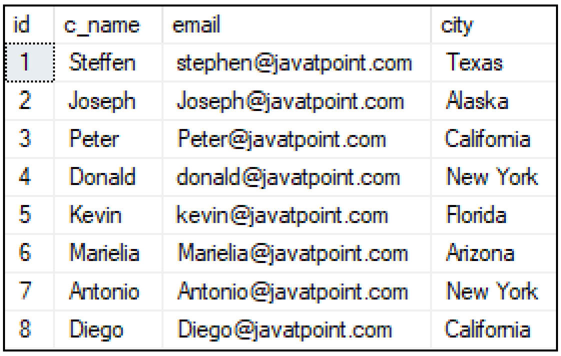
Next, we will insert values into the table. We can execute the below statement to add data into a table:

```
INSERT INTO customer (id, c_name, email, city)
VALUES (1,'Steffen', 'stephen@javatpoint.com', 'Texas'),
(2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
(3, 'Peter', 'Peter@javatpoint.com', 'California'),
(4,'Donald', 'donald@javatpoint.com', 'New York'),
(5, 'Kevin', 'kevin@javatpoint.com', 'Florida'),
(6, 'Marielia', 'Marielia@javatpoint.com', 'Arizona'),
(7,'Antonio', 'Antonio@javatpoint.com', 'New York'),
(8, 'Diego', 'Diego@javatpoint.com', 'California');
```

We can verify the data by executing the **SELECT** statement:

```
SELECT * FROM customer;
```

After executing the query, we can see the below output where we have **eight rows** into the table:



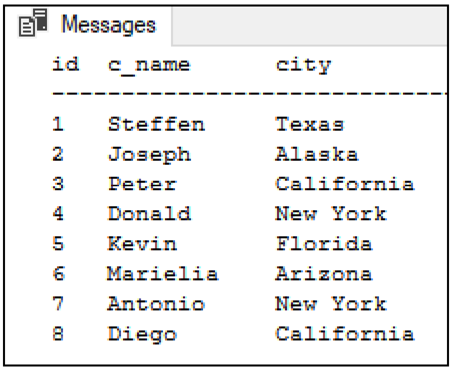
id	c_name	email	city
1	Steffen	stephen@javatpoint.com	Texas
2	Joseph	Joseph@javatpoint.com	Alaska
3	Peter	Peter@javatpoint.com	California
4	Donald	donald@javatpoint.com	New York
5	Kevin	kevin@javatpoint.com	Florida
6	Marielia	Marielia@javatpoint.com	Arizona
7	Antonio	Antonio@javatpoint.com	New York
8	Diego	Diego@javatpoint.com	California

Now, we will create a cursor to display the customer records. The below code snippets explain the all steps of the cursor declaration or creation by putting everything together:

```
--Declare the variables for holding data.
DECLARE @id INT, @c_name NVARCHAR(50), @city NVARCHAR(50)

--Declare and set counter.
DECLARE @Counter INT
SET @Counter = 1

--Declare a cursor
DECLARE PrintCustomers CURSOR
FOR
SELECT id, c_name, city FROM customer

--Open cursor
OPEN PrintCustomers

--Fetch the record into the variables.
FETCH NEXT FROM PrintCustomers INTO
@id, @c_name, @city

--LOOP UNTIL RECORDS ARE AVAILABLE.
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
IF @Counter = 1
BEGIN
PRINT 'id' + CHAR(9) + 'c_name' + CHAR(9) + CHAR(9) + 'city'
PRINT '--------------------------'
END

--Print the current record
PRINT CAST(@id AS NVARCHAR(10)) + CHAR(9) + @c_name + CHAR(9) + CHAR(9) + @city

--Increment the counter variable
SET @Counter = @Counter + 1

--Fetch the next record into the variables.
FETCH NEXT FROM PrintCustomers INTO
@id, @c_name, @city
END

--Close the cursor
CLOSE PrintCustomers

--Deallocate the cursor
DEALLOCATE PrintCustomers
```

After executing a cursor, we will get the below output:



```
id  c_name      city
------------------------------
1   Steffen     Texas
2   Joseph      Alaska
3   Peter       California
4   Donald      New York
5   Kevin       Florida
6   Marielia    Arizona
7   Antonio     New York
8   Diego       California
```

Limitations of SQL Server Cursor

A cursor has some limitations so that it should always use only when there is no option except the cursor. These limitations are:

- Cursor consumes network resources by requiring a network roundtrip each time it fetches a record.
- A cursor is a memory resident set of pointers, which means it takes some memory that other processes could use on our machine.
- It imposes locks on a portion of the table or the entire table when processing data.
- The cursor's performance and speed are slower because they update table records one row at a time.
- Cursors are quicker than while loops, but they do have more overhead.
- The number of rows and columns brought into the cursor is another aspect that affects cursor speed. It refers to how much time it takes to open your cursor and execute a fetch statement.

How can we avoid cursors?

The main job of cursors is to traverse the table row by row. The easiest way to avoid cursors are given below:

Using the SQL while loop

The easiest way to avoid the use of a cursor is by using a while loop that allows the inserting of a result set into the temporary table.

User-defined functions

Sometimes cursors are used to calculate the resultant row set. We can accomplish this by using a user-defined function that meets the requirements.