MASTER SQL

THE ULTIMATE GUIDE TO DATABASE MANAGEMENT

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

DBMS stands for Database Management System. It is a software system that allows users to store, manage, and retrieve data efficiently and securely. A DBMS provides an interface between the end users or application programs and the underlying database, enabling users to interact with the data without having to worry about the underlying details of data storage and organization.

Here are some key features and components of a typical DBMS:

1. **Data Definition Language (DDL):** It allows users to define the database schema, including creating tables, specifying data types, constraints, and relationships between tables.
2. **Data Manipulation Language (DML):** It provides a set of commands for manipulating the data within the database. Common DML commands include INSERT, UPDATE, DELETE, and SELECT for adding, modifying, deleting, and retrieving data.
3. **Data Query Language (DQL):** It enables users to retrieve and manipulate data using queries. SQL (Structured Query Language) is the most widely used DQL for relational databases.
4. **Data Integrity and Security:** A DBMS ensures data integrity by enforcing constraints and rules defined during the database design. It also provides security features to control user access and protect sensitive data.
5. **Transaction Management:** A DBMS supports the concept of transactions, which are a set of operations treated as a single unit of work. It ensures that transactions are executed reliably and consistently, following the principles of ACID (Atomicity, Consistency, Isolation, Durability).
6. **Concurrency Control:** DBMS handles multiple user requests concurrently while ensuring data consistency and preventing conflicts that may arise due to concurrent access.
7. **Backup and Recovery:** DBMS provides mechanisms for backing up the database periodically to prevent data loss in case of system failures or disasters. It also facilitates recovery to restore the database to a consistent state after failures.

DBMSs are widely used in various applications and industries to handle large volumes of data efficiently. They can be categorized into different types based on their data models, such as relational, hierarchical, network, and object-oriented DBMSs. Relational DBMSs, such as Oracle, MySQL, and PostgreSQL, are the most commonly used ones today.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

General Purpose Of DBMS :

The general purpose of a Database Management System (DBMS) is to efficiently and effectively manage and organize data. Here are some of the key purposes and benefits of using a DBMS:

1. **Data Organization and Storage**: A DBMS provides a structured approach to organize and store data in a logical manner. It allows data to be stored in tables, which are further divided into columns (attributes) and rows (records). This structured organization makes it easier to manage and retrieve data.
2. **Data Integrity and Security:** A DBMS enforces data integrity by applying constraints, such as unique key constraints, referential integrity constraints, and data type constraints. It ensures that data remains consistent and accurate by preventing invalid or inconsistent data from being stored in the database. DBMSs also provide security mechanisms to control user access, authenticate users, and protect sensitive data from unauthorized access.
3. **Data Sharing and Collaboration:** A DBMS enables multiple users and applications to access and share data concurrently. It provides mechanisms for concurrent access and manages data conflicts that may arise due to simultaneous operations. This promotes collaboration and data sharing among users and allows multiple applications to access the same data efficiently.
4. **Data Manipulation and Querying:** A DBMS offers a range of data manipulation capabilities, allowing users to insert, update, delete, and retrieve data from the database. It provides query languages, such as SQL, that allow users to express complex queries to retrieve specific data based on various criteria. These capabilities make it easier to manage and analyze data.
5. **Data Consistency and Data Independence:** DBMS ensures data consistency by enforcing ACID properties (Atomicity, Consistency, Isolation, Durability) for transactions. It ensures that all changes made to the database within a transaction are either applied in full or not at all, maintaining the consistency of data. DBMS also provides data independence, allowing changes in the database structure (schema) to be made without affecting the applications that use the data.
6. **Data Scalability and Performance:** DBMSs are designed to handle large volumes of data efficiently. They provide mechanisms for indexing, caching, and optimizing queries to enhance performance. DBMSs also support scalability by allowing data to be distributed across multiple servers or clusters to handle increasing data loads.

Overall, the general purpose of a DBMS is to provide a reliable, secure, and efficient solution for managing and accessing data, enabling organizations and individuals to store, retrieve, and manipulate data effectively for various applications and requirements.

Benefits of DBMS :

Database Management Systems (DBMS) offer numerous benefits compared to traditional file management systems. Here are some key advantages of using a DBMS:

1. **Data Centralization:** DBMS enables centralized storage and management of data. Instead of scattered files and applications, all data is stored in a single, integrated database. This centralization facilitates data access, maintenance, and administration, making it easier to manage and control data effectively.
2. **Data Consistency and Integrity:** DBMS enforces data consistency and integrity through various mechanisms such as constraints, referential integrity, and transaction processing. It ensures that data remains accurate, reliable, and coherent, even when accessed and modified by multiple users concurrently.
3. **Data Sharing and Collaboration**: DBMS supports concurrent data access by multiple users and applications. It provides mechanisms for managing concurrent access, data locking, and data sharing, allowing users to collaborate and work with the same data simultaneously. This promotes teamwork, improves productivity, and reduces data redundancy.
4. **Data Security and Privacy**: DBMS offers robust security features to protect data against unauthorized access, ensuring data confidentiality and privacy. Access control mechanisms, user authentication, and encryption techniques can be implemented to restrict access and protect sensitive information.
5. **Data Independence**: DBMS provides a separation between the physical data storage and the logical view of the data. This data independence allows changes in the database schema or organization without affecting the applications that use the data. It simplifies maintenance, upgrades, and modifications, reducing the impact of structural changes on existing applications.
6. **Data Querying and Analysis**: DBMS provides powerful query languages, such as SQL (Structured Query Language), to retrieve, manipulate, and analyze data. These languages offer a standardized and efficient way to express complex queries and perform operations on the data. It enables users to extract valuable insights, generate reports, and make informed decisions based on the data.
7. **Data Scalability and Performance**: DBMSs are designed to handle large volumes of data efficiently. They provide optimization techniques, indexing, and caching mechanisms to enhance performance and ensure scalability as data sizes and user loads increase. Scaling the system can be achieved by adding more hardware resources or distributing data across multiple servers.
8. **Data Backup and Recovery**: DBMS facilitates regular data backups and provides mechanisms for data recovery in case of system failures, errors, or disasters. It ensures that data remains protected and can be restored to a consistent state, minimizing the risk of data loss and downtime.
9. **Data Integrity Constraints and Validation**: DBMS allows the definition of integrity constraints to enforce rules and restrictions on data values. This helps to maintain data quality by preventing the insertion of invalid or inconsistent data into the database.

10. **Data Persistence and Durability**: DBMS ensures the durability of data by providing mechanisms for durable storage, transaction logging, and write-ahead logging. It ensures that committed changes are persisted and survive system failures or crashes, maintaining the reliability and availability of data.

These benefits of DBMS contribute to improved data management, enhanced productivity, better decision-making, and increased data reliability and security in various applications and industries.

Drawbacks Of File Management System:

File Management Systems (FMS) have several drawbacks compared to modern Database Management Systems (DBMS). Here are some of the limitations and drawbacks of file management systems:

1. **Data Redundancy**: In an FMS, data is often duplicated across multiple files and applications. This redundancy can lead to inconsistencies and data integrity issues. If the same data needs to be updated or modified in multiple files, it can be challenging to ensure that all copies of the data remain synchronized.
2. **Data Inconsistency**: Since FMS lacks centralized control over data, inconsistencies can occur if multiple users or applications access and modify the same data simultaneously. Inconsistent updates or partial updates can lead to data integrity problems and incorrect results.
3. **Lack of Data Integration**: FMS typically lack the ability to integrate and combine data from different sources easily. It can be challenging to retrieve and analyze data across multiple files or applications, making it difficult to gain a holistic view of the data.
4. **Limited Data Sharing**: FMSs do not provide robust mechanisms for concurrent data access and sharing. File locking and manual coordination are often required to prevent conflicts when multiple users or applications attempt to access or modify the same file simultaneously. This limitation hampers collaboration and efficiency.
5. **Poor Data Security**: FMSs often lack advanced security features to protect data. File-level security measures, such as access control lists or permissions, are typically limited and may not provide adequate protection against unauthorized access or data breaches.
6. **Lack of Data Independence**: In FMS, the application programs are closely tied to the physical structure and organization of the files. Any changes in the file structure or organization require modifying the application programs that use those files. This lack of data independence makes maintenance and evolution of the system more challenging and time-consuming.
7. **Limited Data Recovery and Backup**: FMSs may not provide comprehensive backup and recovery mechanisms. In case of system failures or data corruption, it can be difficult to recover lost or corrupted data, resulting in potential data loss and downtime.

8. **Difficulty in Data Manipulation and Querying**: FMSs often lack a standardized query language and data manipulation capabilities. Retrieving specific data or performing complex queries can be cumbersome and may require writing custom code or scripts.
9. **Lack of Scalability**: FMSs may struggle to handle increasing volumes of data and user demands efficiently. Scaling the system to accommodate growth can be challenging and may require significant manual effort.

These drawbacks led to the development of DBMSs, which offer solutions to many of these limitations and provide more efficient, secure, and flexible management of data.

What Is Meta Data In DBMS

In DBMS, metadata refers to the data that provides information about the database itself. It describes the structure, organization, and characteristics of the data stored in the database. Metadata is essential for managing, understanding, and utilizing the data within a database. Here are some key aspects of metadata in DBMS:

1. **Database Schema**: Metadata includes information about the database schema, which defines the logical structure of the database. It describes the tables, attributes (columns), data types, constraints, relationships, and other elements that make up the database schema.
2. **Data Definitions**: Metadata contains data definitions that specify the characteristics and properties of the data elements, such as field names, lengths, formats, and allowed values. It provides the necessary information to interpret and manipulate the data correctly.
3. **Data Relationships**: Metadata defines the relationships and dependencies between different data elements within the database. It describes how tables are related through primary key and foreign key relationships, allowing for proper data integration and querying.
4. **Indexes and Performance**: Metadata includes information about the indexes defined on tables. It specifies the fields included in indexes, their order, and the indexing technique used. This information helps the DBMS optimize query execution by determining the most efficient access paths to retrieve data.
5. **Data Access and Security**: Metadata contains access control information, specifying which users or roles have permission to access, modify, or delete specific data. It helps enforce security policies and restrict unauthorized access to sensitive data.
6. **Data History and Auditing**: Metadata can include information about data history and auditing, such as when a record was created, modified, or deleted, and by whom. This enables tracking and auditing changes made to the data, providing accountability and supporting compliance requirements.
7. **Data Integration and Interoperability**: Metadata facilitates data integration by providing information about external data sources, data mappings, and transformations. It allows for seamless integration of data from disparate sources and promotes interoperability between different systems.

8. **Data Documentation and Data Lineage**: Metadata serves as a documentation source for the database, providing information about the purpose, meaning, and usage of various data elements. It also helps track the lineage of data, tracing its origin and transformation processes.
9. **Query Optimization**: Metadata plays a crucial role in query optimization. It contains statistical information about data distribution, cardinality, and data access patterns. This information helps the query optimizer choose the most efficient query execution plan, improving query performance.
10. **System Configuration and Administration**: Metadata includes information about the database system configuration, such as storage settings, buffer sizes, and other system-level parameters. It assists database administrators in managing and tuning the database system.

In summary, metadata in DBMS is the information about the database structure, data definitions, relationships, access control, and other properties. It helps in managing, understanding, and utilizing the data effectively and efficiently.

3 Broad Classes of Users:

In a database management system (DBMS), users can be broadly classified into three main categories based on their roles and level of interaction with the system. These categories are:

1. **End Users**: End users are individuals who interact with the database system to retrieve, input, and update data. They are typically non-technical users who access the database through front-end applications or interfaces. End users include:
    - **Casual End Users:** These users have limited interaction with the database and typically access pre-defined reports or data views to retrieve information.
    - **Naive End Users**: Naive end users have little or no knowledge of the underlying data model or database structure. They use query tools or forms to retrieve and enter data without having to write complex queries.
    - **Parametric End Users**: Parametric end users have a slightly higher level of understanding compared to naive end users. They can define and execute simple ad-hoc queries or parameterize pre-defined queries to retrieve specific information.

2. **Application Programmers/Developers**: Application programmers or developers are responsible for designing, developing, and maintaining software applications that interact with the database. They create the front-end interfaces and backend code to handle data manipulation and integration with the database. These users include:

   - **Database Application Developers**: These developers design and build applications that utilize the database for data storage and retrieval. They write code or use development tools to implement business logic, data processing, and user interfaces.

   - **Database Designers**: Database designers are involved in the process of designing the database schema, tables, relationships, and constraints. They work closely with application developers to ensure the database meets the requirements of the application.

3. **Database Administrators (DBAs):** Database administrators are responsible for the overall management, administration, and maintenance of the database system. They handle tasks such as database installation, configuration, performance tuning, security management, backup and recovery, and ensuring data integrity. DBAs include:

   - **System DBAs**: System DBAs focus on the installation, configuration, and overall management of the database system. They monitor system performance, manage system resources, and handle system-level tasks such as backup and recovery.
   - **Application DBAs**: Application DBAs are involved in the management of specific database applications. They work closely with application developers to optimize application performance, implement security measures, and ensure the integrity of the application's data.
   - **Data Administrators**: Data administrators are responsible for data management and governance. They define data standards, policies, and procedures, ensure data quality, and handle data-related issues such as data modeling, data security, and data privacy.

These user categories provide a general classification of users in a DBMS, but the roles and responsibilities may vary depending on the specific organization and context.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

What Happens When We Query Database ?

When a query is executed in a database, several steps are involved to process and retrieve the requested data. Here is a high-level overview of what typically happens when a query is executed in a database:

1. **Query Parsing and Analysis**: The database management system (DBMS) receives the query and performs a parsing and analysis phase. It checks the query syntax and validates it against the database's grammar rules. If the query is syntactically correct, the DBMS proceeds to analyze the query structure and determine the best execution plan.
2. **Query Optimization**: The DBMS performs query optimization to determine the most efficient execution plan for retrieving the requested data. This involves considering various factors such as available indexes, statistics about the data distribution, and query cost estimation. The query optimizer selects an optimal execution plan to minimize the query's execution time and resource usage.
3. **Accessing Data**: The DBMS determines how to access the data based on the execution plan. It may involve accessing one or more tables or other database objects. The DBMS retrieves data from the storage, typically using data access methods like index scans, table scans, or a combination of both. If necessary, the DBMS performs joins to combine data from multiple tables.
4. **Data Filtering and Transformation**: As the data is accessed, the DBMS applies any specified conditions or filters in the query to retrieve only the relevant data. It may involve comparing values, evaluating logical expressions, or performing calculations. The DBMS also applies any requested data transformations, such as aggregations, sorting, grouping, or projections, to shape the result set as per the query requirements.
5. **Result Set Generation**: Once the data is retrieved and processed, the DBMS generates the result set that matches the query's criteria. It organizes the data in the specified format, such as rows and columns, and applies any requested formatting or calculations.
6. **Result Set Delivery**: The DBMS delivers the generated result set to the application or client that initiated the query. The data is typically transmitted over a network connection in a suitable format, such as a structured dataset or a serialized response.
7. **Query Completion**: After delivering the result set, the DBMS marks the query as completed and frees up any acquired resources. It may update query statistics for future query optimization and logging purposes.

It's important to note that the specific steps and processes involved may vary depending on the DBMS implementation and the complexity of the query. The DBMS's ability to optimize and execute queries efficiently plays a crucial role in determining the query's performance and the overall responsiveness of the database system.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Describe, Data Models & Categories of It

Data models represent the structure, relationships, and constraints of the data stored in a database. They provide a conceptual framework for understanding and organizing data. There are different types or categories of data models based on the data structures they use. Here's a brief description of data models and their categories:

1. **Hierarchical Data Model**: The hierarchical data model organizes data in a tree-like structure, with parent-child relationships. It represents data in a series of records connected through parent-child relationships. Each child record can have only one parent, but a parent can have multiple child records. It is primarily used in legacy systems & is less commonly used in modern database management systems.

2. **Network Data Model**: The network data model is similar to the hierarchical model but allows for more complex relationships between records. It represents data using sets and links between records. Records can have multiple parents, and relationships between records are established through pointers or links. The network model can represent more complex relationships than the hierarchical model, but it can be more difficult to implement and maintain.

3. **Relational Data Model**: The relational data model is the most widely used data model in modern database systems. It organizes data into tables, where each table represents an entity or a relationship. The relational model uses keys to establish relationships between tables, allowing data to be queried and manipulated using Structured Query Language (SQL). The relational model provides a simple, flexible, and scalable approach to representing and managing data.

4. **Object-Oriented Data Model**: The object-oriented data model extends the relational model by incorporating object-oriented programming concepts. It represents data as objects, which encapsulate data and behavior together. Objects can have attributes and methods, and they can inherit characteristics from other objects. The object-oriented model allows for complex data structures and supports inheritance, encapsulation, and polymorphism. It is commonly used in object-oriented programming languages and object-oriented database systems.

5. **Entity-Relationship Model**: The entity-relationship (ER) model is a high-level conceptual model that represents the relationships between entities. It focuses on the entities (objects or concepts) in a system and their relationships, attributes, and constraints. The ER model is often used in the early stages of database design to capture the essential structure of the data and relationships before mapping it to a specific data model.

6. **NoSQL Data Models**: NoSQL (Not Only SQL) data models are designed to handle large-scale, distributed, and unstructured data. They provide flexible schema designs and support different data structures, such as key-value pairs, document-oriented, columnar, and graph-based models. NoSQL databases are commonly used in scenarios where traditional relational models are not the best fit, such as web applications, big data processing, and real-time analytics.

Each data model category has its strengths, weaknesses, and suitable use cases. The choice of data model depends on factors such as the nature of the data, the application requirements, scalability needs, and the level of flexibility and complexity desired.

Levels Of Database Architecture:

The three levels of database architecture are commonly referred to as the **external** level, **internal** level, and **conceptual** level. These levels represent different perspectives or views of the database and provide a way to separate the concerns of different users and stakeholders. Here's a brief description of each level:

1. **External Level (also known as View Level or User Level):** The external level is the highest level of abstraction and represents the view of the database from the perspective of individual users or groups of users. It focuses on specific user requirements and defines individual user views or external schemas. Each external schema describes a subset of the database that is relevant to a particular user or group, hiding the rest of the database's complexity. Users interact with the database through these external schemas, which can include customized views, queries, and access controls tailored to their specific needs.
2. **Conceptual Level (also known as Logical Level):** The conceptual level represents the overall logical structure of the entire database system, independent of any specific implementation or technology. It defines the conceptual schema, which describes the entities, relationships, and constraints that apply to the entire database. The conceptual schema serves as an intermediary between the external views and the internal storage structure. It provides a global, integrated view of the database system that is independent of individual user views or applications. Changes to the conceptual schema can impact all external schemas and applications that use the database.
3. **Internal Level (also known as Storage Level or Physical Level):** The internal level is the lowest level of abstraction and represents the physical implementation details of the database. It describes how the data is stored, accessed, and organized on the storage media, such as disks or memory. The internal level deals with data storage structures, indexing mechanisms, data compression techniques, file organization, and other low-level implementation details. It is concerned with optimizing the database's performance, storage efficiency, and security.

The three levels of database architecture provide a hierarchical and modular approach to database design and management. They allow for data independence, as changes at one level do not necessarily require modifications at other levels. The external level focuses on user views and application requirements, the conceptual level captures the overall database structure, and the internal level deals with physical storage and implementation details.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

What Is Schema ?

In a database management system (DBMS), a schema refers to the overall structure or blueprint of a database. It defines the logical and physical organization of the data, including the tables, attributes (columns), data types, relationships, constraints, and other properties.

Here are a few key aspects of a database schema:

1. **Table Structure**: The schema defines the tables or relations in the database. Each table represents a logical entity, such as a person, product, or order, and consists of rows (records) and columns (attributes). The schema specifies the table names, the attributes in each table, and their corresponding data types.
2. **Relationships**: The schema describes the relationships between tables. Relationships define how tables are related to each other through primary keys, foreign keys, and referential integrity constraints. These relationships help maintain data consistency and enable data retrieval across multiple tables.
3. **Constraints**: The schema includes constraints that enforce rules or conditions on the data. Common constraints include primary key constraints, unique constraints, foreign key constraints, check constraints, and default value constraints. Constraints ensure data integrity, enforce business rules, and maintain the consistency and validity of the data.
4. **Views**: The schema can include views, which are virtual tables derived from one or more tables in the database. Views allow users to define customized, filtered, or aggregated perspectives of the data without modifying the underlying tables. They provide a way to control access to sensitive data and simplify complex queries.
5. **Indexes**: The schema may specify indexes, which are data structures that improve the performance of data retrieval operations. Indexes provide quick access to specific data values based on one or more columns, facilitating faster query execution.
6. **Security and Privileges**: The schema may define access control permissions & privileges for different users or roles. It specifies who can perform specific operations on the database objects, such as reading, writing, modifying, or deleting data. Access control ensures data security & privacy.
7. **Data Definitions**: The schema includes the definitions of data elements, such as field names, lengths, formats, and other characteristics. It provides a clear understanding of the meaning and structure of the data elements stored in the database.

The schema acts as a blueprint that guides the creation, organization, and management of the database. It serves as a foundation for data modeling, application development, data integration, and database administration tasks.
RDBMS

RDBMS stands for **Relational Database Management System**. It is a type of database management system that is based on the relational model of data. In an RDBMS, data is organized into tables, which consist of rows and columns. Each table represents an entity or a relationship between entities.

The key features of an RDBMS include:

1. **Data Integrity**: RDBMS enforces data integrity by supporting constraints such as primary keys, foreign keys, and unique keys. These constraints ensure that data is accurate and consistent.
2. **Structured Query Language** (SQL): RDBMS uses SQL as the standard language for managing and manipulating the data stored in the database. SQL allows users to retrieve, insert, update, and delete data from the database.
3. **ACID Properties**: RDBMS ensures ACID (Atomicity, Consistency, Isolation, Durability) properties for data transactions. Atomicity guarantees that a transaction is treated as a single unit of work. Consistency ensures that the database remains in a valid state before and after a transaction. Isolation provides concurrency control, ensuring that multiple transactions do not interfere with each other. Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent failures.
4. **Data Relationships**: RDBMS supports relationships between tables through the use of primary keys and foreign keys. These relationships allow data to be linked and provide the ability to retrieve related information from multiple tables through joins.
5. **Scalability and Performance**: RDBMS systems are designed to handle large amounts of data and can scale vertically (adding more resources to a single machine) or horizontally (distributing the data across multiple machines) to accommodate increasing workloads.

Some popular examples of RDBMSs include Oracle Database, MySQL, Microsoft SQL Server, and PostgreSQL. These systems have been widely adopted and used in various applications ranging from small-scale projects to large enterprise systems.

Important Terms Used In RDBMS:

In the context of RDBMS (Relational Database Management System), several important terms are commonly used to describe the structure and elements of a database. Here are the definitions of some key terms:

1. **Relation/Table**: A relation, also known as a table, is a two-dimensional structure that organizes data into rows and columns. Each table represents a specific entity or concept in the database. For example, a table "Employees" may store information about employees, with each row representing an individual employee and each column representing a specific attribute of an employee.
2. **Tuple/Row**: A tuple, also referred to as a row, represents a single record or instance of data within a table. It consists of a set of values, where each value corresponds to a specific attribute or column in the table. Using the "Employees" table example, a tuple would contain the data for a single employee, such as their name, age, and job title.

3. **Attribute/Column**: An attribute, also known as a column, represents a specific characteristic or property of the data stored in a table. Each column has a name and a data type that defines the kind of data it can hold. In the "Employees" table, attributes/columns could include "Name," "Age," "Job Title," and so on.
4. **Degree**: The degree of a relation refers to the number of attributes or columns in a table. It represents the width or the number of fields in a row. For instance, if a table has five attributes (columns), it is said to have a degree of 5.
5. **Cardinality**: Cardinality refers to the number of tuples or rows in a table. It represents the length or the number of records in a table. For example, if a table contains 100 tuples (rows), its cardinality is 100.
6. **Primary Key**: A primary key is a unique identifier for each tuple (row) in a table. It is a column or a combination of columns that uniquely identifies each record in the table. The primary key ensures the integrity and uniqueness of the data in the table, and it is used for referencing and linking tables in relational databases.
7. **Domain**: A domain refers to the set of all possible values that an attribute (column) can take. It defines the data type, constraints, and range of values that an attribute can hold. Common domains include integers, strings, dates, booleans, and more.
8. **Foreign key** A foreign key is a column or a set of columns in a table that establishes a link between the data in two related tables. It represents a relationship between two tables based on the values of the foreign key column(s) in one table matching the primary key column(s) in another table.

   In simpler terms, a foreign key in a table refers to the primary key of another table, creating a logical connection between the two tables. It enforces referential integrity, ensuring that the values in the foreign key column(s) of a table match the values in the primary key column(s) of the referenced table.

Understanding these terms is essential for effectively designing, querying, and working with relational databases. They provide the foundational concepts for organizing and manipulating data within an RDBMS.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

What Are Keys In RDBMS?

In RDBMS (Relational Database Management System), keys are important concepts used to identify and establish relationships between data in tables. Here are the definitions of different types of keys:

1. **Key**: A key is a column or a combination of columns that uniquely identifies each row (tuple) in a table. It helps establish relationships and ensures data integrity within a database. Keys can be categorized into different types based on their properties.

2. **Simple Key**: A simple key is a single column that uniquely identifies each tuple in a table. It consists of a single attribute. For example, in a table called "Students," a "StudentID" column that contains unique student identification numbers can be a simple key.

3. **Composite Key**: A composite key is a key that consists of multiple columns (attributes) combined to uniquely identify each tuple in a table. It requires the combination of values from two or more columns to create a unique identifier. Continuing with the "Students" table example, a composite key could be a combination of "StudentID" and "BatchID" columns, where both columns together uniquely identify each student's record within a specific batch.

4. **Super Key**: A super key is a set of one or more attributes (columns) that can uniquely identify tuples within a table. It may contain more attributes than required to uniquely identify a tuple. In other words, a super key is a superset of a candidate key. For instance, in a table of "Customers," a super key could be a combination of "CustomerID," "Email," and "Phone" columns, as it can uniquely identify each customer's record.

5. **Candidate Key**: A candidate key is a minimal set of attributes (columns) that can uniquely identify each tuple in a table. It means there are no unnecessary attributes in a candidate key. A table can have multiple candidate keys. For example, in a table of "Employees," a candidate key could be the "EmployeeID" column, as it uniquely identifies each employee's record.

Let's summarize with an example:

Consider a table called "Books" with the following columns:

- BookID (unique identifier for each book)
- Title (title of the book)
- Author (author of the book)
- ISBN (International Standard Book Number)

In this example:

- BookID can be a simple key, as it uniquely identifies each book.
- Title, Author, and ISBN together can form a composite key, as the combination of these attributes uniquely identifies each book.
- A super key could be a combination of BookID, Title, and Author, as it uniquely identifies each book but contains additional attributes.
- BookID and ISBN can be candidate keys, as they are minimal sets of attributes that uniquely identify each book.

It's important to note that the selection of keys depends on the specific requirements and constraints of the database design. Keys play a crucial role in maintaining data integrity, establishing relationships between tables, and ensuring efficient data retrieval and manipulation within an RDBMS.

Difference Between DBMS & RDBMS:

Feature	DBMS	RDBMS
Data Model	Various data models such as hierarchical, network, or flat	Relational data model
Data Structure	Structured and unstructured data	Structured data (organized into tables)
Data Relationships	Limited or no support for defining relationships	Supports relationships using keys (primary and foreign)
Data Integrity	Limited data integrity checks and constraints	Strong data integrity with constraints and keys
Query Language	May have its own proprietary query language	Standardized query language (SQL)
ACID Compliance	May or may not adhere to ACID properties	Adheres to ACID properties for data transactions
Scalability	Limited scalability options	Scalable architecture (vertical and horizontal scaling)
Joins	Limited or no support for complex joins	Supports complex joins between tables
Normalization	Limited or no support for normalization	Supports normalization techniques
Examples	File system, MongoDB, Redis	MySQL, Oracle Database, PostgreSQL

It's important to note that while RDBMS is a specific type of DBMS, it offers additional features and benefits specifically tailored to manage structured data using the relational model. RDBMS has gained significant popularity due to its ability to handle complex data relationships, enforce data integrity, and provide a standardized query language (SQL) for efficient data retrieval and manipulation

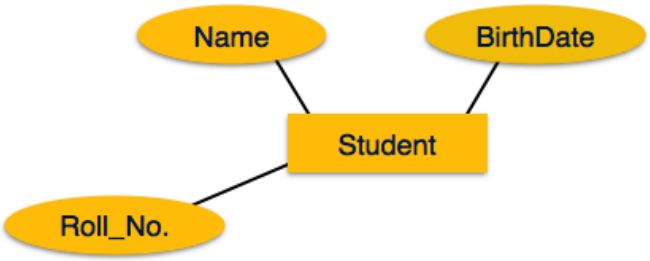
ER-Model (Entity Relationship Model):

ER-Modeling, also known as Entity-Relationship Modeling, is a technique used in DBMS (Database Management System) to design and represent the conceptual structure of a database. It focuses on identifying and defining the entities, relationships, and attributes within a system.

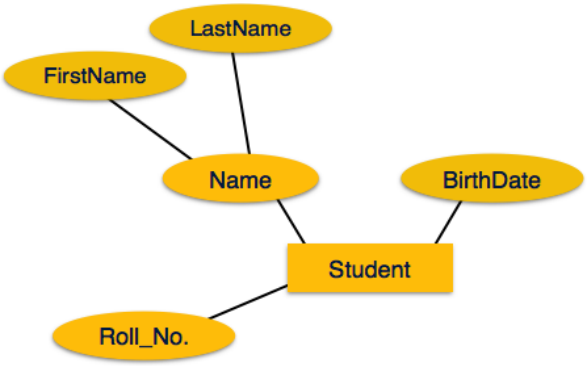
The ER-Modeling approach uses graphical notations to represent the various components of a database schema. The key elements in ER-Modeling are:

1. **Entity**: An entity represents a real-world object, concept, or thing with distinct characteristics. In the ER diagram, an entity is represented by a rectangle. For example, in a database for a university, entities could include "Student," "Teacher," or "Projects."

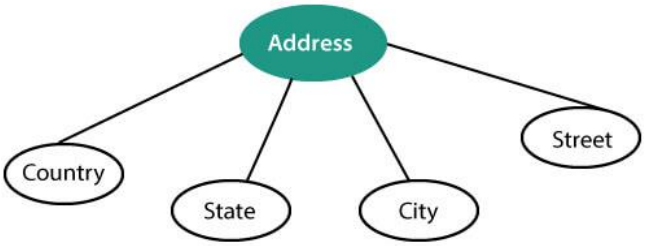
2. **Attributes**: Attributes are the properties or characteristics of an entity. They describe the specific details or qualities associated with an entity. Attributes are represented by ovals in the ER diagram. For example, attributes for the "Student" entity could include "Roll_no," "Name," "DOB," .



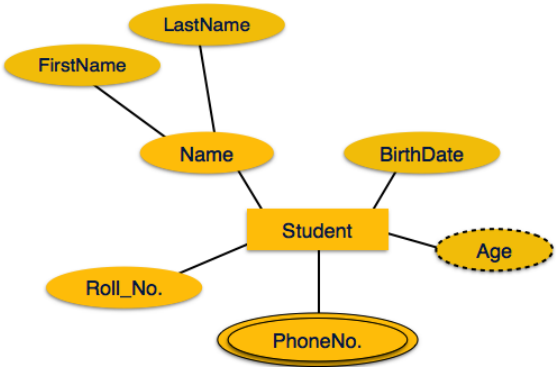
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, **composite attributes** are represented by ellipses that are connected with an ellipse / ovals.

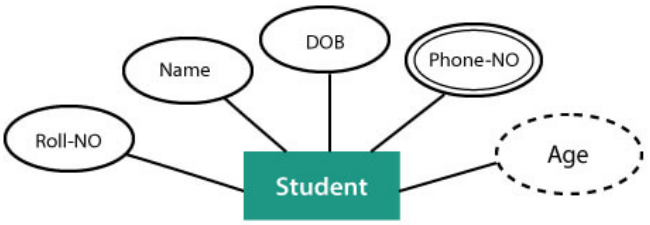


A **Composite attribute** is an attribute that can be further divided into sub-attributes, each representing a distinct component of the whole attribute. It allows for representing complex information as a combination of simpler attributes. **For example**, consider an address attribute that includes sub-attributes such as street, city, state, and Country.



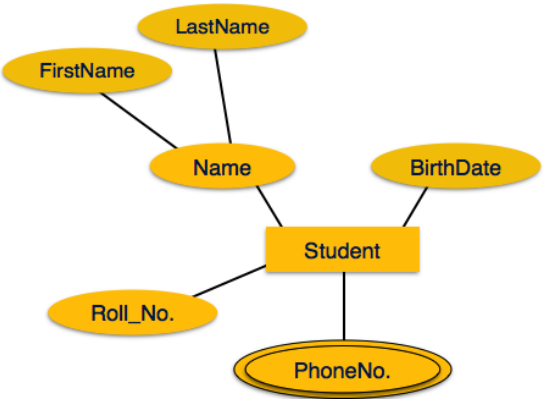
A **Derived attribute** is an attribute that can be derived or calculated from other attributes in the database. It is not directly stored in the database but can be computed based on the values of other attributes. Derived attributes provide derived information that is useful for analysis or reporting. Derived attributes are depicted by **Dashed ellipse**. For instance, in a database for Students records, the age of an employee can be derived from their birthdate.





A

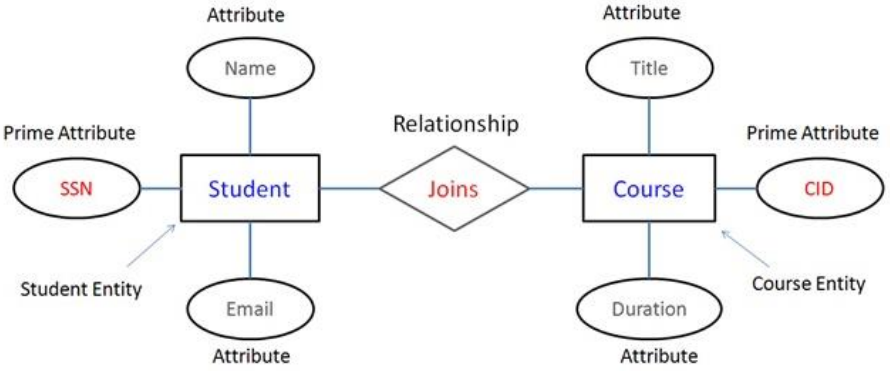
Multivalued attribute is an attribute that can have multiple values for a single entity or tuple. It represents a set of values rather than a single value. This type of attribute is useful when an entity can have multiple values for a specific characteristic. Multivalued attributes are depicted by **Double ellipse For example**, In a database for student records, we have an entity called "Student." One of the attributes for the student entity is "Phone Number." Typically, a student can have more than one phone number, such as a mobile number and a home number. In this case, the "Phone Number" attribute becomes a multivalued attribute because it can have multiple values for a single student.



Attribute Type	Description	Example
Composite Attribute	Attribute that can be divided into sub-attributes	Address (street, city, state, postal code)
Derived Attribute	Attribute calculated or derived from other attributes	Age (derived from birthdate)
Multivalued Attribute	Attribute that can have multiple values for an entity	Skills (programming, communication, leadership)

Understanding these attribute types is essential for accurately representing and modeling data within a DBMS, ensuring that the database schema effectively captures the required information and relationships.

3. **Relationships**: Relationships depict the associations or connections between entities. They describe how entities are related to each other. Relationships are represented by diamond-shaped symbols in the ER diagram. All the entities (rectangles) participating in a relationship, are connected to it by a line. **For example**, a "Student" entity may have a relationship with a "Course" entity representing the enrollment of students in courses.



Degree of Relationship:

The degree of a relationship in entity-relationship modeling refers to the number of entity sets participating in that relationship. It represents the number of entities involved in a relationship.

The degree of a relationship can be categorized into **three types**:

1. **Unary Relationship (Degree 1):**

- A unary relationship involves a single entity set. It is a self-referencing relationship where instances of the same entity set are related to each other.
- An example of a unary relationship is an "Employee" entity set where an employee can have a "Manager" attribute that refers to another employee in the same entity set.

2. **Binary Relationship (Degree 2):**
    - A binary relationship involves two distinct entity sets. It is the most common type of relationship in entity-relationship modeling.
    - Examples of binary relationships include the association between a "Student" entity and a "Course" entity or the relationship between an "Employee" entity and a "Department" entity.

3. **Ternary Relationship (Degree 3):**
    - A ternary relationship involves three distinct entity sets.
    - An example of a ternary relationship could be a relationship between a "Customer" entity, an "Order" entity, and a "Product" entity in an e-commerce database. The ternary relationship represents the fact that a customer places an order for a specific product.

The degree of relationship helps to determine the complexity and the number of entity sets involved in a relationship. It is an important aspect to consider when designing and understanding the structure of a database.

Participation constraints:

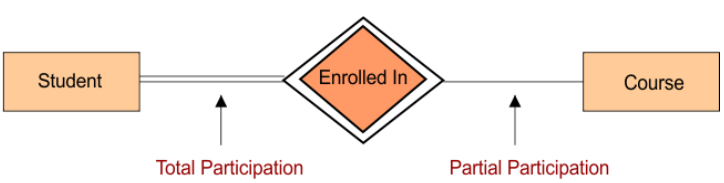
Participation constraints, also known as participation rules or constraints, are rules that specify the minimum participation of entities in a relationship. They define whether an entity's participation in a relationship is mandatory or optional. Participation constraints are applied to the ends of a relationship and help ensure data integrity and enforce business rules in a database.

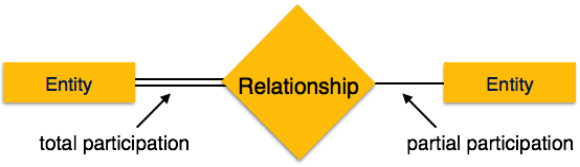
There are **two** types of participation constraints:

1. **Total Participation (Mandatory Participation):**
    - Total participation indicates that every instance of an entity set must participate in the relationship. It means that the entity's participation in the relationship is mandatory.
    - Denoted by a double line connecting the entity to the relationship line in an entity-relationship diagram (ER diagram).
    - For example, if there is total participation of students in a relationship with courses, it means that every student must be enrolled in at least one course.

2. **Partial Participation (Optional Participation):**
    - Partial participation indicates that an instance of an entity set may or may not participate in the relationship. It means that the entity's participation in the relationship is optional.
    - Denoted by a single line connecting the entity to the relationship line in an ER diagram.
    - For example, if there is partial participation of employees in a relationship with projects, it means that some employees may be assigned to projects while others may not be assigned to any project.

Participation constraints help ensure that the relationships between entities are accurately represented and align with the business rules and requirements of the database system. They play a crucial role in maintaining data integrity by enforcing constraints on the participation of entities in relationships.

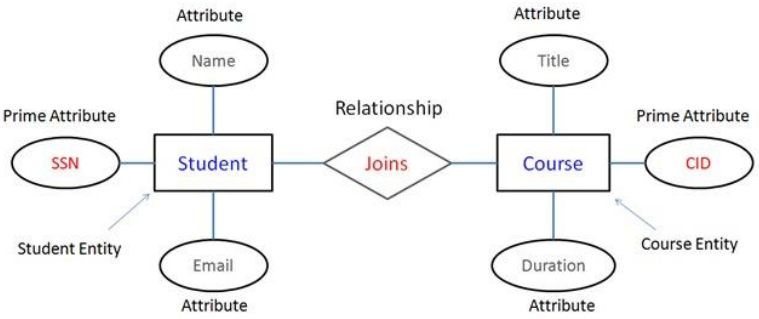




Here,

- According to Partial participation, there may exist some courses for which no enrollments are made.
- According to Total participation, each student must be enrolled in at least one course.

4. **Cardinality**: Cardinality specifies the number of instances of one entity that can be associated with the number of instances of another entity in a relationship. It defines the multiplicity or participation of entities in a relationship. Cardinality is represented using notations such as "one" (1), "many" (*), or specific numeric values. Instead of '*' can also use 'N' to represent "many".
   For example, a student can be enrolled in one or many courses, while a course can have many students.



One-to-One (1 - 1) Cardinality:

This means that one instance of an entity is associated with exactly one instance of another entity, & vice versa.
For example, consider a scenario where each student is assigned a unique student ID, and this ID is linked to one specific course. In this case, the cardinality is 1-1 because each student is enrolled in only one course, and each course has only one student assigned to it.

One-to-Many (1 - *) Cardinality:

This means that one instance of an entity is associated with multiple instances of another entity, but each instance of the other entity is associated with only one instance of the first entity.
For example, let's say that each student can enroll in multiple courses, but each course can have only one student. In this case, the cardinality is 1-* because one student can be enrolled in multiple courses, while each course is associated with only one student.

Many-to-One (* - 1) Cardinality:

This means that multiple instances of an entity are associated with a single instance of another entity.
For instance, consider a scenario where multiple students are enrolled in the same course, but each student is associated with only one course. Here, the cardinality is *-1 because multiple students can be enrolled in the same course (one course), but each student is associated with only one course.

Many-to-Many (* - *) Cardinality:

This means that multiple instances of an entity can be associated with multiple instances of another entity.
For example, suppose multiple students can enroll in multiple courses. In this case, the cardinality is * - * because each student can be enrolled in multiple courses, and each course can have multiple students.

To summarize:

1-1: Each student is associated with one course, and each course has only one student.
1-*: Each student can be enrolled in multiple courses, but each course is associated with only one student.
*-1: Multiple students are enrolled in the same course, but each student is associated with only one course.
* - *: Multiple students can enroll in multiple courses.

These cardinality notations help define the relationship and participation constraints between entities, allowing for accurate database modeling and understanding of how entities are related to each other.

ER-Modeling helps in visualizing and understanding the structure of a database system, its entities, attributes, and relationships. It serves as a blueprint for database design and can be transformed into a physical database schema using various normalization techniques.

The resulting ER diagram provides a clear representation of the database schema, which can be used for communication among stakeholders, database design, and implementation in DBMS systems.

Difference Between Relationship & Cardinality:

The terms "relationship" and "cardinality" are related but refer to different aspects in entity-relationship modeling:

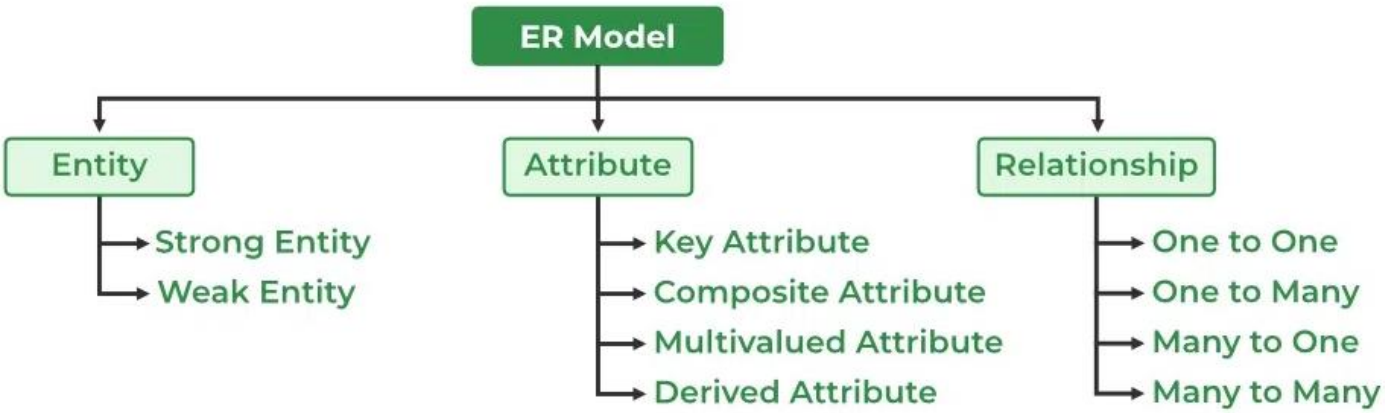
Relationship:

- A relationship represents the association or connection between entities in a database.
- It defines how two or more entities are related and interact with each other.
- Relationships are represented by diamond-shaped symbols in an entity-relationship diagram (ER diagram).
- Examples of relationships include the association between a student and a course, a customer and an order, or an employee and a department.

Cardinality:

- Cardinality describes the multiplicity or participation of entities in a relationship.
- It specifies the number of instances of one entity that can be associated with the number of instances of another entity in a relationship.
- Cardinality is represented using notations such as "one" (1), "many" (*), or specific numeric values.
- It helps determine the minimum and maximum number of instances that can participate in a relationship.
- Cardinality constraints are applied to the ends of a relationship to indicate the allowed number of occurrences.
- Examples of cardinality include one-to-one (1:1), one-to-many (1:N), and many-to-many (N:N) relationships.

In summary, a relationship defines the association between entities, while cardinality specifies the allowed number of instances participating in that relationship. The relationship determines how entities are connected, while cardinality provides the rules for how many instances of one entity can be associated with the instances of another entity. Both concepts are important in accurately representing the structure and behavior of a database in entity-relationship modeling.



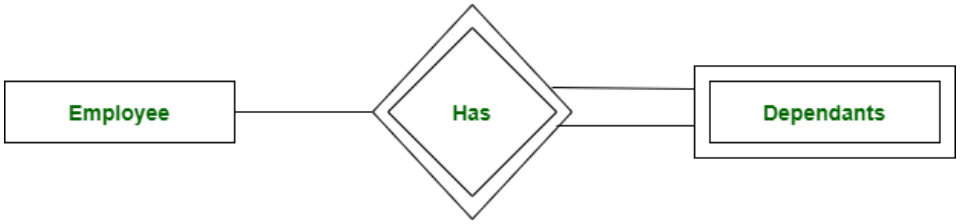
Entity Set	Strong Entity Set	
	Weak Entity Set	
Attributes	Simple Attribute	
	Composite Attribute	
	Single-valued Attribute	
	Multivalued Attribute	
	Derived Attribute	
	Null Attribute	
Relationship	Strong Relationship	
	Weak Relationship	

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

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

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

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

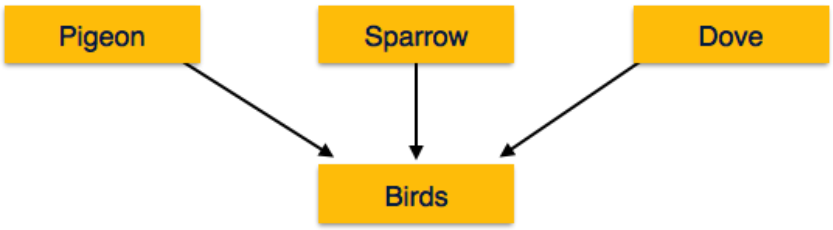


Generalization, Specialization, Inheritance & Aggregation:

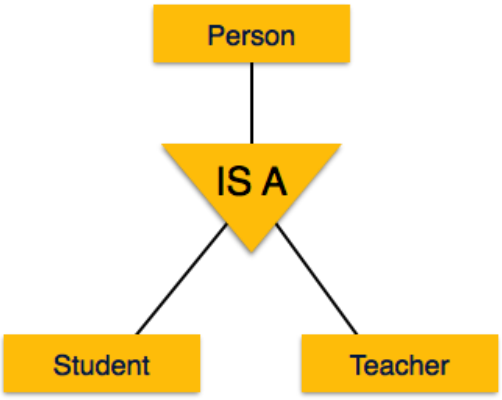
The ER Model has the power of expressing database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as we go deep in the hierarchy, it gives us the detail of every entity included.

Going up in this structure is called **generalization**, where entities are clubbed together to represent a more generalized view. For example, a particular student named Mira can be generalized along with all the students. The entity shall be a student, and further, the student is a person. The reverse is called **specialization** where a person is a student, and that student is Mira.

Generalization:
As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities, is called generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization:
Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group ‘Person’ for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.

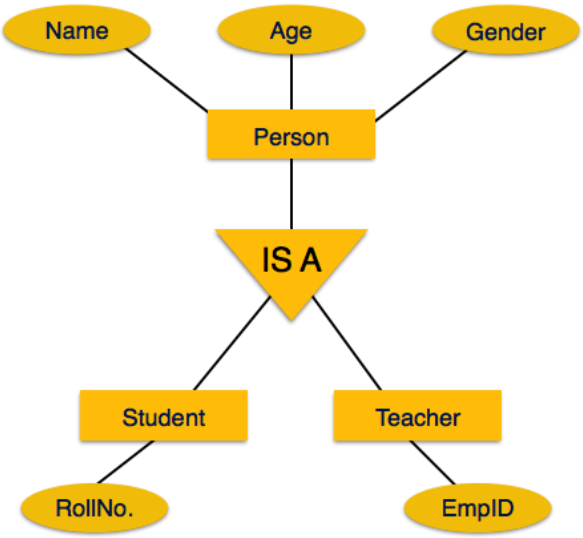


Similarly, in a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.

Inheritance:
We use all the above features of ER-Model in order to create classes of objects in object-oriented programming. The details of entities are generally hidden from the user; this process known as abstraction.

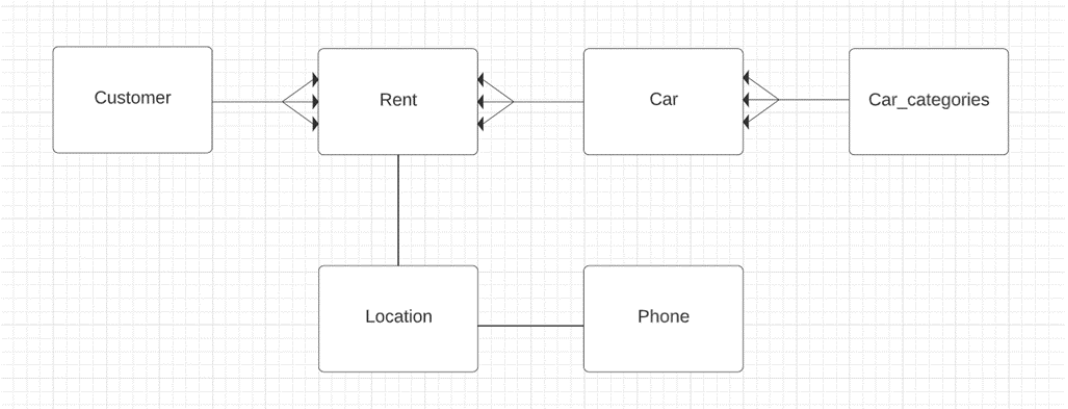
Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.

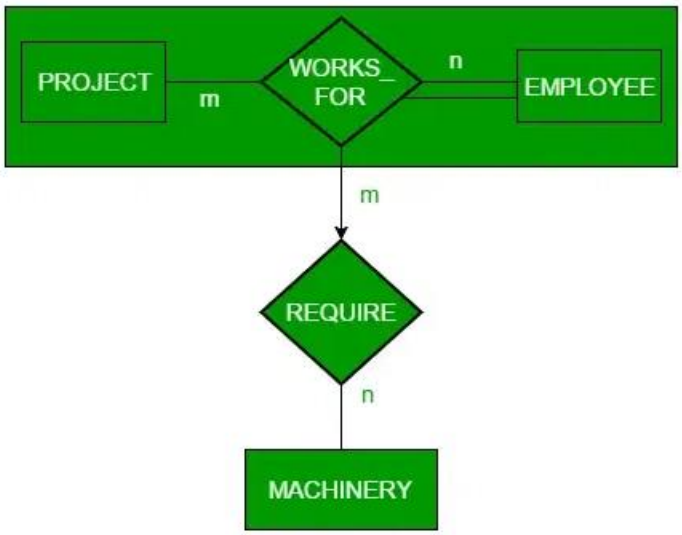
For example, the attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.



Aggregation:
Aggregation represents a relationship where one entity consists of or is composed of multiple other entities. It signifies a whole-part relationship between entities, where the whole entity has an aggregation relationship with its parts.

Example1: In a car rental system, there can be an entity called "Rental" that aggregates the entities "Car" and "Customer." The "Rental" entity represents the overall rental transaction and is composed of the "Car" entity (representing the rented car) and the "Customer" entity (representing the customer who rented the car). The "Rental" entity aggregates the "Car" and "Customer" entities to form a complete rental transaction.





Aggregation

Example2:
In the given example, let's consider the following entities and relationships:

- Entities: Employee, Project, Machinery
- Relationships: WORKS_FOR (between Employee and Project), REQUIRE (between the aggregated entity and Machinery)

The WORKS_FOR relationship represents the association between an Employee and a Project, indicating that an Employee works on a particular Project. However, in this scenario, we also have the requirement that an Employee working on a Project may require certain Machinery to perform their tasks.

To model this requirement using aggregation, we can create an aggregated entity that combines the Employee and Project entities. This aggregated entity represents the specific instance where an Employee is assigned to work on a Project. This aggregated entity can have its own attributes, such as StartDate or EndDate, to capture additional information about the assignment.
The aggregated entity (Employee-Project) will have a relationship called REQUIRE with the Machinery entity. This signifies that the aggregated entity, which represents the specific assignment of an Employee to a Project, requires certain Machinery to fulfill their job responsibilities.

In summary:

- The WORKS_FOR relationship captures the association between an Employee and a Project.
- By using aggregation, we create an aggregated entity (Employee-Project) that combines the Employee and Project entities.
- The REQUIRE relationship is established between the aggregated entity (Employee-Project) and the Machinery entity, indicating the requirement of Machinery for the specific Employee-Project assignment.

These concepts—**generalization, specialization, inheritance,** and **aggregation**—help in structuring & modeling complex relationships and hierarchies in database systems and object-oriented programming. They provide a way to organize and represent entities and their relationships in a meaningful and efficient manner.

'IS A' hierarchies, also known as inheritance hierarchies or subtype/supertype hierarchies, are a concept in database modeling that allows entities to inherit attributes and relationships from a higher-level entity. It is based on the principle of generalization and specialization.

In an **'IS A'** hierarchy, entities are organized into a hierarchy where the lower-level entities inherit characteristics from the higher-level entities. This relationship indicates that the lower-level entities are specific types or subtypes of the higher-level entity.

Here's an example to illustrate this concept:

Consider an entity called "Vehicle" as the higher-level entity. The 'IS A' hierarchy may include subtypes such as "Car," "Motorcycle," & "Truck." Each subtype inherits the attributes & relationships from the "Vehicle" entity but may also have its own unique attributes and relationships.

In this hierarchy:

- "Car," "Motorcycle," and "Truck" are the subtypes or lower-level entities.
- "Vehicle" is the supertype or higher-level entity.

Attributes and relationships defined in the "Vehicle" entity, such as "Manufacturer," "Model," & "Year," would be inherited by the subtypes. Each subtype may also have additional attributes or relationships specific to its type.

The 'IS A' hierarchy allows for specialization and differentiation of entities based on their unique characteristics while maintaining a common set of attributes and relationships shared by the higher-level entity.

Database modeling techniques, such as entity-relationship diagrams (ER diagrams) or Unified Modeling Language (UML) class diagrams, often use 'IS A' hierarchies to represent inheritance and specialization relationships in a structured manner.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Normalization:

Database **Normalization** is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Advantages of Normalization:

1. **Data Integrity**: Normalization helps maintain data integrity by reducing data redundancy & ensuring data consistency. By eliminating duplicate information and dependencies, normalization minimizes the chances of inconsistencies and conflicting data.
2. **Efficient Storage**: Normalization optimizes storage space by breaking down data into smaller, well-structured tables. This eliminates redundant data, reduces storage requirements, and improves the overall efficiency of the database.
3. **Improved Data Consistency**: Normalization reduces the likelihood of data anomalies such as insertion, deletion, and update anomalies. By organizing data logically and removing data dependencies, changes to the database can be made without risking data inconsistencies.
4. **Simplified Updates**: With normalized tables, updates and modifications to the database become simpler. Since data is organized in a granular manner, changes can be made to specific tables without affecting other related data. This improves database maintenance and makes it easier to update and manage data.
5. **Enhanced Query Performance**: Normalization facilitates efficient querying and improves overall database performance. Smaller, normalized tables with minimal redundancy allow for faster data retrieval, as there is less data to scan and process during queries.
6. **Scalability and Flexibility**: Normalization enables better scalability and flexibility in database design. As the database grows and new data entities or relationships are added, the normalized structure makes it easier to accommodate changes and extend the database schema without significant modifications to existing tables.
7. **Simplified Database Design**: Normalization provides guidelines and principles for designing a database schema. It helps database designers organize data logically and systematically, resulting in a more structured and well-defined database design.
8. **Improved Data Analysis**: Normalization supports effective data analysis and reporting. By reducing redundancy and ensuring data consistency, it becomes easier to perform data analysis, generate meaningful insights, and produce accurate reports.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Anomalies in database design:

In database design, anomalies refer to irregularities or issues that can occur when the database schema is not properly structured or normalized. These anomalies can affect data integrity, consistency, and the ability to perform operations effectively.

The common types of anomalies include:

- **Insertion Anomaly**:
  Insertion anomaly occurs when we cannot insert certain data into a database without adding additional, unrelated data. It happens when a table is not properly designed, leading to difficulties in adding new records.

  Example:

  Consider a table called "Employee" with attributes (EmployeeID, Name, Department). Suppose the table is designed in a way that requires a department to be specified for every new employee entry. If we want to insert a new employee record but do not yet know their department, we would encounter an insertion anomaly because we cannot add the employee without providing a department value.

- **Deletion Anomaly**:
  Deletion anomaly occurs when removing data from a database unintentionally removes other data that is still necessary or relevant. It happens when dependencies are not properly managed in the database design.

  Example:

  Continuing with the "Employee" table, let's assume that an employee can be associated with multiple projects, and we have a table called "Projects" with attributes (ProjectID, ProjectName, EmployeeID). If an employee works on multiple projects, and their projects are stored in the "Projects" table, deleting an employee from the "Employee" table would also delete their project information from the "Projects" table. This unintentional removal of project data is a deletion anomaly because the projects may still be relevant and associated with other employees.

- **Update Anomaly:**
  Update anomaly occurs when modifying data in a database leads to inconsistencies or redundancies due to the lack of proper normalization or data organization.

  Example:

  Consider a table called "Customer" with attributes (CustomerID, CustomerName, Address). Suppose a customer has multiple orders, and we store the orders in the "Orders" table with attributes (OrderID, CustomerID, OrderDate). If a customer changes their address, and we update their address in the "Customer" table, but forget to update the address in the corresponding orders in the "Orders" table, we introduce inconsistencies. This is an update anomaly because the address is duplicated and inconsistent across the tables.

Dependencies in the context of database design refer to the relationships between attributes or sets of attributes within a relation (table). They describe how changes in one attribute or set of attributes affect other attributes in the same relation. There are different types of dependencies, including functional dependencies, full Dependency, partial dependencies, and transitive dependencies.

Functional Dependency:

Functional dependency describes the relationship between two sets of attributes in a relation. It means that the value of one set of attributes determines the value of another set of attributes. In other words, if we know the values of certain attributes, we can determine the values of other related attributes.

Example:
Consider a relation "Employee" with attributes (EmployeeID, Name, Department). If we observe that each unique EmployeeID is associated with a unique Name, we can say there is a functional dependency: EmployeeID -> Name. It means that the Name attribute is functionally dependent on the EmployeeID attribute. Knowing the EmployeeID, we can determine the corresponding Name.

Full Dependency:

Full dependency occurs when an attribute depends on a whole composite key and not on any subset of that key. It means that all the attributes in the composite key are necessary to determine the value of the dependent attribute.

Example:
Suppose we have a relation "Course" with attributes (CourseID, CourseName, Department). If we observe that the Department attribute depends on the entire composite key (CourseID, CourseName) and not on any subset of it, we have a full dependency: CourseID, CourseName -> Department. It means that both CourseID and CourseName are necessary to determine the corresponding Department.

Partial Dependency:

Partial dependency occurs when an attribute depends on only a part of a composite key, rather than the entire composite key. It means that the value of the dependent attribute can be determined by only considering a subset of the composite key.

Example:
Continuing with the "Course" relation, let's assume that the Department attribute depends only on the CourseID attribute and not on CourseName. In this case, we have a partial dependency: CourseID -> Department. It means that the Department can be determined based on the CourseID alone, without considering the CourseName.

Transitive Dependency:

Transitive dependency occurs when an attribute depends on another attribute through a third attribute. In other words, the value of one attribute indirectly determines the value of another attribute.

Example:
Consider a relation "Student" with attributes (StudentID, CourseID, Professor). If we observe that the Professor attribute depends on the CourseID attribute, and the CourseID attribute depends on the StudentID attribute, we have a transitive dependency: StudentID -> CourseID -> Professor. In this case, the Professor attribute is transitively dependent on the StudentID attribute.

Understanding these dependencies is crucial in database design as they help in normalization, identifying redundancies, and ensuring data integrity. By identifying and managing these dependencies, we can create well-structured and efficient database schemas that maintain data consistency and minimize data anomalies.

Functional Dependency

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

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

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

- **Reflexive rule** – If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule** – If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule** – Same as transitive rule in algebra, if a → b holds and b → c holds, then a → c also holds. a → b is called as a functionally that determines b.

Trivial Functional Dependency

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

There are several normal forms (NF) in database normalization, each with its own set of rules. The most commonly used normal forms are:

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

- Ensures atomicity by eliminating repeating groups & making each attribute hold only atomic values.
- Each column in a table should contain only atomic values, and each row should be unique.
- Anomalies such as data redundancy and insertion, deletion, and update anomalies are addressed.

We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

Each attribute must contain only a single value from its pre-defined domain.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

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

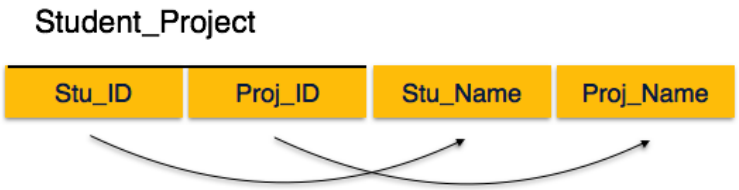
- Builds on 1NF and eliminates partial dependencies.
- Requires that all non-key attributes depend on the entire primary key, not just a part of it.
- Anomalies such as partial dependency are eliminated.

Before we learn about the second normal form, we need to understand the following −

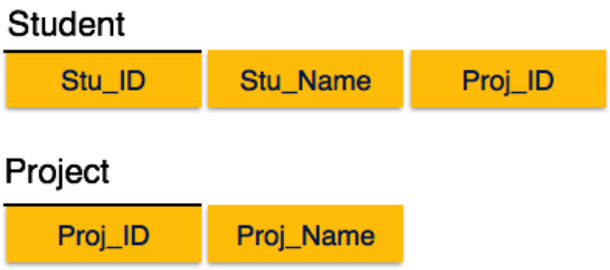
- **Prime attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.

- **Non-prime attribute** − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true.



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.



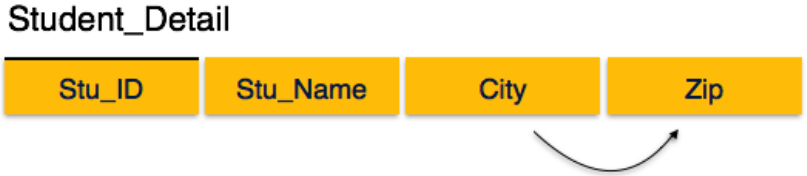
We broke the relation in two as depicted in the above picture. So there exists no partial dependency

3. Third Normal Form (3NF):

- Builds on 2NF and eliminates transitive dependencies.
- Requires that all non-key attributes depend directly on the primary key, not on other non-key attributes.
- Anomalies such as transitive dependency are eliminated.

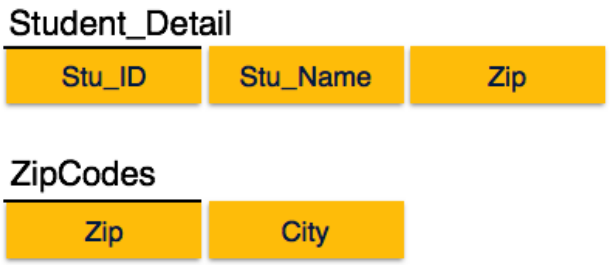
TutorialsPoint.com
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy −

- No non-prime attribute is transitively dependent on prime key attribute.
- For any non-trivial functional dependency, X → A, then either −
    - X is a super key or,
    - A is the prime attribute.



We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows −



4. Boyce-Codd Normal Form (BCNF):

BCNF is a higher level of normalization that addresses certain types of dependencies known as functional dependencies. BCNF states that for every non-trivial functional dependency (X -> Y) in a relation, X must be a superkey.

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that –

- For any non-trivial functional dependency, X → A, X must be a super-key.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu_ID → Stu_Name, Zip
and
Zip → City

Which confirms that both the relations are in BCNF.

Example:
Consider a table called "EmployeeSkills" with the following attributes: (EmployeeID, Skill, Department). Suppose we have the following functional dependencies:

EmployeeID -> Department (An employee is associated with only one department)

EmployeeID, Skill -> Department (An employee with a particular skill is associated with only one department)

In this case, the table violates BCNF because the functional dependency (EmployeeID, Skill) -> Department has a determinant (EmployeeID, Skill) that is not a superkey. To achieve BCNF, we can decompose the table into two separate tables: "Employees" (EmployeeID, Department) and "Skills" (EmployeeID, Skill).

5. Fifth Normal Form (5NF):

5NF, also known as Project-Join Normal Form (PJNF), deals with the elimination of join dependencies, which are dependencies that arise when combining multiple relations to retrieve information.

Example:

Consider a database for a library system with two tables: "Books" (BookID, Title, Author) and "Borrowers" (BorrowerID, Name, BookID, Title). The "BookID" and "Title" attributes are a composite key in the "Books" table. However, the "Title" attribute is also present in the "Borrowers" table.

To achieve 5NF, we can decompose the tables into three separate tables:

"Books" (BookID, Title, Author)

"Borrowers" (BorrowerID, Name)

"BookBorrowed" (BorrowerID, BookID)

In this decomposition, we eliminate the redundancy of the "Title" attribute by creating a separate "BookBorrowed" table that relates the "Borrowers" and "Books" tables using their respective IDs.

By achieving 5NF, the database structure ensures that join dependencies are eliminated, and information can be retrieved without unnecessary redundancy and inconsistencies.

Both **BCNF** and **5NF** are higher normal forms that aim to eliminate specific types of dependencies and improve the integrity and structure of a database. These normal forms help maintain data consistency, eliminate redundancy, and support efficient data retrieval and manipulation operations.

Denormalization:

Denormalization is the process of intentionally adding redundancy to a database schema by combining or duplicating data from separate tables. It is done to improve the performance of queries and simplify data retrieval in certain situations.

In a normalized database, the goal is to eliminate data redundancy and ensure data integrity by organizing data into separate tables based on functional dependencies. However, in some cases, the normalization process can lead to complex joins and increased query complexity, which may impact query performance.

Denormalization addresses these performance concerns by reintroducing redundancy and combining related data into fewer tables or even a single table. By doing so, denormalization aims to optimize query performance by reducing the number of joins required and providing faster data retrieval.

There are different forms of denormalization techniques:

1. Flattening or Combining Tables: Multiple related tables are combined into a single table to reduce the need for joins and simplify queries.
2. Duplicating Data: Redundant copies of data are stored in multiple tables to eliminate the need for complex joins and improve query performance.
3. Adding Derived Columns: Additional columns are added to a table that contain pre-calculated or derived data, eliminating the need for complex calculations during query execution.

Denormalization should be used judiciously and carefully because it introduces redundancy, which can lead to data inconsistency if not properly managed. It is often employed in scenarios where read performance is crucial and the cost of maintaining data consistency is outweighed by the performance gains.

It's important to note that denormalization is a trade-off between query performance and data consistency. It should be considered based on specific use cases, query patterns, and the nature of the application to ensure that the benefits of improved performance outweigh the drawbacks of increased redundancy.

SELECT Variants & Essential Points to Consider:

1. SELECT Statement:

The SELECT statement is a fundamental SQL command used to retrieve data from one or more database tables. It allows you to specify which columns and records you want to retrieve based on specific conditions and criteria.

The basic syntax of the SELECT statement is as follows:

SELECT column1, column2, ...FROM table WHERE condition;

Here, `column1, column2, ...` represents the columns you want to select, `table` represents the table from which you want to retrieve data, and `condition` represents the optional conditions for filtering the data.

2. SELECT *:

The asterisk (*) is a shorthand notation in the SELECT statement that represents "all columns." Using `SELECT *` retrieves all columns from the specified table(s).

Example:

*SELECT * FROM employees;*

This query retrieves all columns and all records from the "employees" table.

3. SELECT DISTINCT:

The DISTINCT keyword is used in conjunction with the SELECT statement to retrieve unique values from a specified column or a combination of columns.

Example:

SELECT DISTINCT department FROM employees;

This query retrieves unique department values from the "employees" table, eliminating duplicate entries.

4. SELECT INTO:

The SELECT INTO statement is used to create a new table and populate it with the result set obtained from a SELECT query. It allows you to create a table on-the-fly based on the columns and records retrieved.

Example:

SELECT column1, column2, ... INTO new_table FROM old_table;

This query creates a new table called "new_table" and populates it with the selected columns from "old_table."

5. SELECT TOP (or LIMIT):

The TOP keyword (in Microsoft SQL Server) or the LIMIT keyword (in most other database systems) is used to restrict the number of rows returned by a SELECT query.

Example:

```
SELECT TOP 5 *FROM employees;
```

This query retrieves the top 5 rows from the "employees" table.

6. SELECT with Aggregate Functions:

Aggregate functions, such as COUNT, SUM, AVG, MIN, and MAX, can be used in the SELECT statement to perform calculations on a set of rows and return a single result.

Example:

```
SELECT COUNT(*) AS total_employees FROM employees;
```

This query calculates the total number of employees in the "employees" table using the COUNT function.

Essential Points:
- The SELECT statement is used to retrieve data from a database table.
- You can specify the columns you want to select using column names or use "*" to select all columns.
- The WHERE clause allows you to apply conditions for filtering the data.
- The DISTINCT keyword retrieves unique values from a column or a combination of columns.
- SELECT INTO creates a new table based on the result set of a SELECT query.
- The TOP (or LIMIT) keyword limits the number of rows returned by a query.
- Aggregate functions can be used to perform calculations on a set of rows.

These SELECT variants and essential points provide a solid foundation for querying databases and retrieving the desired data based on specific requirements.

7. SELECT with Joins:

The SELECT statement can be combined with JOIN operations to retrieve data from multiple tables based on related columns. Joins allow you to combine rows from different tables into a single result set.

Example:

```
SELECT employees.employee_name, departments.department_name FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

This query retrieves the employee name and corresponding department name by joining the "employees" and "departments" tables based on the department ID.

8. SELECT with Sorting:

The ORDER BY clause is used in the SELECT statement to sort the result set based on one or more columns. It allows you to specify the sorting order as ascending (ASC) or descending (DESC).

Example:

```
SELECT employee_name, salary FROM employees ORDER BY salary DESC;
```

This query retrieves employee names and salaries from the "employees" table, sorted in descending order of salaries.

9. SELECT with Filtering:

The WHERE clause in the SELECT statement is used to filter rows based on specific conditions. It allows you to specify logical operators (such as =, >, <, LIKE, etc.) to narrow down the result set.

Example:

```
SELECT employee_name, hire_date FROM employees WHERE hire_date > '2022-01-01';
```

This query retrieves employee names and hire dates from the "employees" table, filtering only those who were hired after January 1, 2022.

10. SELECT with Aliases:

Aliases can be used in the SELECT statement to provide temporary names for columns or tables. Aliases are helpful for providing more meaningful or concise names in the result set.

Example:

```
SELECT employee_name AS Name, salary AS Salary FROM employees;
```

This query retrieves employee names and salaries from the "employees" table, but assigns the aliases "Name" and "Salary" to the respective columns in the result set.

11. SELECT with Conditional Logic:

The SELECT statement can utilize conditional logic using the CASE statement. It allows you to perform conditional evaluations and return different values based on specified conditions.

Example:

```
SELECT employee_name,
    CASE
        WHEN salary > 50000 THEN 'High'
        WHEN salary > 30000 THEN 'Medium'
        ELSE 'Low' END AS SalaryLevel FROM employees;
```

This query retrieves employee names and assigns a salary level ('High', 'Medium', or 'Low') based on their salary using the CASE statement.

Essential Points:

- JOIN operations combine rows from multiple tables in a SELECT statement.
- The ORDER BY clause is used to sort the result set based on one or more columns.
- The WHERE clause filters rows based on specified conditions.
- Aliases provide temporary names for columns or tables in the result set.
- The CASE statement allows for conditional evaluations and returns different values based on conditions.

These additional SELECT variants and essential points expand the capabilities of the SELECT statement and provide more flexibility in retrieving, filtering, sorting, and transforming data from database tables.

ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Operator	Description
SELECT	Retrieves data from one or more tables
FROM	Specifies the table(s) to retrieve data from
WHERE	Filters rows based on a condition
GROUP BY	Groups rows based on one or more columns
HAVING	Filters groups based on a condition
ORDER BY	Sorts the result set based on one or more columns
JOIN	Combines rows from two or more tables based on a related column
INNER JOIN	Returns only matching rows from both tables
LEFT JOIN	Returns all rows from the left table and matching rows from the right table
RIGHT JOIN	Returns all rows from the right table and matching rows from the left table
FULL JOIN	Returns all rows when there is a match in either the left or right table
UNION	Combines result sets of two or more SELECT statements
UNION ALL	Combines result sets of two or more SELECT statements (including duplicates)
INTERSECT	Returns the common rows between two SELECT statements
EXCEPT	Returns the rows from the first SELECT statement that are not in the second SELECT statement
IN	Checks if a value matches any value in a list or subquery
NOT IN	Checks if a value does not match any value in a list or subquery
EXISTS	Checks if a subquery returns any rows
NOT EXISTS	Checks if a subquery does not return any rows
BETWEEN	Checks if a value is within a range of values
LIKE	Searches for a specified pattern in a column value
NOT LIKE	Searches for values that do not match a specified pattern
IS NULL	Checks if a value is NULL (missing or unknown)
IS NOT NULL	Checks if a value is not NULL
= (Equal to)	Checks if two values are equal
<> or != (Not equal)	Checks if two values are not equal
> (Greater than)	Checks if one value is greater than another
< (Less than)	Checks if one value is less than another
>= (Greater than or equal to)	Checks if one value is greater than or equal to another
<= (Less than or equal to)	Checks if one value is less than or equal to another
AND	Logical operator for combining multiple conditions (all conditions must be true)
OR	Logical operator for combining multiple conditions (at least one condition must be true)
NOT	Logical operator for negating a condition
DISTINCT	Returns unique values in the result set
AS	Renames a column or table in the result set
COUNT	Returns the number of rows or non-null values in a column
SUM	Calculates the sum of values in a column

ANKKIT KUMAR GUPPTA

DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

AVG	Calculates the average value of a column
MAX	Returns the maximum value in a column
MIN	Returns the minimum value in a column
UPPER	Converts a string to uppercase
LOWER	Converts a string to lowercase
CONCAT	Concatenates two or more strings
SUBSTRING	Extracts a substring from a string
LENGTH	Returns the length of a string
ROUND	Rounds a numeric value to a specified number of decimal places
CASE	Performs conditional logic in a SELECT statement
NULLIF	Returns NULL if two expressions are equal, otherwise returns the first expression
COALESCE	Returns the first non-null value from a list of expressions
EXISTS	Checks if a subquery returns any rows
ANY/ALL	Compares a value with a list of values using specified conditions
CAST	Converts one data type to another
DATE functions	Various functions for manipulating dates and times
String functions	Various functions for manipulating strings
Numeric functions	Various functions for performing calculations on numeric values
Aggregate functions	Functions that operate on a set of values and return a single value
LIKE	Compares a value to a pattern using wildcard characters
IN	Checks if a value matches any value in a list
NOT IN	Checks if a value does not match any value in a list
EXISTS	Checks if a subquery returns any rows
NOT EXISTS	Checks if a subquery does not return any rows
BETWEEN	Checks if a value is within a range of values
NOT BETWEEN	Checks if a value is not within a range of values
CASE	Performs conditional logic in a query
UNION	Combines the results of two or more SELECT statements
INTERSECT	Returns the common rows between two SELECT statements
EXCEPT	Returns the distinct rows from the first SELECT statement that are not in the second SELECT statement
ALL	Compares a value to all values returned by a subquery
ANY/SOME	Compares a value to any value returned by a subquery
TOP/LIMIT	Limits the number of rows returned by a query
OFFSET/FETCH	Specifies the starting row and the number of rows to be returned from a query

CREATE TABLE employee (Emp_id INT PRIMARY KEY, First_name VARCHAR(50), Last_name VARCHAR(50), E_mail VARCHAR(100), Phone_number VARCHAR(20), Hire_date DATE, Job_id VARCHAR(20), Salary DECIMAL(10,2), Commission_pct DECIMAL(4,2), Manager_id INT, Department_id INT);

```
INSERT ALL
  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
  VALUES (101, 'John', 'Doe', 'john.doe@example.com', '1234567890', DATE '2022-01-01', 'J001', 5000, 0.1, 1001, 10)

  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
  VALUES (102, 'Jane', 'Smith', 'jane.smith@example.com', '9876543210', DATE '2022-02-01', 'J002', 6000, 0.2, 1002, 20)

  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
  VALUES (150, 'David', 'Lee', 'david.lee@example.com', '4567890123', DATE '2022-05-01', 'J005', 7000, 0.15, 1005, 50)

INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
  VALUES (104, 'Ankit', 'Gupta', 'ankitgupta6009@gmail.com', '9673170306', DATE '2022-01-13', 'J004', 60000, 0.5, 1005, 15)

  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
  VALUES (105, 'Nandini', 'Gupta', 'ng160593@gmail.com', '9044719651', DATE '2022-02-04', 'J005', 50000, 0.5, 1013, 16)

INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
    VALUES (106, 'PRIYA', 'GUPTA', 'PRIYA@example.com', '1234567890', TO_DATE('2022-01-01', 'YYYY-MM-DD'), 'J006', 5000.00, 0.10, 1001, 10)

  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary, Commission_pct, Manager_id, Department_id)
    VALUES (107, 'KAUSHAL', 'GUPTA', 'KAUSHAL@example.com', '9876543210', TO_DATE('2022-02-01', 'YYYY-MM-DD'), 'J007', 6000.00, 0.20, 1002, 20)
```

```
  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary,
Commission_pct, Manager_id, Department_id)
    VALUES (108, 'ARUN', 'MODANWAL', 'ARUN@example.com', '5551234567', TO_DATE('2022-03-01',
'YYYY-MM-DD'), 'J008', 7000.00, 0.15, 1001, 15)

  INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary,
Commission_pct, Manager_id, Department_id)
    VALUES (109, 'LAXMI', 'MODANWAL', 'LAXMI@example.com', '9998887777', TO_DATE('2022-04-
01', 'YYYY-MM-DD'), 'J009', 5500.00, 0.12, 1002, 16)

INTO employee (Emp_id, First_name, Last_name, E_mail, Phone_number, Hire_date, Job_id, Salary,
Commission_pct, Manager_id, Department_id)
VALUES (110, 'ANJU', 'GUPTA', 'ANJU@example.com', '4445556666', TO_DATE('2022-05-01', 'YYYY-
MM-DD'), 'J0010', 8000.00, 0.18, 1003, 30)
SELECT 1 FROM DUAL;
```

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	101	John	Doe	john.doe@example.com	1234567890	01-JAN-22	J001	5000	0.1	1001	10
2	102	Jane	Smith	jane.smith@example.com	9876543210	01-FEB-22	J002	6000	0.2	1002	20
3	150	David	Lee	david.lee@example.com	4567890123	01-MAY-22	J005	7000	0.15	1005	50
4	104	Ankit	Gupta	ankitgupta6009@gmail.com	9673170306	13-JAN-22	J004	60000	0.5	1005	15
5	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16
6	106	PRIYA	GUPTA	PRIYA@example.com	1234567890	01-JAN-22	J006	5000	0.1	1001	10
7	107	KAUSHAL	GUPTA	KAUSHAL@example.com	9876543210	01-FEB-22	J007	6000	0.2	1002	20
8	108	ARUN	MODANWAL	ARUN@example.com	5551234567	01-MAR-22	J008	7000	0.15	1001	15
9	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16
10	110	ANJU	GUPTA	ANJU@example.com	4445556666	01-MAY-22	J0010	8000	0.18	1003	30

```
CREATE TABLE department (
  department_id INT PRIMARY KEY,
  department_name VARCHAR(50),
  location_ID INT
);
 INSERT ALL
  INTO department (department_id, department_name, location_ID) VALUES (10, 'Finance',1)
  INTO department (department_id, department_name, location_ID) VALUES (20, 'Sales', 2)
  INTO department (department_id, department_name, location_ID) VALUES (30, 'Marketing', 3)
  INTO department (department_id, department_name, location_ID) VALUES (15, 'Human Resources', 4)
  INTO department (department_id, department_name, location_ID) VALUES (50, 'IT', 5)
  INTO department (department_id, department_name, location_ID) VALUES (16, 'DATA', 4)
SELECT * FROM dual;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Finance	1
2	20	Sales	2
3	30	Marketing	3
4	15	Human Resources	4
5	50	IT	5
6	16	DATA	4

```
CREATE TABLE location (
  location_id INT PRIMARY KEY,
  city VARCHAR(50),
  state VARCHAR(50),
  country VARCHAR(50));

INSERT ALL
  INTO location (location_id, city, state, country) VALUES (1, 'New York City', 'New York', 'United States')
  INTO location (location_id, city, state, country) VALUES (2, 'Los Angeles', 'California', 'United States')
  INTO location (location_id, city, state, country) VALUES (3, 'Chicago', 'Illinois', 'United States')
  INTO location (location_id, city, state, country) VALUES (4, 'Houston', 'Texas', 'United States')
  INTO location (location_id, city, state, country) VALUES (5, 'San Francisco', 'California', 'United States')
SELECT * FROM dual;
```

	LOCATION_ID	CITY	STATE	COUNTRY
1	1	New York City	New York	United States
2	2	Los Angeles	California	United States
3	3	Chicago	Illinois	United States
4	4	Houston	Texas	United States
5	5	San Francisco	California	United States

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Select Variants Queries:

- select * from employee;
- select emp_id, first_name from employee;
- select emp_id, salary, salary+100 from employee;
- select emp_id, salary, 12*salary+100 from employee;
- select emp_id, salary, 12*(salary+100) from employee;
- select emp_id, salary, salary+(salary*commission_pct) from employee;
- select distinct commission_pct from employee;
- select first_name||last_name from employee;
- select first_name|| ' ' ||last_name from employee;
- select first_name|| q'['s]'|| ' ' || last_name from employee;
- select first_name from employee where emp_id=104;
- select first_name from employee where emp_id<>104;
- select emp_id, first_name from employee where salary >22000;
- select first_name from employee where emp_id<104;
- select emp_id, first_name from employee where salary >=22000;
- select first_name from employee where emp_id<=104;
- select emp_id, first_name, salary from employee where emp_id in (102,104,105);
- select emp_id, first_name from employee where first_name like 'a%';
- select emp_id, first_name from employee where first_name like '__nd%';
- select emp_id, first_name from employee where first_name like '_a%';
- select emp_id, first_name from employee where commission_pct is null;
- select emp_id, first_name from employee where commission_pct is not null;
- select emp_id, first_name from employee where department_id not in (10,20,50);
- select emp_id, first_name from employee where first_name like 'a%' and department_id=15;
- select emp_id, first_name, salary from employee where salary between 6000 and 80000;
- select emp_id, first_name from employee where first_name like '__nd%' or department_id=16;

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

SORTING:

Sorting in SQL refers to the process of arranging the result set of a query in a specific order based on one or more columns. It allows you to organize the data in a meaningful way for analysis or presentation purposes. The ORDER BY clause is used to specify the columns by which the result set should be sorted.

Values that can be sorted include various data types such as:

1. **Numeric Values**: Integer, decimal, or floating-point numbers can be sorted in ascending (smallest to largest) or descending (largest to smallest) order.

2. **Character Strings**: Alphanumeric strings, words, or phrases can be sorted alphabetically in ascending or descending order. The sorting is typically based on the ASCII or Unicode values of the characters.

3. **Dates and Times**: Dates and times can be sorted chronologically in ascending or descending order. This allows you to arrange events or data points based on their occurrence or time values.

4. **Boolean Values**: Boolean values (true or false) can be sorted based on their logical order, where false comes before true or vice versa.

When using the ORDER BY clause, you can specify the column(s) by which the result set should be sorted. Multiple columns can be used for sorting, and the sorting order can be specified for each column individually.

For example, to sort a result set of employees by their last name in ascending order and then by their first name in descending order, you can use the following SQL query:

```
SELECT * FROM employees ORDER BY last_name ASC, first_name DESC;
```

This query will return the employees' data sorted by last name in ascending order. If there are employees with the same last name, they will be further sorted by their first name in descending order.

QUERIES: Sorting (Display the result in orderly way) ORDER BY

- Select emp_id, department_id, job_id from employee order by emp_id ;
  --ascending order is by default
- Select emp_id, department_id, job_id from employee order by emp_id desc;
- Select emp_id, department_id, job_id as job from employee order by job desc;
- Select emp_id, department_id, job_id from employee order by 2;
  --here 2 is column position which department_id.
- Select emp_id, department_id, job_id from employee order by emp_id, job_id desc;

Sorting by EMP_ID (ascending order): The result set will be arranged in ascending order based on the values in the EMP_ID column. This means the rows with the smallest EMP_ID values will appear first, followed by rows with progressively larger EMP_ID values.

Sorting by JOB_ID (descending order): If there are rows with the same EMP_ID value, the JOB_ID column will be used to further sort the result set.
In this case, the sorting will be in descending order, meaning the rows with the largest JOB_ID values will appear first within each group of rows with the same EMP_ID.

ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Substitution variables: (&)

Substitution variables are placeholders in SQL that allow you to dynamically replace them with specific values during the execution of a query or script. They are often denoted by a prefix symbol, such as the ampersand (&) in Oracle SQL.

Substitution variables are typically used in interactive SQL sessions or scripts where you want to prompt the user for input or provide a way to parameterize your queries. They offer flexibility and allow you to reuse the same query with different values without modifying the query itself.

Here's an **example** of how substitution variables can be used in SQL:

SELECT * FROM employees WHERE department_id = &dept_id;

In this example, the substitution variable `&dept_id` is used to prompt the user for a department ID value. When the query is executed, the user will be prompted to enter a specific department ID. The value entered by the user will replace the substitution variable in the query, & the query will be executed using the provided value.

Substitution variables can be used in various SQL statements, including SELECT, INSERT, UPDATE, and DELETE. They are particularly useful when you want to make your queries more dynamic and parameterized, allowing you to easily customize the behavior of your queries based on user input or other variables.

It's important to note that the usage of substitution variables may vary depending on the database management system you are using. Different database systems may have different syntax or methods for implementing substitution variables.

You can use Substitutional variables to supplement in

- WHERE clause
- ORDER BY clause
- COLUMN name
- TABLE name
- Entire select statements
- Define statement
- Verify command

SYNTAX:

```
Select &column_name  from &table_name where &condition order by &order_column
```

ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

SUBSTITUTION VARIALBES QUERIES:

- Select emp_id,first_name, manager_id from employee where department_id= &department_id;
- Select &emp_id,first_name, manager_id from employee where department_id= 16;
- Select emp_id,first_name, manager_id from &table_name where department_id = 16;
- Select emp_id,first_name, manager_id from employee where department_id= 16 order by &order_column desc;

----------`. ----------

DEFINE DEPARTMENT_ID=16

Select emp_id,first_name, manager_id from employee Where department_id = &department_id;

UNDEFINE DEPARTMENT_ID

- **`DEFINE DEPARTMENT_ID=16`**: This line defines a substitution variable `DEPARTMENT_ID` and assigns it the value `16Substitution variables are a way to dynamically input values into a SQL query.
- **`SELECT EMP_ID, FIRST_NAME, MANAGER_ID`**: This is the `SELECT` statement that specifies the columns to retrieve from the `employee` table. It selects the `EMP_ID`, `FIRST_NAME`, and `MANAGER_ID` columns.
- **`FROM EMPLOYEE`:** This clause specifies the table from which to retrieve the data, which is the `employee` table in this case.
- **`WHERE DEPARTMENT_ID = &DEPARTMENT_ID`**: This is the `WHERE` clause, which filters the results based on a condition. It compares the `DEPARTMENT_ID` column in the `employee` table with the value of the substitution variable `&DEPARTMENT_ID`. The `&DEPARTMENT_ID` represents the value you previously defined using the `DEFINE` statement.
- **`UNDEFINE DEPARTMENT_ID`**: This line undefines the substitution variable `DEPARTMENT_ID` after the query execution. It clears the defined value.

In summary, this query retrieves the `EMP_ID`, `FIRST_NAME`, and `MANAGER_ID` of employees from the `employee` table whose `DEPARTMENT_ID` matches the value specified in the substitution variable.

SET VERIFY ON

Select emp_id, first_name, salary from employee Where emp_id =&employee_number;

- `SET VERIFY ON`: This line enables the verification of substitution variables in the SQL*Plus environment. When `VERIFY` is set to `ON`, SQL*Plus displays the substituted values in the output.

- `SELECT EMP_ID, FIRST_NAME, SALARY`: This is the `SELECT` statement that specifies the columns to retrieve from the `employee` table. It selects the `EMP_ID`, `FIRST_NAME`, and `SALARY` columns.

- `FROM EMPLOYEE`: This clause specifies the table from which to retrieve the data, which is the `employee` table in this case.

- `WHERE EMP_ID =&EMPLOYEE_NUMBER`: This is the `WHERE` clause, which filters the results based on a condition. It compares the `EMP_ID` column in the `employee` table with the value of the substitution variable `&EMPLOYEE_NUMBER`. The `&EMPLOYEE_NUMBER` represents the value that will be prompted to be entered by the user at runtime.

  Note: In Oracle SQL, substitution variables are denoted by the ampersand (&) symbol followed by the variable name. In this case, `&EMPLOYEE_NUMBER` is the substitution variable that prompts the user to enter a value for the `EMPLOYEE_NUMBER` during query execution.

  Overall, this query retrieves the `EMP_ID`, `FIRST_NAME`, and `SALARY` of an employee from the `employee` table whose `EMP_ID` matches the value entered for the substitution variable `&EMPLOYEE_NUMBER`.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Single-Row Functions & Conditional Expressions:

In SQL, **single-row functions** are functions that operate on individual rows of a query result set and return a single value for each row. These functions can be used to perform calculations, manipulate strings or dates, convert data types, and more. Single-row functions are typically used within the SELECT statement to modify or transform the data being retrieved.

There are several categories of single-row functions in SQL:

1. Numeric Functions: These functions perform mathematical operations on numeric data types.

Examples include:
- ABS: Returns the absolute value of a number.
- ROUND: Rounds a number to a specified number of decimal places.
- TRUNC: Truncates a number to a specified number of decimal places.
- SQRT: Calculates thCXe square root of a number.

2. String Functions: These functions manipulate and operate on string data.

Examples include:
- CONCAT: Concatenates two or more strings together.
- LENGTH: Returns the length (number of characters) of a string.
- SUBSTR: Extracts a substring from a string based on starting position and length.
- UPPER/LOWER: Converts a string to uppercase or lowercase, respectively.

3. Date Functions: These functions operate on date and time values.

Examples include:
- SYSDATE: Returns the current date and time.
- EXTRACT: Extracts a specific part (year, month, day, etc.) from a date.
- TO_CHAR: Converts a date to a specific format as a string.
- ADD_MONTHS: Adds a specified number of months to a date.

4. Conversion Functions: These functions convert data from one type to another.

Examples include:
- TO_NUMBER: Converts a string to a numeric value.
- TO_CHAR: Converts a value to a character string.
- TO_DATE: Converts a string to a date value.

These are just a few examples of the commonly used single-row functions in SQL. Each database system may have its own set of functions and syntax, so it's important to refer to the specific documentation of the database you are using for a comprehensive list of available functions and their usage.

single-row functions can be used in conjunction with the SELECT, WHERE, and ORDER BY clauses in SQL queries.

1. SELECT Clause: Single-row functions are often used within the SELECT clause to perform calculations or transformations on specific columns or expressions. They allow you to modify the retrieved data before it is presented in the result set. For example:

```
SELECT column1, column2, UPPER(column3) AS modified_column FROM table_name;
```

In this example, the UPPER() function is used to convert the values in column3 to uppercase for each row in the result set.

2. WHERE Clause: Single-row functions can also be used within the WHERE clause to filter rows based on specific conditions. You can use these functions to manipulate the column values being compared or apply certain criteria to the data. For example:

```
SELECT * FROM table_name WHERE LENGTH(column1) > 5;
```

In this example, the LENGTH() function is used to filter the rows where the length of column1 is greater than 5.

3. ORDER BY Clause: Single-row functions can be used within the ORDER BY clause to sort the result set based on modified values. You can apply these functions to the columns being sorted to change the sorting behavior. For example:

```
SELECT * FROM table_name ORDER BY ROUND(column1, 2) DESC;
```

In this example, the ROUND() function is used to round the values in column1 to two decimal places, and the result set is sorted in descending order based on the rounded values.

By using single-row functions in conjunction with these clauses, you can perform various calculations, manipulations, and transformations on the data during the query execution.

General Functions:

- **Null Handling Functions:**

1. NVL:

Syntax: NVL(expression1, expression2)

Description: NVL function returns the value of expression1 if it is not null; otherwise, it returns expression2.
Example: SELECT NVL(salary, 0) FROM employees;

This query returns the salary of employees. If the salary is null, it will be replaced with 0 in the result set.

2. NVL2:

Syntax: NVL2(expression1, expression2, expression3)

Description: NVL2 function returns expression2 if expression1 is not null; otherwise, it returns expression3.

Example: SELECT NVL2(commission_pct, 'Eligible', 'Not Eligible') FROM employees;

This query checks if an employee is eligible for commission. If the commission_pct is not null, it returns 'Eligible'; otherwise, it returns 'Not Eligible'.

3. NULLIF:

Syntax: NULLIF(expression1, expression2)

Description: NULLIF function returns null if expression1 is equal to expression2; otherwise, it returns expression1.

Example: SELECT NULLIF(salary, 0) FROM employees;

This query compares the salary of employees with 0. If the salary is equal to 0, it returns null; otherwise, it returns the salary value.

4. COALESCE:

Syntax: COALESCE(expression1, expression2, ...)

Description: COALESCE function returns the first non-null expression from the given list of expressions.

Example: SELECT COALESCE(salary, commission_pct, 0) FROM employee;

This query returns the first non-null value from the salary, commission_pct, and 0. If all values are null, it returns 0.

Conditional Expressions:

5. CASE:
The CASE expression allows you to evaluate a list of conditions and return a specific result based on the first matching condition. There are two forms of the CASE expression: simple CASE and searched CASE.

1. **Simple CASE:**

Syntax:
```
CASE expression
   WHEN value1 THEN result1
   WHEN value2 THEN result2
   ELSE result
END
```

Example:
```
CASE department_id
   WHEN 10 THEN 'Finance'
   WHEN 20 THEN 'HR'
   ELSE 'Other'
   END
```

2. **Searched Case:**

Syntax:
```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE result
  END
```

Example:
```
SELECT
   CASE
     WHEN salary > 5000 THEN 'High'
     WHEN salary > 3000 THEN 'Medium'
     ELSE 'Low'
   END AS salary_category
 FROM employee;
```

This query categorizes employees' salaries as 'High', 'Medium', or 'Low' based on the salary value.

6. DECODE:

The DECODE function in Oracle SQL is used to perform conditional comparisons and return a result based on the first matching condition. It provides a way to achieve similar functionality as the `IF-THEN-ELSE` construct or `CASE` expression in other programming languages.

Syntax:
DECODE(expression, value1, result1, value2, result2, ..., default_result)

The DECODE function compares the expression with the given values in pairs. If the expression matches a value, it returns the corresponding result. If no match is found, it returns the default_result (optional).

Here's an example to illustrate the usage of the DECODE function:

Example: SELECT DECODE(job_id, 'J001', 'Manager', 'J002', 'Assistant', 'Other') FROM employee;

This query checks the job_id of employees. If the job_id is 'J001', it returns 'Manager'; if it is 'J002', it returns 'Assistant'; otherwise, it returns 'Other'.

The DECODE function allows you to handle multiple conditions within a single expression, making the code more concise and readable.

Note that the DECODE function is specific to Oracle SQL. Other database systems may have similar conditional functions with different names or syntax, such as `CASE` in standard SQL or `IFNULL` in MySQL.

These general functions provide flexibility in manipulating and transforming data within SQL queries based on specific conditions or requirements.

Nested functions

Nested functions, also known as function composition, refer to the practice of using the result of one function as an argument for another function within the same expression. This allows for complex and intricate calculations or transformations by combining multiple functions together. Here's an example to illustrate nested functions:

Let's say we have a table called `employees` with columns `first_name`, `last_name`, and `salary`. We want to retrieve the full names of employees in uppercase letters. However, if an employee's salary is above a certain threshold, we want to append the word "High earner" to their name.

Here's how we can achieve this using nested functions:

```
SELECT
UPPER(CONCAT(first_name, ' ', last_name)) AS full_name,
CASE
WHEN salary > 5000 THEN CONCAT(UPPER(CONCAT(first_name, ' ', last_name)), ' - High earner')
ELSE UPPER(CONCAT(first_name, ' ', last_name))
END AS modified_name FROM employees;
```

```
SELECT
  UPPER(first_name || ' ' || last_name) AS full_name,
  CASE
    WHEN salary > 5000 THEN UPPER(first_name || ' ' || last_name || ' - High earner')
    ELSE UPPER(first_name || ' ' || last_name)
  END AS modified_name FROM employees;
```

This code is supported by oracle instead of above code

In the above example, we use nested functions to accomplish the desired result.

- First, we use the `CONCAT` function to concatenate the `first_name` and `last_name` columns, resulting in the full name of the employee.

- Then, we apply the `UPPER` function to convert the full name to uppercase.

- Next, we use the `CASE` statement to check if the employee's salary is greater than 5000.

- If the salary condition is true, we use nested functions to concatenate the modified name with the "High earner" suffix, using another `CONCAT` and `UPPER` function.

- If the salary condition is false, we simply use the modified name without any additional suffix.

By utilizing nested functions, we can perform multiple operations or conditions within a single SQL statement, enabling more complex transformations and calculations on the data.

ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

GROUPING & FILERING:

In Oracle SQL, group functions, also known as aggregate functions, are used to perform calculations on sets of rows and return a single result for each group. These functions operate on a group of rows rather than individual rows. Group functions are typically used in combination with the **GROUP BY** clause to specify the groups on which the calculations should be performed.

Here are some commonly used group functions in Oracle SQL:

1. COUNT: Returns the number of rows in a group.
Syntax: `COUNT(expression)` or `COUNT(*)`
Example: `SELECT COUNT(*) FROM employees`

2. SUM: Calculates the sum of a numeric column in a group.
Syntax: `SUM(expression)`
Example: `SELECT SUM(salary) FROM employees`

3. AVG: Calculates the average value of a numeric column in a group.
Syntax: `AVG(expression)`
Example: `SELECT AVG(salary) FROM employees`

4. MIN: Retrieves the minimum value of a column in a group.
Syntax: `MIN(expression)`
Example: `SELECT MIN(salary) FROM employees`

5. MAX: Retrieves the maximum value of a column in a group.
Syntax: `MAX(expression)`
Example: `SELECT MAX(salary) FROM employees`

6. GROUP_CONCAT: Concatenates the values of a column within a group into a single string.
Syntax: `LISTAGG(expression, separator)`
Example: `SELECT LISTAGG(last_name, ', ') WITHIN GROUP (ORDER BY last_name) FROM employees`

7. STDDEV or STDDEV_POP: Calculates the standard deviation of a numeric column in a group.
Syntax: `STDDEV(expression)` or `STDDEV_POP(expression)`
Example: `SELECT STDDEV(salary) FROM employees`

8. VARIANCE or VAR_POP: Calculates the variance of a numeric column in a group.
Syntax: `VARIANCE(expression)` or `VAR_POP(expression)`
Example: `SELECT VARIANCE(salary) FROM employees`

These group functions are often used in conjunction with the GROUP BY clause to perform calculations on subsets of data and obtain aggregated results. They provide powerful tools for summarizing and analyzing data in Oracle SQL queries.

Key Points:

- All group functions ignore NULL values, but NVL functions forces to include NULL values
- Divide the table into smaller groups
- Cannot use column alias in the GROUP clause
- Use WHERE conditions to exclude the rows before grouping, but cannot use for restriction
- Use HAVING clause to restrict the groups
- Use ORDER BY clause to display in order
- GROUP BY clause is mandatory in Nesting-Group functions

QUERIES:

- Select count(*) from employee group by department_id;
- Select department_id, count(last_name) from employee group by department_id;
- Select department_id, count(last_name) from employee group by department_id order by department_id;
- Select department_id, avg(salary) as avg_salary from employee group by department_id having avg(salary) > 4000;
- Select max(avg(salary)) from employee group by department_id;
- Select department_id, sum(salary) from employee group by department_id;
- Select job_id, sum(salary) as payroll From employee Where job_id not like '%002%' Group by job_id Having sum(salary)>4000 order by sum(salary);

Oracle joins:

Oracle joins are used to combine rows from two or more tables based on related columns. Here are explanations of different types of Oracle joins with their syntax and examples:

1. Equi Join:

Syntax:

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

Example:

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

This join returns rows where the values of the joined columns in both tables are equal.

2. Non-Equi Join:

Syntax:

```
SELECT columns FROM table1 JOIN table2 ON table1.column <operator> table2.column;
```

Example:

```
SELECT e.emp_id, e.first_name, m.manager_id
FROM employees e
JOIN employees m ON e.emp_id != m.emp_id AND e.manager_id = m.emp_id;
```

This join uses a comparison operator other than equality, such as `<`, `>`, `<=`, `>=`, `<>`, etc., to join the tables.

3. Outer Joins (Left and Right):

Syntax (Left Outer Join):

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

Example (Left Outer Join):

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
```

This join returns all rows from the left table (table1) and the matching rows from the right table (table2). If no match is found, NULL values are returned for the right table columns.

Syntax (Right Outer Join):

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

Example:

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

In this example, we are performing a right outer join between the "employees" and "departments" tables based on the "department_id" column.

The result will include all rows from the right table "departments" and only the matching rows from the left table "employees". If there is no match found in the left table, NULL values will be returned for the left table columns.

This right outer join will return all the departments, including those that don't have any corresponding employees. The columns selected from the left table ("employees") will have NULL values for those rows where there is no matching department in the right table ("departments").

4. Self Join:

Syntax:

```
SELECT columns FROM table1 t1 JOIN table1 t2 ON t1.column = t2.column;
```

Example:

```
SELECT e.emp_id, e.first_name, m.first_name AS manager_name
FROM employees e
JOIN employees m ON e.manager_id = m.emp_id;
```

This join is used to join a table to itself, typically when there is a hierarchical relationship within the table.

5. Cross Join:

Syntax:

```
SELECT columns FROM table1 CROSS JOIN table2;
```

Example:

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

This join returns the Cartesian product of the two tables, combining each row from the first table with every row from the second table.

6. Natural Join:

Syntax:

```
SELECT columns FROM table1 NATURAL JOIN table2;
```

Example:

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
NATURAL JOIN departments d;
```

This join automatically matches the columns with the same name in both tables.

7. Full Outer Join:

Syntax:

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

Example:

```
SELECT e.emp_id, e.first_name, d.department_name
FROM employees e
FULL JOIN departments d ON e.department_id = d.department_id;
```

This join returns all rows from both tables and includes NULL values where there is no match between the joined columns.

These are the common types of joins in Oracle SQL, each serving different purposes for combining data from multiple tables. The appropriate join type depends on the specific requirements of your query.

Ambiguous Column Error:

The "ambiguous column" error occurs when a column name is referenced in a SQL query, but the database system cannot determine which table the column belongs to due to the column name being used in multiple tables specified in the query.

This error typically occurs in situations where you have joined multiple tables that have columns with the same name, and you reference that column without specifying the table alias or table name. As a result, the database system is unable to determine which table's column should be used in the query, leading to ambiguity.

To resolve the ambiguous column error, you need to provide a clear reference to the column by specifying the table alias or table name along with the column name in your query. Here are some ways to handle ambiguous columns:

1. Specify the table alias or table name before the column name:

```
SELECT t1.column_name, t2.column_name FROM table1 t1 JOIN table2 t2 ON t1.column = t2.column;
```

2. Use the table name or alias as a prefix when referencing the column:

```
SELECT table1.column_name, table2.column_name FROM table1 JOIN table2 ON table1.column = table2.column;
```

By providing a clear reference to the column, you help the database system understand which table's column you are referring to, resolving the ambiguity and eliminating the error.

It's important to note that when joining multiple tables with columns of the same name, it's considered a best practice to use table aliases and explicitly specify the table alias or table name when referencing the columns. This practice enhances the clarity and maintainability of your SQL queries.

[BY PPT]

Natural Join:

- The join can happen on only those columns having same data type and column names
- USING clause:
  - Joins with USING clause, used when column names are same but not the data types
  - Also, Simple join or Equi Join or Inner Join are all same.
  - But do not qualify a column that is used in USING clause
  - If the same column is used elsewhere in the SQL statement, do not alias it
- ON clause:
  - Use ON clause to specify arbitrary conditions or specify column to join
  - The join condition is separated from other search conditions
  - ON clause makes code easy to understand

Queries:

```
SELECT EMP_ID, TO_CHAR(HIRE_DATE,'fmDD MONTH YYYY') AS HIRE_DATE, JOB_ID, DEPARTMENT_ID, DEPARTMENT_NAME FROM EMPLOYEE NATURAL JOIN DEPARTMENT;
```

	EMP_ID	HIRE_DATE	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	101	1 JANUARY 2022	J001	10	Finance
2	102	1 FEBRUARY 2022	J002	20	Sales
3	150	1 MAY 2022	J005	50	IT
4	104	13 JANUARY 2022	J004	15	Human Resources
5	105	4 FEBRUARY 2022	J005	16	DATA
6	106	1 JANUARY 2022	J006	10	Finance
7	107	1 FEBRUARY 2022	J007	20	Sales
8	108	1 MARCH 2022	J008	15	Human Resources
9	109	1 APRIL 2022	J009	16	DATA
10	110	1 MAY 2022	J0010	30	Marketing

```
SELECT EMP_ID, TO_CHAR(HIRE_DATE,'fmDD MONTH YYYY') AS HIRE_DATE, JOB_ID, DEPARTMENT_ID, DEPARTMENT_NAME FROM EMPLOYEE JOIN DEPARTMENT
USING (DEPARTMENT_ID);
```

	EMP_ID	HIRE_DATE	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	101	1 JANUARY 2022	J001	10	Finance
2	102	1 FEBRUARY 2022	J002	20	Sales
3	150	1 MAY 2022	J005	50	IT
4	104	13 JANUARY 2022	J004	15	Human Resources
5	105	4 FEBRUARY 2022	J005	16	DATA
6	106	1 JANUARY 2022	J006	10	Finance
7	107	1 FEBRUARY 2022	J007	20	Sales
8	108	1 MARCH 2022	J008	15	Human Resources
9	109	1 APRIL 2022	J009	16	DATA
10	110	1 MAY 2022	J0010	30	Marketing

```
SELECT E.EMP_ID, TO_CHAR(E.HIRE_DATE,'fmDD MONTH YYYY') AS HIRE_DATE, E.JOB_ID, D.DEPARTMENT_ID, D.DEPARTMENT_NAME FROM EMPLOYEE E JOIN DEPARTMENT D
USING (DEPARTMENT_ID) WHERE  D.DEPARTMENT_ID=20;
```

```
ORA-25154: column part of USING clause cannot have qualifier
25154. 00000 -  "column part of USING clause cannot have qualifier"
*Cause:    Columns that are used for a named-join (either a NATURAL join
           or a join with a USING clause) cannot have an explicit qualifier.
*Action:   Remove the qualifier.
Error at Line: 277 Column: 8
```

We are getting this error because of ‘using’ clause we used in above query, so if you are using a column in ‘using’ clause then you cannot use it elsewhere. So to fix this error we move to next/below query, here comes the use of ‘ON” clause.
Mostly USING clause is not used.

```
SELECT E.EMP_ID, TO_CHAR(E.HIRE_DATE,'fmDD MONTH YYYY') AS HIRE_DATE, E.JOB_ID,
D.DEPARTMENT_ID, D.DEPARTMENT_NAME FROM EMPLOYEE E JOIN DEPARTMENT D
ON E.DEPARTMENT_ID = D.DEPARTMENT_ID WHERE  E.DEPARTMENT_ID=20;
```

	EMP_ID	HIRE_DATE	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	102	1 FEBRUARY 2022	J002	20	Sales
2	107	1 FEBRUARY 2022	J007	20	Sales

```
SELECT E.FIRST_NAME||' '||E.LAST_NAME AS EMPLOYEE_NAME,D.DEPARTMENT_NAME,L.CITY
FROM EMPLOYEE E JOIN DEPARTMENT D ON E.DEPARTMENT_ID=D.DEPARTMENT_ID
JOIN LOCATION L ON D.LOCATION_ID=L.LOCATION_ID;
```

	EMPLOYEE_NAME	DEPARTMENT_NAME	CITY
1	John Doe	Finance	New York City
2	Jane Smith	Sales	Los Angeles
3	David Lee	IT	San Francisco
4	Ankit Gupta	Human Resources	Houston
5	Nandini Gupta	DATA	Houston
6	PRIYA GUPTA	Finance	New York City
7	KAUSHAL GUPTA	Sales	Los Angeles
8	ARUN MODANWAL	Human Resources	Houston
9	LAXMI MODANWAL	DATA	Houston
10	ANJU GUPTA	Marketing	Chicago

```
SELECT E.FIRST_NAME||' '||E.LAST_NAME AS EMPLOYEE_NAME,D.DEPARTMENT_NAME,L.CITY
FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
JOIN LOCATION L USING (LOCATION_ID);
```

	EMPLOYEE_NAME	DEPARTMENT_NAME	CITY
1	John Doe	Finance	New York City
2	Jane Smith	Sales	Los Angeles
3	David Lee	IT	San Francisco
4	Ankit Gupta	Human Resources	Houston
5	Nandini Gupta	DATA	Houston
6	PRIYA GUPTA	Finance	New York City
7	KAUSHAL GUPTA	Sales	Los Angeles
8	ARUN MODANWAL	Human Resources	Houston
9	LAXMI MODANWAL	DATA	Houston
10	ANJU GUPTA	Marketing	Chicago

```
SELECT E.EMP_ID, TO_CHAR(E.HIRE_DATE,'fmDD MONTH YYYY') AS HIRE_DATE, E.JOB_ID,
D.DEPARTMENT_ID, D.DEPARTMENT_NAME FROM EMPLOYEE E JOIN DEPARTMENT D
ON E.DEPARTMENT_ID = D.DEPARTMENT_ID WHERE  E.DEPARTMENT_ID=20 AND
JOB_ID='J007';
```

	E...	HIRE_DATE	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	107	1 FEBRUARY 2022	J007	20	Sales

SELF JOIN:

A self join is a type of join operation in which a table is joined with itself. It involves referencing the same table twice in the join operation and using different aliases to distinguish between the two instances of the table. This allows you to establish a relationship between different rows within the same table.
The ON clause used to join coulmns that have different names, within the table or in different table

Here's an example to illustrate a self join:

Consider a table called "employees" with the following columns:

- `emp_id`: Employee ID (unique identifier)
- `first_name`: First name of the employee
- `last_name`: Last name of the employee
- `manager_id`: ID of the employee's manager (references emp_id)

To perform a self join on the "employees" table to retrieve the names of employees and their respective managers, you can use aliases to differentiate between the employee and manager:

```
SELECT e.first_name AS employee_name, m.first_name AS manager_name FROM employees e
JOIN employees m ON e.manager_id = m.emp_id;
```

In this example, the table "employees" is joined with itself using the aliases "e" for the employee and "m" for the manager. The join condition is specified to match the manager_id of the employee with the emp_id of the manager.

The result of this self join query will provide the names of employees and their respective managers, establishing a relationship within the same table.

QUERIES:

```
SELECT E.FIRST_NAME||' '||E.LAST_NAME EMPLOYEE, M.FIRST_NAME||' '||M.LAST_NAME
MANAGER FROM EMPLOYEE E JOIN EMPLOYEE M ON E.MANAGER_ID=M.EMP_ID;
```

NON-EQUI JOIN:

A non-equi join is a type of join operation where the join condition does not use the equality operator (=) to match values between the joined tables. Instead, it uses other comparison operators such as greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), or not equal to (!= or <>).

In a non-equi join, rows from one table are matched with multiple rows from the other table based on the specified non-equality condition. This type of join allows for more flexible and complex comparisons between the joined tables.

EXAMPLE:

```
SELECT E.FIRST_NAME||' '||E.LAST_NAME AS EMPLOYEE_NAME, E.SALARY, J.GRADE_LEVEL
FROM EMPLOYEE E JOIN JOB_GRADE J ON E.SALARY BETWEEN J.LOWEST_SALARY AND
J.HIGHEST_SALARY;
```

This query demonstrates a non-equijoin by joining the "EMPLOYEE" and "JOB_GRADE" tables based on the condition that the "SALARY" of an employee falls within the range defined by the "LOWEST_SALARY" and "HIGHEST_SALARY" columns of the "JOB_GRADE" table.

The join condition E.SALARY BETWEEN J.LOWEST_SALARY AND J.HIGHEST_SALARY establishes a non-equi join, as it compares the "SALARY" column of the "EMPLOYEE" table to a range defined by the "LOWEST_SALARY" and "HIGHEST_SALARY" columns of the "JOB_GRADE" table.

The result of this query will retrieve the employee's full name, salary, and the corresponding grade level based on the salary falling within the specified range.

Non-equi joins are useful when you need to establish relationships between tables based on conditions other than strict equality, allowing for more flexible and varied join operations.

QUERIES:

```
SELECT E.FIRST_NAME||' '||E.LAST_NAME EMPLOYEE_NAME,E.SALARY,J.GRADE_LEVEL
FROM EMPLOYEE E JOIN JOB_GRADE J ON E.SALARY BETWEEN J.LOWEST_SALARY AND
HIGHEST_SALARY;
```

	EMPLOYEE_NAME	SALARY	GRADE_LEVEL
1	John Doe	5000	A
2	PRIYA GUPTA	5000	A
3	John Doe	5000	B
4	Jane Smith	6000	B
5	PRIYA GUPTA	5000	B
6	KAUSHAL GUPTA	6000	B
7	LAXMI MODANWAL	5500	B
8	John Doe	5000	C
9	Jane Smith	6000	C
10	PRIYA GUPTA	5000	C
11	KAUSHAL GUPTA	6000	C
12	ARUN MODANWAL	7000	C
13	LAXMI MODANWAL	5500	C
14	Jane Smith	6000	D
15	KAUSHAL GUPTA	6000	D
16	ARUN MODANWAL	7000	D
17	ANJU GUPTA	8000	D
18	ARUN MODANWAL	7000	E
19	ANJU GUPTA	8000	E

CROSS JOIN:

A cross join, also known as a Cartesian join, is a type of join operation that produces the Cartesian product of two tables. It combines each row from the first table with every row from the second table, resulting in a combination of all possible pairs of rows.

To explain the concept of a cross join, let's consider two tables created in a hypothetical database:

Table 1:"Colors"
Color
Red
Blue
Green

Table 2: "Sizes"
Size
S
M
L

By performing a cross join between these two tables, we would get the following result:

SELECT * FROM Colors CROSS JOIN Sizes;

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

| Color | Size |

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

| Red | S |

| Red | M |

| Red | L |

| Blue | S |

| Blue | M |

| Blue | L |

| Green | S |

| Green | M |

| Green | L |

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

In the resulting output, each row from the "Colors" table is combined with every row from the "Sizes" table, resulting in all possible combinations.

A cross join is useful in scenarios where you need to generate all possible combinations of rows from two or more tables. However, it can lead to a large result set, so it should be used with caution and filtered further if necessary.

In the example above, a cross join between the "Colors" and "Sizes" tables produces all possible color-size combinations. The resulting dataset can then be further processed or filtered as per the requirements of the specific use case.

QUERIES:

SELECT LAST_NAME,DEPARTMENT_NAME FROM EMPLOYEE CROSS JOIN DEPARTMENT;

Left Join (or Left Outer Join):

The left join returns all records from the left table (the table before the JOIN keyword) and the matching records from the right table (the table after the JOIN keyword). If there is no match, NULL values are returned for the right table.

Example: Retrieve all employees and their corresponding departments, including employees without a department:

SELECT E.*, D.department_name FROM employee E LEFT JOIN department D ON E.department_id = D.department_id;

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	101	John	Doe	john.doe@example.com	1234567890	01-JAN-22	J001	5000	0.1	1001	10	Finance
2	106	PRIYA	GUPTA	PRIYA@example.com	1234567890	01-JAN-22	J006	5000	0.1	1001	10	Finance
3	102	Jane	Smith	jane.smith@example.com	9876543210	01-FEB-22	J002	6000	(null)	1002	20	Sales
4	107	KAUSHAL	GUPTA	KAUSHAL@example.com	9876543210	01-FEB-22	J007	6000	0.2	1002	20	Sales
5	110	ANJU	GUPTA	ANJU@example.com	4445556666	01-MAY-22	J0010	8000	0.18	1003	30	Marketing
6	104	Ankit	Gupta	ankitgupta6009@gmail.com	9673170306	13-JAN-22	J004	60000	0.5	1005	15	Human Resources
7	108	ARUN	MODANWAL	ARUN@example.com	5551234567	01-MAR-22	J008	7000	0.15	1001	15	Human Resources
8	150	David	Lee	david.lee@example.com	4567890123	01-MAY-22	J005	(null)	0.15	1005	50	IT
9	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16	DATA
10	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16	DATA

Right Join (or Right Outer Join):

The right join returns all records from the right table and the matching records from the left table. If there is no match, NULL values are returned for the left table.

Example: Retrieve all departments and their corresponding employees, including departments without any employees:

SELECT D.*, E.first_name, E.last_name FROM department D RIGHT JOIN employee E ON D.department_id = E.department_id;

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	FIRST_NAME	LAST_NAME
1	10	Finance	1	John	Doe
2	10	Finance	1	PRIYA	GUPTA
3	20	Sales	2	Jane	Smith
4	20	Sales	2	KAUSHAL	GUPTA
5	30	Marketing	3	ANJU	GUPTA
6	15	Human Resources	4	Ankit	Gupta
7	15	Human Resources	4	ARUN	MODANWAL
8	50	IT	5	David	Lee
9	16	DATA	4	Nandini	Gupta
10	16	DATA	4	LAXMI	MODANWAL

Inner Join:

The inner join returns only the records that have matching values in both tables. It excludes the unmatched records from both tables.

Example: Retrieve all employees and their corresponding locations, excluding employees without a location:

SELECT E.*, L.city, D.DEPARTMENT_NAME FROM employee E INNER JOIN DEPARTMENT D ON E.department_id = D.department_id INNER JOIN LOCATION L ON D.LOCATION_ID=L.LOCATION_ID;

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	CITY	DEPARTMENT_NAME
1	101	John	Doe	john.doe@example.com	1234567890	01-JAN-22	J001	5000	0.1	1001	10	New York City	Finance
2	102	Jane	Smith	jane.smith@example.com	9876543210	01-FEB-22	J002	6000	(null)	1002	20	Los Angeles	Sales
3	150	David	Lee	david.lee@example.com	4567890123	01-MAY-22	J005	(null)	0.15	1005	50	San Francisco	IT
4	104	Ankit	Gupta	ankitgupta6009@gmail.com	9673170306	13-JAN-22	J004	60000	0.5	1005	15	Houston	Human Resources
5	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16	Houston	DATA
6	106	PRIYA	GUPTA	PRIYA@example.com	1234567890	01-JAN-22	J006	5000	0.1	1001	10	New York City	Finance
7	107	KAUSHAL	GUPTA	KAUSHAL@example.com	9876543210	01-FEB-22	J007	6000	0.2	1002	20	Los Angeles	Sales
8	108	ARUN	MODANWAL	ARUN@example.com	5551234567	01-MAR-22	J008	7000	0.15	1001	15	Houston	Human Resources
9	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16	Houston	DATA
10	110	ANJU	GUPTA	ANJU@example.com	4445556666	01-MAY-22	J0010	8000	0.18	1003	30	Chicago	Marketing

Full Join (or Full Outer Join):

The full join returns all records from both the left and right tables. It includes the matching records from both tables and NULL values for the non-matching records.

Example:

SELECT E.*, D.department_name FROM employee E FULL JOIN department D ON E.department_id = D.department_id;

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	101	John	Doe	john.doe@example.com	1234567890	01-JAN-22	J001	5000	0.1	1001	10	Finance
2	102	Jane	Smith	jane.smith@example.com	9876543210	01-FEB-22	J002	6000	(null)	1002	20	Sales
3	150	David	Lee	david.lee@example.com	4567890123	01-MAY-22	J005	(null)	0.15	1005	50	IT
4	104	Ankit	Gupta	ankitgupta6009@gmail.com	9673170306	13-JAN-22	J004	60000	0.5	1005	15	Human Resources
5	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16	DATA
6	106	PRIYA	GUPTA	PRIYA@example.com	1234567890	01-JAN-22	J006	5000	0.1	1001	10	Finance
7	107	KAUSHAL	GUPTA	KAUSHAL@example.com	9876543210	01-FEB-22	J007	6000	0.2	1002	20	Sales
8	108	ARUN	MODANWAL	ARUN@example.com	5551234567	01-MAR-22	J008	7000	0.15	1001	15	Human Resources
9	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16	DATA
10	110	ANJU	GUPTA	ANJU@example.com	4445556666	01-MAY-22	J0010	8000	0.18	1003	30	Marketing

In this example, we are performing a full join between the "employee" and "department" tables based on the common column "department_id". This query will return all records from both tables, including matched records and unmatched records.

The result will include all employees and their corresponding department names. If there is a matching department ID between the tables, the department name will be displayed. If there is no match, NULL values will be returned for the department name.

Please note that the specific column names and relationships in your actual tables may vary, so ensure to adjust the query accordingly based on your table structure.

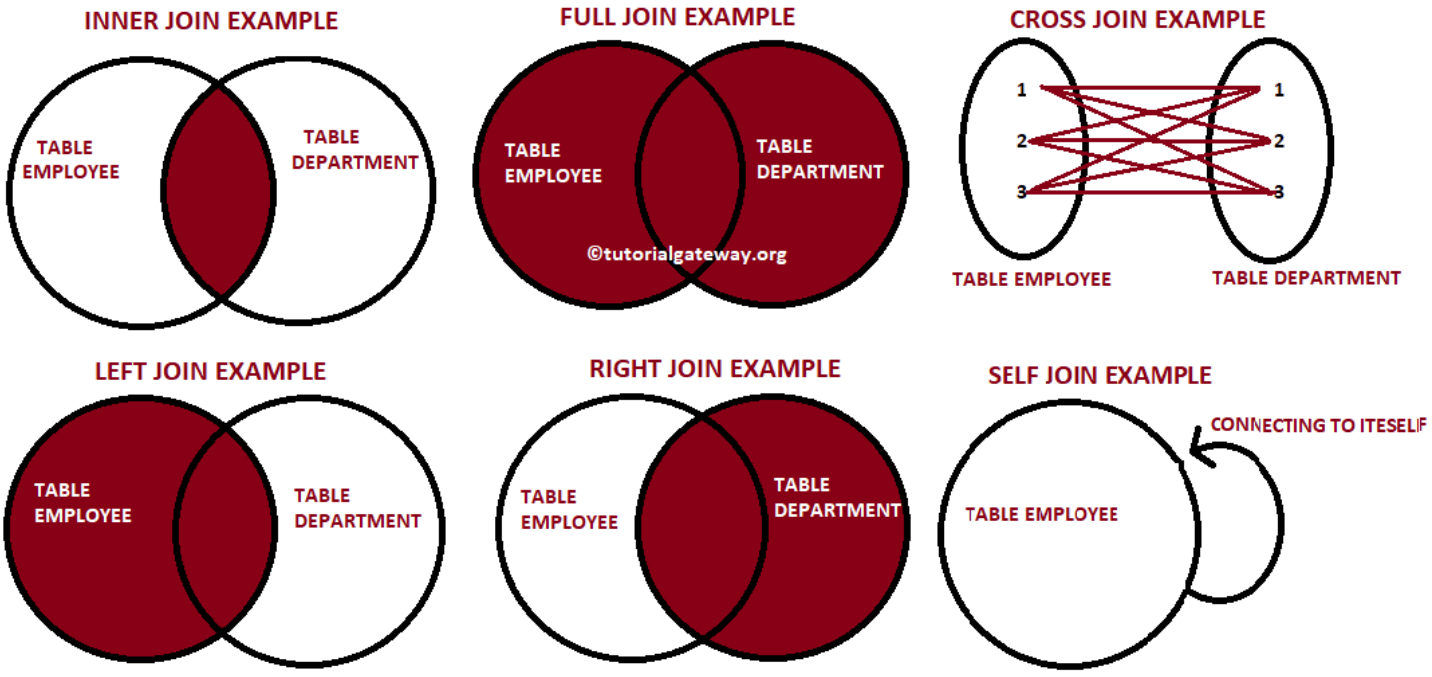
Note: Ensure that the column names and table names used in the examples match your actual table structure.

ANKKIT KUMAR GUPPTA

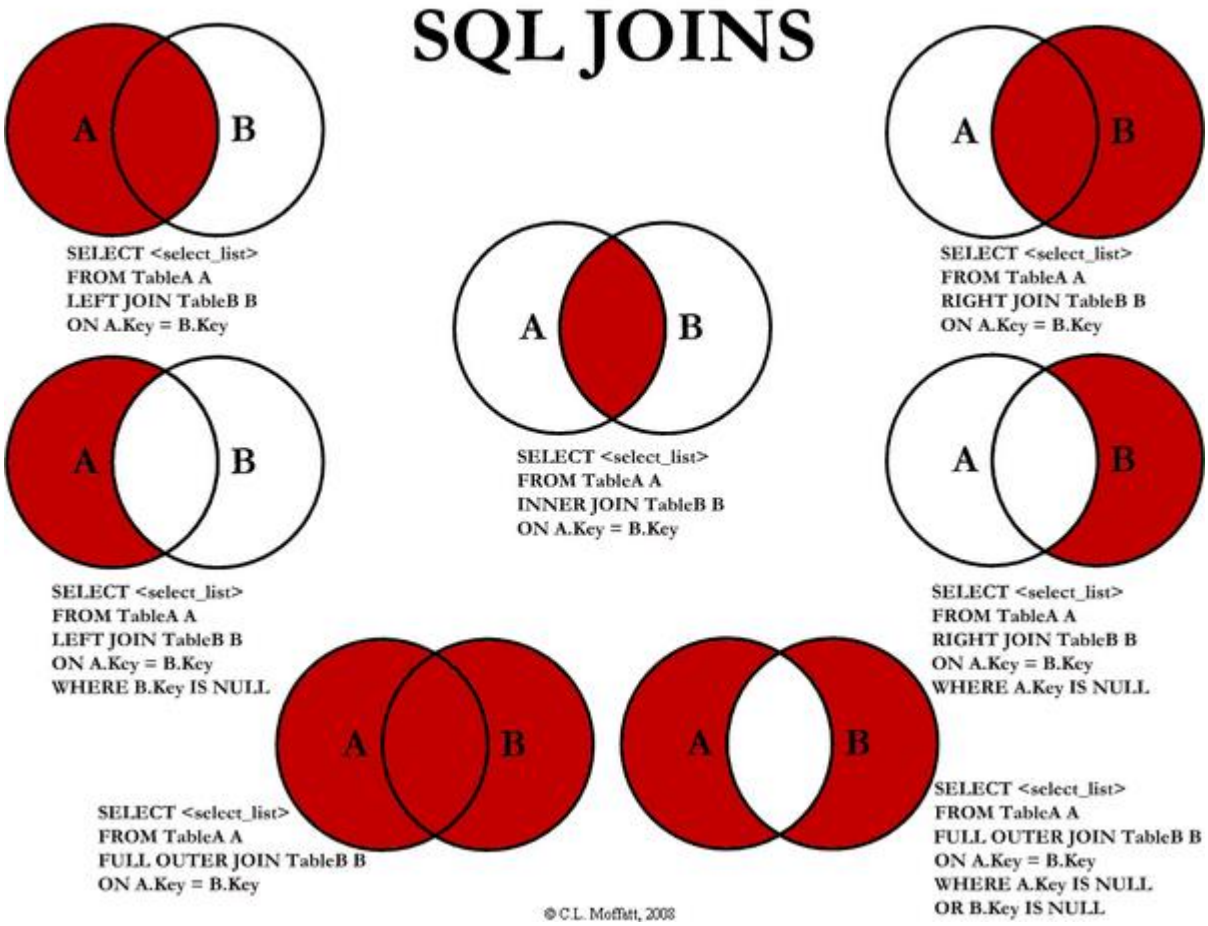
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

1.



2.



ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

SUB-QUERIES:

A subquery, also known as a nested query or inner query, is a query that is embedded within another query. It allows you to retrieve data from one table based on the results or conditions of another query. Subqueries can be used in different parts of a SQL statement, including the SELECT, FROM, WHERE, and HAVING clauses.

Here's a breakdown of the components and usage of subqueries:

Subquery in the SELECT clause:

```
SELECT column1, column2, (SELECT subquery) FROM table;
```

Subquery in the FROM clause:

```
SELECT column1, column2 FROM (SELECT subquery) AS alias;
```

Subquery in the WHERE clause:

```
SELECT column1, column2 FROM table WHERE column1 = (SELECT subquery);
```

Subquery in the HAVING clause:

```
SELECT column1, COUNT(column2) FROM table GROUP BY column1 HAVING COUNT(column2) > (SELECT subquery);
```

Usage and Purpose:

Filter Data: Subqueries can be used to filter data by providing conditions based on the results of the subquery. For example, retrieving employees who earn more than the average salary:

```
SELECT employee_name, salary FROM employee WHERE salary > (SELECT AVG(salary) FROM employee);
```

Generate Derived Data: Subqueries can be used to generate derived data or computed values within a query. For example, calculating the percentage of employees in each department:

```
SELECT department_name, (SELECT COUNT(*) FROM employee WHERE department_id = d.department_id) / (SELECT COUNT(*) FROM employees) AS percentage FROM departments d;
```

Perform Joins: Subqueries can be used to perform joins between tables within a query. For example, retrieving employees who belong to departments with a specific location:

```
SELECT employee_name FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE location = 'New York');
```

Subqueries can also be used with INSERT, UPDATE, and DELETE statements to perform operations based on the results of the subquery.

It's important to note that subqueries can have an impact on performance, so it's advisable to optimize queries and consider alternative approaches like joins or temporary tables when dealing with complex scenarios.

[BY PPT]

WHAT IS SUB-QUERY?

- A Sub-Query is a SELECT statement that is embedded in the clause of another SELECT statement.
- Can place the sub-query in number of SQL clauses:
  - WHERE
  - HAVING
  - FROM
- It is also referred as Nested select, Sub-select and Inner select
- Sub-Query executes first, and then the main query

NOTE: Comparison conditions is of 2 classes:

1. Single row operators (=, <>, <, <=,>,>=)

2. Multiple row operators (IN, ANY, ALL, EXISTS)

Types of Sub-Queries: (single & multirow sub-query is more important)

There are several types of subqueries that can be used in SQL. Let's explore each type with examples:

1. **Scalar Subquery:**
   A scalar subquery is a subquery that returns a single value, typically used within the SELECT, WHERE, or HAVING clauses.

   Example: Retrieve the names of employees who have a higher salary than the average salary.

   SELECT First_Name, Salary FROM employee WHERE salary > (SELECT AVG(salary) FROM employee);

	FIRST_NAME	SALARY
1	Ankit	60000
2	Nandini	50000

2. **Single-Row Subquery:**
   A single-row subquery is a subquery that returns a single row and can be used in places where a single value is expected.

   Example: Retrieve the name of the department where the employee with ID 101 belongs.

   SELECT department_name FROM department WHERE department_id = (SELECT department_id FROM employees WHERE emp_id = 101);

	DEPARTMENT_NAME
1	Finance

3. **Multiple-Row Subquery:**
   A multiple-row subquery is a subquery that returns multiple rows and can be used in places where multiple values are expected.

   Example: Retrieve the names of employees who have the same Salary as the employee with ID 102.

   SELECT First_Name FROM employee WHERE salary IN (SELECT salary FROM employee WHERE emp_id = 102);

	FIRST_NAME
1	Jane
2	KAUSHAL

4. **Correlated Subquery:**
   A correlated subquery refers to a subquery that depends on the values from the outer query and is executed for each row of the outer query.

   Example: Retrieve the names of employees who earn more than the average salary within their respective departments.

   SELECT First_Name FROM employee e WHERE salary > (SELECT AVG(salary) FROM employee WHERE department_id = e.department_id);

	FIRST_NAME
1	Ankit
2	Nandini

5. **Nested Subquery:**
   A nested subquery is a subquery that is placed within another subquery.

   Example: Retrieve the names of employees who belong to departments located in the same city as the department with ID 20.

SELECT e.First_Name FROM employee e WHERE e.department_id IN (SELECT d.department_id FROM department d JOIN location l ON d.location_id = l.location_id WHERE l.city = (SELECT l.city FROM department d JOIN location l ON d.location_id = l.location_id WHERE d.department_id = 20));

	FIRST_NAME
1	Jane
2	KAUSHAL

These are the commonly used types of subqueries in SQL. Each type serves different purposes and allows for more complex and flexible querying capabilities by leveraging the results of one query within another.

Define all the quantifiers in SQL with example

In SQL, quantifiers are used in combination with subqueries to compare a value with a set of values. Here are the commonly used quantifiers along with examples using the Employee, Department, and Location tables:

'ANY' Quantifier:
The 'ANY' quantifier returns true if the comparison condition is true for at least one value in the set.

Example:
```
SELECT department_name FROM departments WHERE department_id = ANY (SELECT department_id FROM employees WHERE salary > 5000);
```

This query retrieves the names of departments where at least one employee has a salary greater than 5000.

'ALL' Quantifier:
The 'ALL' quantifier returns true if the comparison condition is true for all values in the set.

Example:
```
SELECT department_name FROM departments WHERE department_id = ALL (SELECT department_id FROM employees);
```

This query retrieves the names of departments where all employees belong to the same department.

'SOME' or 'ANY' Quantifier:
The 'SOME' or 'ANY' quantifier is synonymous and can be used interchangeably. It returns true if the comparison condition is true for at least one value in the set.

Example:
```
SELECT department_name FROM departments WHERE department_id = SOME (SELECT department_id FROM employees WHERE salary > 5000);
```

This query retrieves the names of departments where at least one employee has a salary greater than 5000. It is equivalent to using the 'ANY' quantifier.

'IN' Operator:
The 'IN' operator is a shorthand notation for the 'ANY' quantifier. It returns true if the value matches any value in the set.

Example:
```
SELECT employee_name FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 1);
```

This query retrieves the names of employees who belong to departments located in Location ID 1.

These are the commonly used quantifiers in SQL. They allow for comparisons with sets of values and enable more flexible querying and data manipulation based on conditions.

Set Operators:

1. Set operators combine the results of 2 or more component queries into one result.

Set operators are used to combine the results of multiple SELECT queries into a single result set. The common set operators in SQL are UNION, INTERSECT, and EXCEPT (or MINUS, depending on the database system). Each set operator has specific rules for combining the results.

Example:
Let's say we want to retrieve the names of employees from two tables, "employee" and "temporary_employee." We can use the UNION operator to combine the results:

```
SELECT First_Name FROM employee
UNION
SELECT First_Name FROM temporary_employee;
```

2. Queries having set operators are called as compound queries.

A compound query is a SQL query that involves the use of set operators (UNION, INTERSECT, or EXCEPT) to combine the results of multiple queries. These queries are also referred to as compound queries because they combine the results of component queries into a single result set.

Example:
Continuing from the previous example, if we use the UNION operator to combine the results of two queries, it becomes a compound query.

3. Guidelines:

- The expressions in the select lists must match in number.

The SELECT lists in all component queries of a compound query must have the same number of expressions, and the expressions must have compatible data types.

Example:
Let's consider two tables, "employees" and "managers," and we want to retrieve the names and salaries of employees and managers together using the UNION operator:

```
SELECT First_Name, Salary FROM employees
UNION
SELECT First_Name, Salary FROM managers;
```

- The data type of each column in the second query must match the data type of its corresponding column in the first query.

- Parentheses can be used to alter the sequence of execution.
  Parentheses can be used to group & change the sequence of execution when using multiple set operators in a single query.
- ORDER BY clause can appear only at the very end of the statement.
  The ORDER BY clause must be applied to the final result set of the compound query, and it should be placed at the end of the entire statement.

Example:
Let's retrieve the names of employees and managers from the "employees" and "managers" tables, respectively, and sort the results alphabetically:

```
(SELECT First_Name FROM employees)
UNION
(SELECT First_Name FROM managers)
ORDER BY First_Name;
```

In summary, set operators and compound queries in SQL allow us to combine and manipulate data from multiple queries. By following the guidelines and using appropriate parentheses and ORDER BY clauses, we can create powerful and flexible queries to extract valuable insights from the data.

KEY NOTES:

- Column names from the first query appear in the result
- Duplicate rows are automatically eliminated except in union all
- The output is sorted in ascending order by default except in union all

TYPES OF SET OPERATORS:

1. UNION OPERATOR

The UNION set operator in SQL is used to combine the results of two or more SELECT queries into a single result set. It removes duplicate rows from the final result, meaning that each row in the result set will be unique
.

- Returns rows from both queries after eliminating duplications.
- Number of columns being selected must be same
- Data types of selected columns must be in the same data type group
- Column names need not be same
- Union operates over, all of the selected columns
- NULL values are not ignored while checking the duplicates
- Output is sorted in ascending order

Syntax: The basic syntax of the UNION set operator is as follows:

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

Each SELECT query must have the same number of columns in the SELECT list, and the data types of corresponding columns in different SELECT queries must be compatible.

Example:
Consider two tables, "**employees**" and "**temporary_employees**," both with a similar structure:

employees table:

emp_id	first_name	last_name	salary
1	John	Doe	50000
2	Jane	Smith	55000

temporary_employees table:

emp_id	first_name	last_name	salary
3	Michael	Brown	48000
4	Emma	Johnson	52000

Now, let's use the UNION operator to combine the names and salaries of employees from both tables:

```
SELECT first_name, last_name, salary FROM employees
UNION
SELECT first_name, last_name, salary FROM temporary_employees;
```

Result:

first_name	last_name	salary
John	Doe	50000
Jane	Smith	55000
Michael	Brown	48000
Emma	Johnson	52000

In this example, the UNION operator combined the results from both tables and removed duplicate rows (if any) to give us a single result set. The final result contains the unique names and salaries of all employees from both tables. If there were any duplicate rows, only one instance of each duplicate would have appeared in the result due to the removal of duplicates by the UNION operator.

2. UNION ALL OPERATOR

The UNION ALL set operator in SQL is similar to the UNION operator but **does not remove duplicate** rows from the combined result set. It simply concatenates the results of multiple SELECT queries into a single result set, including all rows from each query, even if there are duplicates.

- Returns rows from both queries, including all duplicates
- Both UNION and UNION ALL are same except below 2
- Duplicate rows are not eliminated
- Outputs are not sorted by default

Syntax: The basic syntax of the UNION ALL set operator is as follows:

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...
FROM table2;
```

Each SELECT query must have the same number of columns in the SELECT list, and the data types of corresponding columns in different SELECT queries must be compatible.

Example:
Consider two tables, "**employees**" and "**temporary_employees**," with similar data as before:

employees table:

emp_id	first_name	last_name	salary
1	John	Doe	50000
2	Jane	Smith	55000

temporary_employees table:

emp_id	first_name	last_name	salary
3	Michael	Brown	48000
4	Emma	Johnson	52000
5	John	Doe	50000

Now, let's use the UNION ALL operator to combine the names and salaries of employees from both tables:

```
SELECT first_name, last_name, salary FROM employees
UNION ALL
SELECT first_name, last_name, salary FROM temporary_employees;
```

Result:

first_name	last_name	salary
John	Doe	50000
Jane	Smith	55000
Michael	Brown	48000
Emma	Johnson	52000
John	Doe	50000

In this example, the UNION ALL operator combined the results from both tables, including all rows from each table, even if there were duplicate rows. As a result, the final result set contains all the names and salaries from both tables, including the duplicate entry for John Doe.

The key difference between UNION and UNION ALL is that UNION removes duplicate rows, while UNION ALL does not. Therefore, if you need to retain duplicate rows, you can use UNION ALL to combine the results of multiple queries.

3. INTERSECT OPERATOR:

The INTERSECT operator in SQL is used to return the common records (or the intersection) between two or more SELECT statements. It combines the result sets of two or more SELECT queries and returns only the rows that appear in all result sets.

- Returns rows that are common to both queries.
- Number of columns and the data types of the columns must be identical
- Column names can be different
- Revising the order of the tables, does not change the result
- NULL values are not ignored

Syntax: The basic syntax of the INTERSECT operator is as follows:

```
SELECT column1, column2, ...FROM table1
INTERSECT
SELECT column1, column2, ...FROM table2;
```

Each SELECT query must have the same number of columns in the SELECT list, and the data types of corresponding columns in different SELECT queries must be compatible.

Example:
Consider two tables, "**employees**" and "**temporary_employees**," with similar data as before:

employees table:

emp_id	first_name	last_name	salary
1	John	Doe	50000
2	Jane	Smith	55000

temporary_employees table:

emp_id	first_name	last_name	salary
3	Michael	Brown	48000
4	John	Doe	50000

Now, let's use the INTERSECT operator to find the common names and salaries of employees from both tables:

```
SELECT first_name, last_name, salary FROM employees
INTERSECT
SELECT first_name, last_name, salary FROM temporary_employees;
```

Result:

first_name	last_name	salary
John	Doe	50000

In this example, the INTERSECT operator returned the common row from both tables. The only common row between the two tables is the one for John Doe with a salary of 50000.

So, the INTERSECT operator can be very useful when you want to find common rows between two or more tables.

4. MINUS OPERATOR:

The MINUS operator in SQL is used to retrieve the rows that are present in the result of the first query but not present in the result of the second query. It essentially performs a set difference operation between the two result sets.

- Returns all the distinct rows selected by the first query, but not present in the second query result set.
- Number of columns must be same
- Data type must be from same data type group
- Column names can be different

Syntax: The basic syntax of the MINUS operator is as follows:

```
SELECT column1, column2, ...FROM table1
MINUS
SELECT column1, column2, ...FROM table2;
```

Each SELECT query must have the same number of columns in the SELECT list, and the data types of corresponding columns in different SELECT queries must be compatible.

Example:
Consider two tables, "**employees**" and "**temporary_employees**," with similar data as before:

employees table:

emp_id	first_name	last_name	salary
1	John	Doe	50000
2	Jane	Smith	55000

temporary_employees table:

emp_id	first_name	last_name	salary
3	Michael	Brown	48000
4	John	Doe	50000

Now, let's use the MINUS operator to find the employees who are not temporary employees:

```
SELECT first_name, last_name, salary FROM employees
MINUS
SELECT first_name, last_name, salary FROM temporary_employees;
```

Result:

first_name	last_name	salary
Jane	Smith	55000

In this example, the MINUS operator returned the row for Jane Smith, as she is an employee but not a temporary employee. The row for John Doe is not included in the result because he appears in both tables, and the MINUS operator removes rows that are common in both tables.

So, the MINUS operator can be used to find the differences between two result sets, helping you to identify the records that are present in one set but not in the other.

MATCHING THE SELECT STATEMENTS

- For UNION operator, must match the number of columns and data types.
- If column is not present in other table have to use the DUMMY table to match the SELECT statement to perform UNION operator

ORDER BY IN SET OPERATIONS

- The order by clause can appear only once at the end of the compound query
- The component queries cannot have independent order by clauses
- The order by recognizes only the columns of the first SELECT query
- By default, the first column of the first SELECT query is used to sort the output in an ascending order
- Cannot use second query to sort the result

Queries:

```
SELECT DEPARTMENT_ID FROM EMPLOYEE
UNION
SELECT DEPARTMENT_ID FROM DEPARTMENT ORDER BY DEPARTMENT_ID;
```

```
SELECT DEPARTMENT_ID FROM EMPLOYEE
UNION ALL
SELECT DEPARTMENT_ID FROM DEPARTMENT ORDER BY DEPARTMENT_ID;
```

```
SELECT DEPARTMENT_ID FROM EMPLOYEE
INTERSECT
SELECT DEPARTMENT_ID FROM DEPARTMENT;
```

```
SELECT DEPARTMENT_ID FROM EMPLOYEE
MINUS
SELECT DEPARTMENT_ID FROM DEPARTMENT ORDER BY DEPARTMENT_ID;
```

```
SELECT LOCATION_ID, DEPARTMENT_NAME "DEPARTMENT", TO_CHAR(NULL) "Warehouse
Location" FROM DEPARTMENT
UNION
SELECT LOCATION_ID, TO_CHAR (NULL) "Department", STATE_PROVINCE FROM LOCATION;
```

- We use (TO_CHAR (NULL) "Department") as a dummy column because there is no column named as Department in Table LOCATION.
- We use (TO_CHAR(NULL) "Warehouse Location") as a dummy column because there is no column named as Warehouse Location in Table Department
- we done this just to match the number of columns.

```
SELECT EMP_ID, DEPARTMENT_ID FROM EMPLOYEE
UNION
SELECT DEPARTMENT_ID, 0 FROM DEPARTMENT ORDER BY DEPARTMENT_ID DESC;
```

- Order by clause is always going to refer the first query so "ORDER BY DEPARTMENT_ID DESC;" is referring to first query i: e, "SELECT EMP_ID, DEPARTMENT_ID"
- Column names are also retrieved from the first query only

READ CONSISTENCY & FOR UPDATE:

- Read consistency guarantees a consistent view of data at all times.
  - Read operation (SELECT)
  - Write operation (INSERT, UPDATE, DELETE)
- Changes made by one user do not conflict with changes made by another user
- It ensures that on the same data
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers

Read consistency is a crucial concept in database management systems that ensures users receive a consistent and reliable view of data at all times, regardless of concurrent read and write operations by multiple users. Read consistency applies to read operations (SELECT) and write operations (INSERT, UPDATE, DELETE) performed on the database.

Key aspects of read consistency:

1. Consistent View of Data: Read consistency guarantees that each user sees a consistent snapshot of the data in the database, irrespective of any ongoing write operations by other users. This means that the data accessed by a user remains unchanged during their entire transaction.

2. Avoiding Conflicts: Changes made by one user's transaction do not conflict with changes made by another user's transaction. This ensures that each user's transaction works independently and doesn't affect the data being accessed by other users.

3. Non-Blocking Reads and Writes: Read consistency ensures that readers (users executing SELECT queries) do not wait for writers (users performing INSERT, UPDATE, DELETE operations), and writers do not wait for readers. This concurrency control mechanism allows multiple users to access and modify the data simultaneously without interfering with each other's operations.

4. Writers Wait for Writers: In cases where multiple users are trying to modify the same data simultaneously (e.g., two users updating the same row), write operations are serialized, and writers may have to wait for each other to complete their updates.

Example:
Suppose we have a database table named "employees" with columns "emp_id," "first_name," "last_name," and "salary." Multiple users are concurrently accessing and modifying data in the "employees" table. Read consistency ensures that each user sees a consistent and stable view of the data throughout their transactions.

User 1 executes a SELECT query to retrieve employee information:

```
SELECT first_name, last_name, salary FROM employees  WHERE emp_id = 101;
```

User 2 simultaneously updates the salary of employee 101:

```
UPDATE employees SET salary = salary * 1.1 WHERE emp_id = 101;
```

Read consistency guarantees that when User 1 executes the SELECT query, they will see the original salary of employee 101 before the update by User 2. It ensures that User 1's query is not affected by User 2's write operation.

In summary, read consistency is a critical aspect of database management systems that ensures users have a reliable and non-blocking experience while accessing and modifying data concurrently. It helps maintain data integrity and prevents conflicts between users' transactions, ensuring a consistent view of the data at all times.

FOR UPDATE IN SELECT

- SELECT will not lock the records
- Only the records which has been changed but not committed are locked, but still others can see the previous commit status of those records.
- If in case you want to lock the record & then do the changes & commit, then **FOR UPDATE** is used.
- FOR UPDATE enables row level locks as per the WHERE condition in the SELECT, hence others cannot do the change until the lock is released by COMMIT / ROLLBACK
- DEFAULT option is to WAIT
- NOWAIT, is one of the optional keyword to be used with FOR UPDATE, if you don't want your query to wait until the lock is released on the table

1. SELECT will not lock the records:

When you use a simple SELECT statement without the FOR UPDATE clause, it only reads data from the table and does not lock any records. Other transactions can still modify or delete these records while your SELECT statement is running.

Example: Simple SELECT without FOR UPDATE

```
SELECT first_name, last_name, salary FROM employees;
```

In this example, the SELECT statement retrieves data from the "employees" table without locking any records. Other transactions can still modify or delete the data in the "employees" table while this SELECT statement is running.

2. Only the records which have been changed but not committed are locked, but still, others can see the previous commit status of those records:

If you use the FOR UPDATE clause in your SELECT statement, it locks only the records that meet the conditions specified in the WHERE clause and are not yet committed in the current transaction. Other transactions can still see the committed data, but they cannot modify or delete the locked records until the lock is released.

Example:

```
-- Transaction 1
BEGIN;
UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;

-- Transaction 2 (Different session)
SELECT first_name, last_name, salary FROM employees WHERE department_id = 10
FOR UPDATE;
```

In this example, Transaction 1 updates the salaries of employees in department 10 but has not committed the changes. In Transaction 2, the SELECT statement with FOR UPDATE only locks the rows with department_id = 10 that have not been committed yet, allowing Transaction 2 to see the previous salary values of the locked rows.

3. If, in case, you want to lock the record & then do the changes & commit, then FOR UPDATE is used:

The FOR UPDATE clause is commonly used when you want to lock certain rows for update within your transaction. It ensures that no other transactions can modify or delete the locked rows until your transaction is completed (committed or rolled back).

Example:

```
-- Transaction
BEGIN;
SELECT first_name, last_name, salary FROM employees WHERE department_id = 20
FOR UPDATE;

-- Perform updates on the selected rows
UPDATE employees SET salary = salary * 1.05 WHERE department_id = 20;

-- Commit the transaction to release locks
COMMIT;
```

In this example, the SELECT statement with FOR UPDATE locks the rows with department_id = 20 for update. The subsequent UPDATE statement modifies the salaries of those employees. The locks will be released when the transaction is committed.

4. FOR UPDATE enables row-level locks as per the WHERE condition in the SELECT, hence others cannot do the change until the lock is released by COMMIT/ROLLBACK:

When you use FOR UPDATE in a SELECT statement, it enables row-level locks on the selected rows based on the conditions specified in the WHERE clause. These locks prevent other transactions from modifying or deleting the locked rows until the lock is released by a COMMIT or ROLLBACK statement.

Example:

```
-- Transaction 1
BEGIN;
SELECT first_name, last_name, salary FROM employees WHERE department_id = 30
FOR UPDATE;

-- Transaction 2 (Different session)
UPDATE employees SET salary = salary * 1.08 WHERE department_id = 30;
```

In this example, Transaction 1 locks the rows in department 30 using FOR UPDATE. Transaction 2 tries to update the salaries of the same employees in department 30 but will have to wait until Transaction 1 releases the locks (via COMMIT or ROLLBACK) before it can make changes.

5. DEFAULT option is to WAIT:

By default, when you use FOR UPDATE without any other options, the SELECT statement will wait for the locks to be released by other transactions before returning the results. This ensures that the selected rows remain locked until your transaction finishes.

Example:

```
-- Transaction 1
BEGIN;
SELECT first_name, last_name, salary FROM employees WHERE department_id = 40
FOR UPDATE;

-- Transaction 2 (Different session)
SELECT first_name, last_name, salary FROM employees WHERE department_id = 40
FOR UPDATE;
```

In this example, both Transaction 1 and Transaction 2 are trying to lock the rows with department_id = 40 using FOR UPDATE. By default, Transaction 2 will wait for Transaction 1 to release the locks before it can proceed.

6. NOWAIT is one of the optional keywords to be used with FOR UPDATE if you don't want your query to wait until the lock is released on the table:

By using the NOWAIT option along with FOR UPDATE in your SELECT statement, you instruct the database not to wait if the selected rows are already locked by another transaction. Instead, the statement will raise an error immediately if it cannot acquire the locks, allowing you to handle the situation programmatically.

Example:

```
-- Transaction 1
BEGIN;
SELECT first_name, last_name, salary FROM employees WHERE department_id = 50
FOR UPDATE NOWAIT;

-- Transaction 2 (Different session)
SELECT first_name, last_name, salary FROM employees WHERE department_id = 50
FOR UPDATE NOWAIT;
```

In this example, both Transaction 1 and Transaction 2 are trying to lock the rows with department_id = 50 using FOR UPDATE with NOWAIT option. If the rows are already locked by Transaction 1, Transaction 2 will immediately raise an error and will not wait for the locks to be released.

Remember that the use of FOR UPDATE should be carefully considered to avoid potential locking conflicts and deadlocks. It is essential to understand the transaction behavior and concurrency requirements of your application to determine when and how to use the FOR UPDATE clause effectively.

WHAT HAPPENS INSIDE WHEN WE DO DML?

- Read consistency is an automatic implementation.
- It keeps the partial copy in the UNDO SEGMENT (snapshot of the old table data)
- When the changes are done by one user, other users see the data from the snapshot
- Once the user COMMIT or ROLLBACK, others will be able to see the real data instead of snapshot
- When COMMITTED, new data will be available to view
- When ROLLBACK, old data will be reverted and be available for the users
- Also, the UNDO SEGMENT will be released of memory for reuse.

ANKKIT KUMAR GUPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

Inside a Database Management System (DBMS), when we perform Data Manipulation Language (DML) operations (such as INSERT, UPDATE, DELETE, etc.), the system ensures read consistency for concurrent transactions. Let's understand the process step by step:

1. Read Consistency is an automatic implementation:
When a transaction starts reading data from a table, the system ensures that the data remains consistent throughout the transaction, even if other transactions modify the same data simultaneously.

2. UNDO SEGMENT and Snapshot:
The UNDO SEGMENT is a storage area in the database that holds the previous versions (snapshots) of data that are being modified during a transaction. When a DML operation is initiated, the system takes a snapshot of the data that is about to be modified and stores it in the UNDO SEGMENT.

3. Snapshot Visibility:
While the transaction is in progress and making changes, other users see the data from the snapshot stored in the UNDO SEGMENT instead of seeing the real-time data. This ensures that users get a consistent view of the data while a transaction is underway.

4. COMMIT and ROLLBACK:
When the user issues a COMMIT statement, the changes made in the transaction are permanently saved in the database. Other users can now see the new data.

On the other hand, if the user issues a ROLLBACK statement, all the changes made in the transaction are discarded, and the data is reverted to the state it was in before the transaction started.

5. Release of UNDO SEGMENT:
After the COMMIT or ROLLBACK, the UNDO SEGMENT is released from memory, making it available for reuse by other transactions.

The implementation of read consistency and the use of UNDO SEGMENT ensure that the database maintains data integrity and provides a consistent view of data to all users, even when multiple transactions are executing concurrently. This ensures that each transaction sees a logically consistent state of the data and prevents any data corruption or interference between concurrent transactions.

FOR UPDATE OF COLUMN IN SELECT

- FOR UPDATE will lock all the rows involved in the where condition.
- We can use FOR UPDATE for multiple tables also(join statement)
- If used in join statement,then we can exclusively mention to lock only the relevant rows instead of locking all rows from both the tables
- Hence, FOR UPDATE OF column_name

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

The "FOR UPDATE" clause in a SELECT statement is used in database systems to explicitly request a lock on the selected rows. This is particularly useful in situations where you want to ensure that the selected rows are not modified by other transactions until your transaction is completed. Let's break down the points you provided:

1. Locking Rows with FOR UPDATE:
When you include the "FOR UPDATE" clause in your SELECT statement, it will lock the rows that match your SELECT conditions. This prevents other transactions from modifying or accessing these rows until your transaction is completed.

2. Locking Rows in Join Statements:
You can use "FOR UPDATE" in a JOIN statement involving multiple tables. In this case, you can specify which rows you want to lock, rather than locking all rows from all tables involved in the JOIN.

3. Selective Locking with FOR UPDATE OF Column:
The "FOR UPDATE OF column_name" syntax allows you to specify a particular column that you want to lock. This can be useful when you don't need to lock the entire row, but only a specific column's value. Other transactions can still read the row, but they won't be able to modify the locked column until your transaction is completed.

Example:
Suppose you want to update the salary of an employee and want to ensure that no other transactions modify the salary of the same employee while you're making the update:

```
-- Locks the row for the specified employee's salary update

SELECT salary FROM employee WHERE emp_id = 101
FOR UPDATE OF salary;
```

In this example, the row containing the salary information of the employee with emp_id 101 is locked exclusively for your transaction, preventing other transactions from modifying the salary until your transaction is committed or rolled back.

By using "FOR UPDATE" and "FOR UPDATE OF column_name," you can control the locking behavior and ensure data integrity in concurrent transaction scenarios.

Queries:

- SELECT EMP_ID,FIRST_NAME, HIRE_DATE, JOB_ID, MANAGER_ID FROM EMPLOYEE
  WHERE EMP_ID=110
  FOR UPDATE ORDER BY EMP_ID; [--ALL ROWS ARE LOCKED FOR WHERE CONDITION]

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE
  ORDER BY E.EMP_ID; [--ALL ROWS FOR THE BOTH TABLE LOCKED FOR THE WHERE CONDITION]

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE OF E.SALARY
  ORDER BY E.EMP_ID; [--ALL ROWS OF EMPLOYEE TABLE LOCKED BECAUSE E.SALARY IS MEANT TO BE LOCKED EXCLUSIVELY NOT ANY ROWS FORM DEPARTMENTS TABLE]

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE WAIT 30 [--NUMBERS OF SCEONDS TO WAIT] ORDER BY E.EMP_ID;

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE NOWAIT ORDER BY E.EMP_ID;

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE OF E.SALARY WAIT 30 [--NUMBERS OF SCEONDS TO WAIT]
  ORDER BY E.EMP_ID;

- SELECT E.EMP_ID,E.FIRST_NAME, E.SALARY, E.COMMISSION_PCT, D.LOCATION_ID
  FROM EMPLOYEE E JOIN DEPARTMENT D USING (DEPARTMENT_ID)
  WHERE E.JOB_ID='J0010' AND D.LOCATION_ID=3
  FOR UPDATE OF E.SALARY NOWAIT [--DON'T WAIT] ORDER BY E.EMP_ID;

Schema Objects:

Schema objects are database structures that are created and managed within a database schema. A schema is a collection of database objects, including tables, views, indexes, sequences, procedures, functions, and more. Each schema object represents a distinct element in the database, contributing to the overall structure and functionality of the database.

Here are some common types of schema objects:

1. Tables: Tables are used to store data in a structured format. They consist of rows and columns, and each column has a specific data type.

2. Views: Views are virtual tables created by querying one or more tables. They provide an abstracted or transformed representation of the data without physically storing it.

3. Indexes: Indexes improve the performance of data retrieval by providing a fast access path to data. They are created on columns of tables.

4. Sequences: Sequences are used to generate unique numeric values, often used as primary keys for tables.

5. Synonyms: Synonyms are aliases for schema objects. They provide a way to simplify the naming and referencing of objects.

6. Procedures: Procedures are stored sets of SQL statements that can be executed as a single unit. They are often used to perform specific actions or tasks.

7. Functions: Functions are similar to procedures but return a value. They are often used within SQL statements.

8. Triggers: Triggers are automatically executed when certain events occur in the database, such as data changes or specific times.

9. Packages: Packages are collections of related procedures, functions, and variables grouped together into a single unit.

10. Materialized Views: Materialized views are similar to regular views but store the results of the view query, improving query performance.

Schema objects are organized within a specific schema, which serves as a logical container for the objects. Each schema provides separation and security, allowing different users or applications to have their own isolated sets of objects. The database management system (DBMS) manages these schema objects, and users interact with them through SQL queries and other operations.

VIEWS:

A view in a relational database is a virtual table that is based on the result of a query. It allows you to present the data from one or more tables in a structured manner, without storing the data physically. Views are often used to simplify complex queries, restrict access to certain columns, or provide a consistent interface to users while hiding the underlying complexity of the database schema.

Here's the **syntax** to create a view:

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

Let's illustrate with an example using the employee and department tables:

Suppose you want to create a view that shows the names of employees along with their department names:

```
CREATE VIEW EmployeeDepartmentView AS
SELECT e.first_name || ' ' || e.last_name AS employee_name, d.department_name
FROM employee e JOIN department d ON e.department_id = d.department_id;
```

In this example, the view "**EmployeeDepartmentView**" is created. It combines data from the "employee" and "department" tables to provide a clear representation of employee names and their corresponding department names. **This view can be queried just like a regular table:**

```
SELECT * FROM EmployeeDepartmentView;
```

By querying the view, you'll get the same result as if you had run the original SELECT query that defined the view.

Views offer several benefits, including:

1. Simplicity: Views can simplify complex queries by encapsulating them in a single, easy-to-use structure.

2. Data Security: Views can restrict access to specific columns, allowing users to see only the information they need.

3. Abstraction: Views hide the complexity of the database schema, making it easier for users to interact with the data.

4. Consistency: Views provide a consistent interface to users, even if the underlying schema changes.

5. Performance: Materialized views can improve query performance by pre-computing and storing the results.

Remember that while views provide a convenient way to access data, they don't store the data themselves. They reflect the data in the underlying tables, so any changes made to the base tables are immediately reflected in the view.

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

SIMPLE & COMPLEX VIEW:

Views in a relational database can be classified into two main types: simple views and complex views. Let's explore each type with examples:

1. Simple Views:
Simple views are based on a single underlying table and involve straightforward SELECT statements. They provide an easy way to present a subset of data from a table.

Example of a Simple View:
Suppose you have an "employee" table and you want to create a simple view that shows the names and salaries of employees with a salary greater than 5000:

```
CREATE VIEW HighSalaryEmployees AS
SELECT first_name, last_name, salary FROM employee WHERE salary > 5000;
```

Now you can query the view like this:

```
SELECT * FROM HighSalaryEmployees;
```

2. Complex Views:
Complex views involve multiple underlying tables and can include various operations such as JOINs, subqueries, and aggregate functions. They are used to combine and present data from different tables in a more meaningful way.

Example of a Complex View:
Let's consider an example where you have "employee" and "department" tables. You want to create a complex view that displays the names of employees along with their department names and locations:

```
CREATE VIEW EmployeeDetails AS
SELECT e.first_name || ' ' || e.last_name AS employee_name, d.department_name, l.city FROM employee e
JOIN department d ON e.department_id = d.department_id
JOIN location l ON d.location_id = l.location_id;
```

This view involves multiple tables and JOIN operations to gather information from different sources and present it in a unified format.

Now you can query the complex view:

```
SELECT * FROM EmployeeDetails;
```

In summary, simple views are based on a single table and involve straightforward SELECT statements, while complex views involve multiple tables and may include various operations to combine and present data from different sources. Both types of views offer benefits in terms of data abstraction, security, and simplifying complex queries.

Difference between Simple & Complex view:

Attribute	Simple View	Complex View
Based on	A single table.	Multiple tables.
Operations Used	Basic SELECT statements without JOINs.	JOINs, subqueries, aggregate functions, GROUP BY, and other advanced SQL operations.
Purpose	Display a subset or transformed representation of a table.	Combine and represent data from different tables or provide a summarized or derived representation.
Modifications Allowed	Most of the times, allows DML operations (like INSERT, UPDATE, DELETE) if they affect the underlying table without violating any constraints.	Often doesn't allow direct DML operations, because the view pulls data from multiple sources or has aggregate operations.
Complexity	Simple and straightforward.	Can be intricate due to multiple tables and operations.
Example	**SELECT first_name, last_name FROM employee WHERE status = 'Active';**	**SELECT e.first_name, d.department_name FROM employee e JOIN department d ON e.dept_id = d.dept_id;**

Note: The ability to perform DML operations on a view depends on the specifics of the view and the underlying database system. Some complex views might still allow certain DML operations, while some simple views might restrict them based on constraints.

View creation options:

Here's an explanation of the various view creation options:

1. OR REPLACE: This option is used to modify an existing view with the new definition. If the view already exists, it will be replaced by the new definition without needing to drop the existing view explicitly.

Example:

```
CREATE OR REPLACE VIEW HighSalaryEmployees AS
SELECT first_name, last_name, salary FROM employee WHERE salary > 5000;
```

2. FORCE: The FORCE option is used to create a view even if the base table or tables referenced in the view's definition do not exist. It allows you to create the view and potentially resolve the references later.

Example:

```
CREATE FORCE VIEW EmployeeDetails AS
SELECT e.first_name || ' ' || e.last_name AS employee_name, d.department_name, l.city FROM employee e
JOIN department d ON e.department_id = d.department_id
JOIN location l ON d.location_id = l.location_id;
```

3. ALIAS: An alias is an alternative name given to a view. It's useful for creating a simpler or more descriptive name for the view.

Example:

```
CREATE VIEW EmpDetails AS
SELECT emp_id, first_name, last_name FROM employee;
```

4. NO FORCE: This option is used to create a view only if the base table or tables referenced in the view's definition exist. If any base table does not exist, the view creation will fail.

Example:

```
CREATE NO FORCE VIEW EmployeeDetails AS
SELECT e.first_name || ' ' || e.last_name AS employee_name, d.department_name, l.city
FROM employee e
JOIN department d ON e.department_id = d.department_id
JOIN location l ON d.location_id = l.location_id;
```

5. SUBQUERY: The SUBQUERY option is used to create a view based on the result of a subquery. The view's definition is the result of the subquery.

Example:

```
CREATE VIEW RecentHires AS
SELECT emp_id, first_name, last_name
FROM employee WHERE hire_date >= (SELECT MAX(hire_date) FROM employee) - 365;
```

6. WITH CHECK OPTION: The WITH CHECK OPTION is used to restrict data modifications through a view. If this option is specified, rows inserted or updated through the view must meet the condition specified in the view's WHERE clause.

Example:

```
CREATE VIEW HighSalaryEmployees AS
SELECT first_name, last_name, salary
FROM employee
WHERE salary > 5000 WITH CHECK OPTION;
```

7. WITH READ ONLY: The WITH READ ONLY option is used to indicate that the view is read-only, and no DML operations (INSERT, UPDATE, DELETE) can be performed through the view.

Example:

```
CREATE VIEW EmpDetailsReadOnly AS SELECT emp_id, first_name, last_name
FROM employee WITH READ ONLY;
```

These view creation options provide additional flexibility and control over how views are created and used in your database.

Note: OR REPLACE, view can be altered

DML THROUGH VIEWS:

Cannot remove a row if view contains	Cannot add a row if view contains
▪ Group functions ▪ A GROUP BY clause ▪ DISTINCT keyword ▪ ROWNUM keyword	▪ Group functions ▪ A GROUP BY clause ▪ DISTINCT keyword ▪ ROWNUM keyword ▪ Columns defined by expressions ▪ NOT NULL columns in base table that are not selected by view

Queries:

```
CREATE VIEW EMP_VW
AS SELECT EMP_ID, HIRE_DATE, DEPARTMENT_ID FROM EMPLOYEE  WHERE SALARY>8000;
```

	EMP_ID	HIRE_DATE	DEPARTMENT_ID
1	104	13-JAN-22	15
2	105	04-FEB-22	16

DESCRIBE EMP_VW; (TO SEE THE STRUCTURE OF THE VIEW)

```
CREATE OR REPLACE VIEW EMP_VW1 (EMPLOYEE_ID, NAME, TOTAL_SALARY)
AS
SELECT EMP_ID, FIRST_NAME, SALARY FROM EMPLOYEE
WHERE SALARY>8000;
                                        WITH ALIAS [IN BRACKET ALL ARE ALIAS NAMES
```

SELECT * FROM EMP_VW1;

	EMPLOYEE_ID	NAME	TOTAL_SALARY
1	104	Ankit	60000
2	105	Nandini	50000

```
CREATE OR REPLACE VIEW DEPT_SUM_VW (DNAME, MINSAL, MAXSAL, AVGSAL)
AS
SELECT  D.DEPARTMENT_NAME, MIN(E.SALARY),MAX(E.SALARY),AVG(E.SALARY)
FROM EMPLOYEE E JOIN DEPARTMENT D
ON E.DEPARTMENT_ID=D.DEPARTMENT_ID GROUP BY D.DEPARTMENT_NAME;
                                                                [ COMPLEX VIEW]
```

SELECT * FROM DEPT_SUM_VW;

	DNAME	MINSAL	MAXSAL	AVGSAL
1	Sales	6000	6000	6000
2	Marketing	8000	8000	8000
3	Finance	5000	5000	5000
4	IT	(null)	(null)	(null)
5	DATA	5500	50000	27750
6	Human Resources	7000	60000	33500

ANKKIT KUMAR GUPPTA
DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

```
CREATE OR REPLACE VIEW EMP_VW1
AS
SELECT * FROM EMPLOYEE WHERE DEPARTMENT_ID=16
WITH CHECK OPTION CONSTRAINT EMP_VW_CHK;

[WITH CHECK OPTION]
```

```
SELECT * FROM EMP_VW1;
```

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16
2	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16

```
CREATE OR REPLACE VIEW EMP_VW1
AS
SELECT * FROM EMPLOYEE
WHERE DEPARTMENT_ID=16
WITH READ ONLY;
                                        [WITH READ ONLY OPTION]
```

```
SELECT * FROM EMP_VW1;
```

	EMP_ID	FIRST_NAME	LAST_NAME	E_MAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	105	Nandini	Gupta	ng160593@gmail.com	9044719651	04-FEB-22	J005	50000	0.5	1013	16
2	109	LAXMI	MODANWAL	LAXMI@example.com	9998887777	01-APR-22	J009	5500	0.12	1002	16

ANKKIT KUMAR GUPPTA

DATA ANALYST | PROMPT ENGINEER

WITHOUT DATA, YOU'RE JUST ANOTHER PERSON WITH AN OPINION

If you would like to learn more & stay updated, please follow me on LinkedIn