UNIT 1: INTRODUCTION TO RELATIONAL MODEL

Database Systems: Overview and Key Concepts

A **Database System** is a collection of data that is organized in a structured manner, managed by a **Database Management System (DBMS)**. The DBMS allows users to store, retrieve, update, and manage data efficiently. In addition to storing data, a database system often includes various functionalities, such as enforcing data integrity, providing security, and ensuring efficient access to data.

1. Purpose of Database Systems

The primary purpose of a **Database System** is to store, retrieve, and manage data in a way that allows users and applications to interact with large amounts of data in a consistent and efficient manner. The database system serves several key purposes:

Key Purposes of Database Systems:

1. **Efficient Data Storage and Retrieval**:
    - Database systems are designed to efficiently store large volumes of data and ensure that data can be accessed quickly when needed.
    - They provide optimized indexing and querying capabilities, ensuring fast access even with large datasets.
2. **Data Integrity**:
    - Ensuring the accuracy, consistency, and reliability of data is a core purpose of a database system.
    - **Data Integrity Constraints** such as **primary keys**, **foreign keys**, and **unique constraints** help maintain the integrity of data during operations like inserts, updates, and deletes.
3. **Data Security**:
    - Protecting data from unauthorized access or modifications is critical.
    - Database systems include mechanisms such as user authentication, role-based access control, and encryption to ensure that only authorized users can access and modify data.
4. **Data Independence**:

- A good database system provides **data independence**, meaning that users or applications do not need to worry about the physical storage details or how data is internally structured.
- **Logical Data Independence** allows changes to the logical schema without affecting the applications.
- **Physical Data Independence** allows changes to the physical storage without affecting the logical view of the data.

5. **Concurrency Control**:
   - Database systems allow multiple users or applications to access the database concurrently without conflicts.
   - The system manages **concurrency control** to ensure that transactions are executed in a way that preserves data consistency and correctness (e.g., through **locking** mechanisms and **transactions**).
6. **Transaction Management**:
   - Database systems ensure the **ACID** properties of transactions: **Atomicity**, **Consistency**, **Isolation**, and **Durability**.
   - This ensures that even in the case of system failures, data remains consistent and reliable, and transactions are either fully completed or fully rolled back.
7. **Backup and Recovery**:
   - Ensuring that data can be recovered in case of failure (hardware or software failure) is another important function of a database system.
   - Databases include mechanisms for **automatic backups** and **recovery procedures** to restore data to a consistent state after a failure.
8. **Support for Complex Queries**:
   - Database systems enable users to perform complex queries using **Structured Query Language (SQL)** or other querying languages.
   - They support filtering, sorting, aggregating, and joining of data from multiple tables, making them powerful tools for retrieving meaningful insights from data.
9. **Multi-User Access**:
   - Database systems allow multiple users to interact with the same database simultaneously.
   - They include mechanisms to handle conflicts between different users, ensuring that concurrent access does not lead to inconsistencies.

2. View of Data

The **View of Data** in a database system refers to the way the data is presented to different users or applications. A database can have several **views** that provide different perspectives of the data, depending on the needs of the user or the application.

Types of Data Views:

1. **Physical View**:
    - This is the lowest-level view, representing how data is stored on the hardware (disk, memory, etc.). It deals with data storage and retrieval mechanisms such as **indexes**, **files**, and **disk blocks**.
    - The physical view is managed by the DBMS, and end-users or application developers are typically not concerned with it.
2. **Logical View**:
    - The logical view refers to the way data is organized and structured at a higher level (usually in terms of **tables**, **schemas**, and **relationships** between data).
    - This is the view that is most commonly used by users and applications. It defines what data is available and how it is related but not how it is stored physically.
    - For example, a **database schema** defines the structure of data, including tables, columns, data types, constraints, and relationships.
3. **External View (User View)**:
    - The external view represents how data is viewed by specific users or user groups. Different users may see different views of the data, tailored to their needs.
    - Each **user view** may be a subset of the **logical view**, containing only the data that is relevant to that user or group.
    - For example, a sales manager may have a view of sales data, while a human resources manager may have a different view showing employee data.

Data Independence and Views:

One of the key features of a database system is **data independence**, which allows changes to the physical or logical structure of the data without affecting the views presented to users.

- **Logical Data Independence**: Changes to the logical schema (such as adding new fields or tables) do not affect the external views. This allows users to continue working without disruption even if the structure of the data changes.
- **Physical Data Independence**: Changes to how the data is stored (e.g., moving data to different disk locations) do not affect the logical structure or user views. This ensures that users can work with data without worrying about its physical organization.

Example of Views in Action:

Consider a university database:

- **Logical View**: The logical schema could contain tables for **Students**, **Courses**, **Professors**, and **Enrollments**.
- **Physical View**: The data may be stored on disk in multiple files, and the system may optimize the layout to speed up queries.
- **External View**: A student may have a view that shows their **enrollment records**, while a professor may have a view showing only the **courses they teach**.

This separation of views makes databases more flexible and easier to manage. Users can access and interact with the data in the way that best suits their needs, without needing to understand or manage its physical organization.

3. Applications of Database Systems

Database systems are used in a wide variety of applications across different industries. Some common applications include:

1. **Enterprise Applications**:
    - Large businesses and organizations use database systems to manage core operations, such as **finance**, **inventory management**, **employee records**, and **customer data**.
2. **E-Commerce**:
    - Online shopping platforms use databases to track products, customers, orders, payments, and shipping information.
3. **Banking**:
    - Banks and financial institutions rely on database systems to manage accounts, transactions, loans, and customer data, ensuring security and consistency.
4. **Healthcare**:
    - Hospitals and medical systems store patient records, treatment histories, prescriptions, and scheduling information in databases.
5. **Social Media**:
    - Social networks like Facebook, Twitter, and Instagram rely on databases to store user profiles, posts, comments, and connections between users.
6. **Government**:
    - Government agencies use database systems for managing things like voter registration, tax records, and social security information.
7. **Education**:
    - Educational institutions use databases to store student information, course registrations, grades, and class schedules.
8. **Scientific Research**:
    - Research organizations store large datasets, experiment results, and research findings in databases for easy retrieval and analysis.
9. **Telecommunications**:

- Telecom companies use databases to manage customer information, billing, call records, and network usage.

Database Languages

Database languages are used to interact with a Database Management System (DBMS) for querying, updating, and managing data stored in databases. There are different types of languages in a DBMS, and each serves a specific purpose. The major database languages include:

1. **Data Definition Language (DDL)**:
   - DDL is used to define and manage the structure of the database, including tables, views, indexes, and other objects.
   - Common DDL commands include:
     - **CREATE**: Used to create database objects such as tables, indexes, and views.
     - **ALTER**: Used to modify existing database structures (e.g., adding columns to a table).
     - **DROP**: Used to delete database objects like tables or views.

   Example:

   ```
   CREATE TABLE Employees (
      EmployeeID INT PRIMARY KEY,
      FirstName VARCHAR(50),
      LastName VARCHAR(50),
      Age INT
   );
   ```

2. **Data Manipulation Language (DML)**:
   - DML is used to query and manipulate data stored in the database.
   - DML commands include:
     - **SELECT**: Used to retrieve data from the database.
     - **INSERT**: Used to insert data into a table.
     - **UPDATE**: Used to modify existing data in the table.

        - **DELETE**: Used to remove data from the database.

    Example:

    SELECT * FROM Employees WHERE Age > 30;

3. **Data Control Language (DCL)**:
    - DCL is used to control access to data in the database.
    - Common DCL commands include:
        - **GRANT**: Gives users privileges on database objects.
        - **REVOKE**: Removes privileges from users.

    Example:

    GRANT SELECT, INSERT ON Employees TO User1;

4. **Data Query Language (DQL)**:
    - DQL is used to query the database and retrieve data, and **SELECT** is the primary DQL command.
    - It allows users to retrieve data according to specified criteria.

    Example:

    SELECT * FROM Products WHERE Price > 50;

These languages form the core of SQL (Structured Query Language), which is the standard for querying and managing relational databases.

Relational Databases

A **relational database** is a type of database that stores data in tables, which are composed of rows and columns. Each table is related to others via common attributes or keys, allowing users to link data across multiple tables efficiently. The relational model, proposed by **Edgar Codd** in the 1970s, is based on mathematical set theory, where data is represented as relations (tables) and manipulated using set operations.

Key Concepts in Relational Databases:

1. **Tables (Relations)**:
   - A table in a relational database is a collection of related data entries. It is organized in rows and columns.
   - Each **column** in a table represents an attribute, and each **row** represents a record (tuple) of data.

2. **Primary Key**:
   - A **primary key** is a column (or set of columns) that uniquely identifies each row in a table. Every table should have a primary key to ensure that there are no duplicate rows.

   Example: In the Employees table, the EmployeeID could be the primary key since it uniquely identifies each employee.

3. **Foreign Key**:
   - A **foreign key** is a column (or set of columns) that creates a link between two tables by referencing the primary key of another table.
   - Foreign keys help maintain **referential integrity**, ensuring that relationships between tables are valid.

   Example: In an Orders table, a foreign key might reference the EmployeeID from the Employees table to indicate which employee processed the order.

4. **Normalization**:
   - **Normalization** is the process of organizing data in the database to reduce redundancy and improve data integrity.
   - This process involves dividing a large table into smaller, related tables and defining relationships between them.
   - Common **normal forms** (NF) include 1NF (First Normal Form), 2NF (Second Normal Form), and 3NF (Third Normal Form).
5. **Relationships**:
   - Tables in a relational database are related through keys. The relationships can be:
     - **One-to-One**: Each row in Table A is related to only one row in Table B.
     - **One-to-Many**: Each row in Table A is related to multiple rows in Table B.

        - **Many-to-Many**: Multiple rows in Table A are related to multiple rows in Table B. This relationship typically requires a **junction table**.

6. **SQL Queries**:
    - SQL queries are used to interact with relational databases. Users can perform operations like selecting data, inserting records, updating, or deleting data from tables.

Database Design

Database design is the process of creating a detailed blueprint of the database structure. It involves defining the relationships between tables, ensuring data integrity, and creating an efficient design that supports the needs of users and applications.

Key Steps in Database Design:

1. **Requirements Gathering**:
    - The first step in database design is gathering requirements to understand the data that will be stored and how it will be used. This involves talking to stakeholders, such as business analysts, users, and developers.
2. **Conceptual Design**:
    - The conceptual design defines the overall structure of the database using **Entity-Relationship (ER) diagrams**.
    - Entities represent objects or concepts (e.g., Employees, Orders), and relationships represent associations between entities (e.g., an employee processes an order).
3. **Logical Design**:
    - The logical design translates the conceptual design into a logical schema, which includes the tables, columns, primary and foreign keys, and relationships.
    - This step focuses on ensuring that the database is normalized and that relationships between entities are correctly established.
4. **Normalization**:

- **Normalization** is used to eliminate redundancy and ensure that the database design adheres to certain rules (normal forms) to maintain data integrity.

5. **Physical Design**:
   - The physical design involves translating the logical design into a physical storage structure. This includes decisions on how data will be stored, indexed, and accessed.
   - The physical design is often influenced by factors like performance, storage requirements, and backup strategies.
6. **Implementation**:
   - After the design is finalized, the database is created using a DBMS. This step involves creating tables, defining relationships, and populating the database with initial data.

Database Overall Structure

The **overall structure of a database** includes several components that work together to ensure efficient data storage, retrieval, and management. A typical database system consists of:

1. **Database Schema**:
   - The schema defines the logical structure of the database, including the tables, columns, data types, and constraints. The schema is often defined using DDL statements in SQL.
   - The schema can also include views, indexes, and stored procedures.
2. **Tables**:
   - Tables are the core structure where data is stored. Each table has columns (attributes) and rows (records).
   - Tables are linked to each other through **keys** (primary keys, foreign keys).
3. **Indexes**:
   - **Indexes** are used to speed up data retrieval. They provide a fast lookup mechanism for finding data within large tables.
   - Indexes can be created on columns frequently used in **WHERE** clauses, **JOINs**, or as part of a **ORDER BY** operation.
4. **Views**:
   - **Views** are virtual tables that provide a specific representation of data from one or more tables. They do not store data themselves but rather store queries that retrieve data from underlying tables.

- Views are useful for simplifying complex queries or presenting data in a customized format for specific users.

5. **Stored Procedures**:
   - **Stored procedures** are precompiled SQL code that can be executed on the database. They allow for complex business logic to be executed on the server side.
   - Stored procedures can help with performance, maintainability, and security.
6. **Triggers**:
   - **Triggers** are automatic actions performed in response to certain events in the database, such as data insertion, updating, or deletion.
   - They can enforce business rules or maintain data integrity by automatically modifying other tables when changes are made.
7. **Transactions**:
   - A **transaction** is a sequence of one or more SQL operations that are treated as a single unit of work. Transactions ensure that all operations are completed successfully, or none are applied, preserving the **ACID properties** (Atomicity, Consistency, Isolation, Durability).

Structure of Relational Databases

A **relational database** is structured around the concept of **tables** (or relations), where data is stored in rows (tuples) and columns (attributes). Each table has a unique name and consists of columns with defined data types. These tables are related to each other through **keys**, primarily **primary keys** and **foreign keys**, which allow data to be connected across different tables.

Key Components of Relational Database Structure:

1. **Tables (Relations)**:
   - A table is a collection of related data entries. It consists of rows and columns. Each row in the table represents a single record, while each column represents an attribute of the record.
   - **Example**: A table Employee might have columns like EmployeeID, FirstName, LastName, Age, and Salary.
2. **Columns (Attributes)**:

    - Each column in a table represents an attribute or property of the entity described by the table. For example, in an Employee table, columns could represent attributes like EmployeeID, FirstName, and Salary.
3. **Rows (Tuples)**:
    - Each row represents a single record or instance of the entity. For example, in the Employee table, a row might represent one specific employee's data, such as EmployeeID = 1, FirstName = "John", LastName = "Doe", etc.
4. **Primary Key**:
    - A primary key is a unique identifier for each record in a table. It ensures that no two rows have the same value for the primary key column(s).
5. **Foreign Key**:
    - A foreign key is a column (or set of columns) in one table that refers to the primary key of another table, establishing a relationship between the two tables.
6. **Indexes**:
    - Indexes are used to speed up query processing by allowing faster search operations on tables. They are typically created on columns that are often queried or used for joins.
7. **Views**:
    - A view is a virtual table based on the result of a query. It does not store data itself but provides a way to represent data in a customized format.
8. **Schemas**:
    - A schema defines the structure of the database, including tables, columns, relationships, and constraints. It ensures that data is organized in a logical and efficient manner.

Fundamental Relational Algebra Operations

Relational Algebra is a formal system for manipulating relations (tables) in a database. It consists of a set of operations that take one or more relations as input and produce a new relation as output. These operations form the foundation for querying relational databases.

Basic Relational Algebra Operations:

1. **Select (σ)**:

- The **select** operation (denoted as σ) is used to retrieve rows from a relation that satisfy a given predicate or condition.
- **Syntax**:

  σ(condition)(R)

- **Example**: Retrieve employees with salary greater than 50,000.

  σ(Salary > 50000)(Employee)

2. **Project (π)**:
   - The **project** operation (denoted as π) is used to select specific columns from a relation.
   - **Syntax**:

     π(attribute1, attribute2, ...)(R)

   - **Example**: Retrieve only the FirstName and LastName columns from the Employee table.

     scss
     Copy
     π(FirstName, LastName)(Employee)

3. **Union (∪)**:
   - The **union** operation (denoted as ∪) combines two relations that have the same set of attributes, removing duplicates.
   - **Syntax**:

     R ∪ S

   - **Example**: Combine employees from two different departments.

     EmployeeDept1 ∪ EmployeeDept2

4. **Difference (−)**:
   - The **difference** operation (denoted as −) returns rows that are present in the first relation but not in the second.
   - **Syntax**:

$R - S$

- **Example**: Retrieve employees who are not in the Sales department.

  Employee – SalesDepartmentEmployees

5. **Cartesian Product (×)**:
   - The **cartesian product** operation (denoted as ×) combines every row of one relation with every row of another relation, resulting in a relation that contains all possible combinations of rows from both relations.
   - **Syntax**:

     $R \times S$

   - **Example**: Combine employee and department data (if there's no direct relationship between the two).

     Employee × Department

6. **Rename (ρ)**:
   - The **rename** operation (denoted as ρ) changes the name of a relation or its attributes.
   - **Syntax**:

     ρ(NewName)(R)

Additional & Extended Relational Algebra Operations

In addition to the basic operations, there are several extended relational algebra operations that provide more advanced ways to manipulate relations.

Extended Operations:

1. **Join (⋈)**:
   - The **join** operation combines rows from two relations based on a common attribute (usually a foreign key and primary key).
   - There are several types of joins:
     - **Theta Join**: Join based on a condition.

- **Equi Join**: Join based on equality between columns.
- **Natural Join**: Join based on common attributes.

Syntax (Theta Join):

R ⋈ condition S

Example: Retrieve employees and the departments they work in.

Employee ⋈ Employee.DepartmentID = Department.DepartmentID Department

2. **Intersection (∩)**:
   - The **intersection** operation (denoted as ∩) returns the common rows that are present in both relations.
   - **Syntax**:

     R ∩ S

   - **Example**: Retrieve employees who are in both Department1 and Department2.

     Department1 ∩ Department2

3. **Division (÷)**:
   - The **division** operation (denoted as ÷) is used to find rows in one relation that are related to all rows in another relation.
   - **Syntax**:

     R ÷ S

4. **Example**: Retrieve employees who work in all departments.

   Employee ÷ Department

Null Values in Relational Databases

Null values are used in relational databases to represent missing, unknown, or inapplicable data. In relational algebra and SQL, **Null** is not equivalent to zero or an empty string.

Important Points About Null Values:

1. **Null and Comparisons**:
   - In relational algebra, comparisons involving Null values (such as =, >, <, etc.) are undefined. A Null cannot be compared to another value directly.
   - For instance, Salary > 50000 where Salary is Null will return **undefined**, not true or false.
2. **Handling Nulls**:
   - Many operations (like **join** or **select**) must handle Null values carefully. A **Null** value is treated as unknown, and the results of queries involving Nulls may be affected.
   - **SQL** provides special predicates like IS NULL and IS NOT NULL to test for null values.
3. **Null and Aggregates**:
   - Most aggregate functions (e.g., **SUM**, **AVG**) ignore Null values. For example, AVG(Salary) will ignore rows where Salary is Null.

Modification of the Database

The process of modifying the database includes adding, updating, or deleting data. These operations are usually performed through **Data Manipulation Language (DML)** commands such as INSERT, UPDATE, and DELETE.

1. **Insert (INSERT INTO)**:
   - The INSERT INTO statement is used to add new records to a table.
   - **Example**:

     INSERT INTO Employee (EmployeeID, FirstName, LastName, Age, Salary)
     VALUES (101, 'Alice', 'Smith', 30, 70000);

2. **Update (UPDATE)**:
   - The UPDATE statement is used to modify existing records in a table.
   - **Example**:

     UPDATE Employee SET Salary = 75000 WHERE EmployeeID = 101;

3. **Delete (DELETE FROM)**:

- The DELETE statement is used to remove records from a table.
- **Example**:

  DELETE FROM Employee WHERE EmployeeID = 101;

Handling Constraints:

- When modifying the database, **integrity constraints** such as **foreign keys** and **unique keys** must be respected. If a modification violates a constraint, the operation will be rejected.

UNIT 2: STRUCTURE QUERY LANGUAGE

SQL: Data Definition and Structure of SQL Queries

SQL (Structured Query Language) is the standard language used to communicate with relational databases. It provides a variety of commands for defining, manipulating, and controlling data in relational databases. SQL is divided into different categories based on its functionality, such as **Data Definition Language (DDL)**, **Data Manipulation Language (DML)**, **Data Control Language (DCL)**, and **Data Query Language (DQL)**.

Data Definition Language (DDL)

Data Definition Language (DDL) is a subset of SQL used to define and manage database structures. DDL commands allow users to create, alter, and drop database objects such as tables, indexes, views, and schemas. The most commonly used DDL commands are **CREATE**, **ALTER**, and **DROP**.

1. CREATE

The **CREATE** statement is used to create database objects such as tables, schemas, and indexes.

- **CREATE TABLE**: Used to create a new table.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
```

```
    LastName VARCHAR(50),
    Age INT,
    Salary DECIMAL(10, 2)
);
```

- **CREATE INDEX**: Used to create an index on one or more columns of a table to speed up query processing.

  ```
  CREATE INDEX idx_salary ON Employees (Salary);
  ```

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

  ```
  CREATE VIEW EmployeeSalaries AS
  SELECT FirstName, LastName, Salary
  FROM Employees
  WHERE Salary > 50000;
  ```

2. ALTER

The **ALTER** statement is used to modify an existing database object, such as adding, deleting, or modifying columns in a table.

- **ALTER TABLE**: Used to modify the structure of an existing table.
  - **Add a column**:

    ```
    ALTER TABLE Employees
    ADD DateOfBirth DATE;
    ```

  - **Modify a column**:

    ```
    ALTER TABLE Employees
    MODIFY COLUMN Salary DECIMAL(12, 2);
    ```

  - **Drop a column**:

    ```
    ALTER TABLE Employees
    DROP COLUMN DateOfBirth;
    ```

3. DROP

The **DROP** statement is used to delete database objects like tables, views, and indexes.

- **DROP TABLE**: Used to delete a table and its data from the database.

  DROP TABLE Employees;

- **DROP VIEW**: Used to delete a view from the database.

  DROP VIEW EmployeeSalaries;

- **DROP INDEX**: Used to delete an index from a table.

  DROP INDEX idx_salary;

Structure of SQL Queries

An **SQL query** is a command that interacts with a relational database to retrieve, insert, update, or delete data. The general structure of an SQL query can vary depending on its purpose (e.g., SELECT, INSERT, UPDATE, DELETE), but they generally follow a similar syntax pattern.

1. SELECT Queries

A SELECT query is used to retrieve data from one or more tables.

- **Basic Structure of a SELECT Query**:

  ```
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition
  GROUP BY column
  HAVING condition
  ORDER BY column [ASC|DESC];
  ```

Key Components:

- **SELECT**: Specifies the columns to retrieve.
- **FROM**: Specifies the table(s) from which to retrieve data.

- **WHERE**: Filters rows based on a condition.
- **GROUP BY**: Groups rows sharing a property for aggregate functions (e.g., COUNT, SUM, AVG).
- **HAVING**: Filters groups based on a condition (only works with GROUP BY).
- **ORDER BY**: Sorts the result set based on one or more columns.

Example:

Retrieve employees with salaries greater than 50,000, grouped by department, and sorted by salary in descending order:

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
WHERE Salary > 50000
GROUP BY Department
HAVING AVG(Salary) > 60000
ORDER BY AvgSalary DESC;
```

2. INSERT INTO Queries

The INSERT INTO statement is used to add new rows to a table.

- **Basic Structure of INSERT INTO**:

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

Example:

Insert a new employee into the Employees table:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Age, Salary)
VALUES (101, 'John', 'Doe', 30, 60000);
```

3. UPDATE Queries

The UPDATE statement is used to modify existing data in a table.

- **Basic Structure of UPDATE**:

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

Example:

Update the salary of an employee with EmployeeID = 101:

```
UPDATE Employees
SET Salary = 65000
WHERE EmployeeID = 101;
```

4. DELETE Queries

The DELETE statement is used to remove rows from a table.

- **Basic Structure of DELETE**:

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

Example:

Delete an employee from the Employees table where the EmployeeID is 101:

```
DELETE FROM Employees
WHERE EmployeeID = 101;
```

5. ALTER TABLE Queries

The ALTER TABLE statement is used to modify an existing table structure (e.g., adding or dropping columns).

- **Basic Structure of ALTER TABLE**:

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

Example:

Add a new column DateOfJoining to the Employees table:

```
ALTER TABLE Employees
ADD DateOfJoining DATE;
```

6. JOIN Queries

JOIN operations are used to combine rows from two or more tables based on a related column. There are different types of joins such as **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, and **FULL OUTER JOIN**.

- **INNER JOIN**: Returns rows where there is a match in both tables.

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

- **LEFT JOIN**: Returns all rows from the left table, and the matched rows from the right table. If there is no match, NULL values are returned for the right table.

  sql
  Copy
  ```
  SELECT column1, column2, ...
  FROM table1
  LEFT JOIN table2
  ON table1.common_column = table2.common_column;
  ```

Example:

Retrieve all employees and their department names, even if they don't belong to any department:

```
SELECT Employees.EmployeeID, Employees.FirstName,
Departments.DepartmentName
FROM Employees
```

```
LEFT JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;
```

SQL Query Execution Order

When a SQL query is executed, the database engine processes the query in the following logical order:

1. **FROM**: Retrieves data from the tables.
2. **WHERE**: Filters the data based on the specified condition.
3. **GROUP BY**: Groups the data based on the columns specified.
4. **HAVING**: Filters groups based on the specified condition (after the GROUP BY).
5. **SELECT**: Determines the columns to be included in the final result.
6. **ORDER BY**: Sorts the result set by the specified columns.

1. Set Operations

Set operations are used to combine the results of two or more queries into a single result set. These operations assume that the queries return the same number of columns with compatible data types.

There are **four primary set operations** in SQL:

1. **UNION**:
   - The **UNION** operation combines the results of two queries and removes duplicate rows.
   - **Syntax**:

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

   - **Example**: Combine the list of employees from two different departments:

```
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Sales'
UNION
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Marketing';
```

2. **UNION ALL**:
   - Similar to **UNION**, but it does not remove duplicates.
   - **Syntax**:

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

   - **Example**: Combine the list of employees from two different departments, including duplicates:

```
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Sales'
UNION ALL
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Marketing';
```

3. **INTERSECT**:
   - The **INTERSECT** operation returns only the rows that are common to both queries.
   - **Syntax**:

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

   - **Example**: Find employees who are working in both Sales and Marketing:

```
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Sales'
```

```
INTERSECT
SELECT FirstName, LastName
FROM Employees WHERE Department = 'Marketing';
```

4. **EXCEPT** (or **MINUS** in some databases):
   - The **EXCEPT** operation returns rows from the first query that are not in the second query.
   - **Syntax**:

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

   - **Example**: Find employees who are in the Sales department but not in the Marketing department:

     ```
     SELECT FirstName, LastName
     FROM Employees WHERE Department = 'Sales'
     EXCEPT
     SELECT FirstName, LastName
     FROM Employees WHERE Department = 'Marketing';
     ```

2. Aggregate Functions

Aggregate functions are used to perform calculations on a set of values and return a single result. They are commonly used with the GROUP BY clause to summarize data.

Here are the most commonly used **aggregate functions** in SQL:

1. **COUNT()**:
   - Counts the number of rows in a result set.
   - **Example**:

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

2. **SUM()**:
   - Returns the sum of values in a specified column.
   - **Example**:

```
SELECT SUM(Salary) FROM Employees;
```

3. **AVG()**:
   - Returns the average value of a specified column.
   - **Example**:

```
SELECT AVG(Salary) FROM Employees;
```

4. **MIN()**:
   - Returns the minimum value in a specified column.
   - **Example**:

```
SELECT MIN(Salary) FROM Employees;
```

5. **MAX()**:
   - Returns the maximum value in a specified column.
   - **Example**:

```
SELECT MAX(Salary) FROM Employees;
```

Using Aggregate Functions with GROUP BY: When using aggregate functions, the data is often grouped into subsets using the GROUP BY clause.

- **Example**: Find the total salary by department:

```
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Department;
```

3. Nested Subqueries

A **nested subquery** is a query that is embedded within another query. It is often used in the **WHERE** or **FROM** clause to perform complex filtering or calculations.

Types of Subqueries:

1. **Single-row Subqueries**:
   - A subquery that returns a single row.
   - **Example**: Find employees with a salary greater than the average salary:

     ```
     SELECT FirstName, LastName, Salary
     FROM Employees
     WHERE Salary > (SELECT AVG(Salary) FROM Employees);
     ```

2. **Multi-row Subqueries**:
   - A subquery that returns multiple rows.
   - **Example**: Find employees who work in departments that have a salary greater than 50,000:

     ```
     SELECT FirstName, LastName
     FROM Employees
     WHERE DepartmentID IN (SELECT DepartmentID FROM
     Departments WHERE Salary > 50000);
     ```

3. **Correlated Subqueries**:
   - A subquery that references columns from the outer query.
   - **Example**: Find employees who earn more than the average salary in their department:

     ```
     SELECT FirstName, LastName, Salary, Department
     FROM Employees e
     WHERE Salary > (SELECT AVG(Salary) FROM Employees
     WHERE Department = e.Department);
     ```

4. Complex Queries

Complex queries involve multiple conditions, joins, subqueries, and aggregate functions to retrieve and manipulate data in a more sophisticated manner.

Example of a Complex Query:

Retrieve the average salary for each department, only for departments with more than 3 employees, and order by department name:

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department
HAVING COUNT(EmployeeID) > 3
ORDER BY Department;
```

Views

- A **view** is a virtual table based on the result of a SELECT query.
- Views do not store data themselves but are used to simplify complex queries, improve security by restricting access to specific columns or rows, and make the database structure easier to understand.

Creating a View:

```
CREATE VIEW EmployeeSalaries AS
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary > 50000;
```

Using a View:

Once a view is created, you can use it just like a regular table in queries.

```
SELECT * FROM EmployeeSalaries;
```

Updating a View:

Some views allow updates, but this depends on the SQL implementation and the complexity of the view. Generally, views based on multiple tables or aggregate functions are not directly updatable.

Dropping a View:

To remove a view, use the DROP VIEW command.

DROP VIEW EmployeeSalaries;

Database Modification

Database modification refers to operations that add, update, or delete data in the database. These operations are part of the **Data Manipulation Language (DML)** in SQL. Here are the key commands used for modifying the data:

1.1 INSERT INTO

The INSERT INTO statement is used to add new records to a table.

- **Syntax**:

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

- **Example**: Insert a new employee record into the Employees table:

  INSERT INTO Employees (EmployeeID, FirstName, LastName, Department, Salary)
  VALUES (102, 'Alice', 'Smith', 'HR', 50000);

1.2 UPDATE

The UPDATE statement is used to modify existing data in a table.

- **Syntax**:

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

- **Example**: Update the salary of an employee in the Employees table:

  ```
  UPDATE Employees
  SET Salary = 55000
  WHERE EmployeeID = 102;
  ```

1.3 DELETE

The DELETE statement is used to remove records from a table.

- **Syntax**:

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

- **Example**: Delete an employee with EmployeeID = 102 from the Employees table:

  ```
  DELETE FROM Employees
  WHERE EmployeeID = 102;
  ```

2. Joined Relations

Joined relations are used to retrieve data from two or more related tables. SQL provides several types of joins to combine data from multiple tables based on a related column. These joins allow you to create more meaningful and complex queries.

2.1 INNER JOIN

The **INNER JOIN** returns only the rows where there is a match in both tables.

- **Syntax**:

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

- **Example**: Get the list of employees and their department names:

```
SELECT Employees.FirstName, Employees.LastName,
Departments.DepartmentName
FROM Employees
INNER JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;
```

2.2 LEFT JOIN (LEFT OUTER JOIN)

The **LEFT JOIN** returns all rows from the left table and the matched rows from the right table. If there is no match, NULL is returned for columns from the right table.

- **Syntax**:

```
SELECT column1, column2, ...
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

- **Example**: Get all employees and their departments, including those employees who don’t belong to any department:

```
SELECT Employees.FirstName, Employees.LastName,
Departments.DepartmentName
FROM Employees
LEFT JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;
```

2.3 RIGHT JOIN (RIGHT OUTER JOIN)

The **RIGHT JOIN** returns all rows from the right table and the matched rows from the left table. If there is no match, NULL is returned for columns from the left table.

- **Syntax**:

  ```
  SELECT column1, column2, ...
  FROM table1
  RIGHT JOIN table2
  ON table1.common_column = table2.common_column;
  ```

2.4 FULL JOIN (FULL OUTER JOIN)

The **FULL JOIN** returns all rows from both tables. If there is no match, NULL is returned for columns from the missing table.

- **Syntax**:

  ```
  SELECT column1, column2, ...
  FROM table1
  FULL JOIN table2
  ON table1.common_column = table2.common_column;
  ```

3. SQL Data Types and Schemas

SQL uses **data types** to define the kind of data that can be stored in each column of a table. Additionally, **schemas** provide a way to organize and group database objects.

3.1 SQL Data Types

SQL offers a wide range of data types to store various types of data:

- **Numeric Data Types**:
  - **INT**: Stores integers.
  - **DECIMAL(p, s)** or **NUMERIC(p, s)**: Stores fixed-point numbers, where p is the precision and s is the scale (number of digits after the decimal point).
  - **FLOAT** or **REAL**: Stores floating-point numbers.
- **Character and String Data Types**:
  - **CHAR(n)**: Fixed-length string with a length of n.
  - **VARCHAR(n)**: Variable-length string with a maximum length of n.
  - **TEXT**: Stores long text data.
- **Date and Time Data Types**:
  - **DATE**: Stores date in the format YYYY-MM-DD.
  - **TIME**: Stores time in the format HH:MM:SS.
  - **DATETIME** or **TIMESTAMP**: Stores both date and time.
- **Boolean Data Type**:
  - **BOOLEAN**: Stores TRUE or FALSE.
- **Binary Data Types**:
  - **BLOB**: Stores binary data, such as images or files.
- **Other Data Types**:
  - **ENUM**: A string object with a list of predefined values.
  - **JSON**: Stores JSON data.

3.2 SQL Schema

A **schema** is a logical container for database objects such as tables, views, indexes, and procedures. It helps organize database objects into manageable groups. Schemas allow for better database management, especially in large databases.

- **Syntax for Creating a Schema**:

  CREATE SCHEMA schema_name;

- **Example**: Create a schema called HR and then create a table within that schema:

  CREATE SCHEMA HR;
  CREATE TABLE HR.Employees (

```
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Department VARCHAR(50)
);
```

4. Integrity Constraints

Integrity constraints are rules that help maintain the correctness and consistency of data in the database. They ensure that data is accurate, reliable, and conforms to specific rules.

4.1 Primary Key (PK)

A **primary key** uniquely identifies each row in a table. Each table can have only one primary key, and the primary key column(s) must contain unique values and cannot be NULL.

- **Syntax**:

```
CREATE TABLE table_name (
    column1 INT PRIMARY KEY,
    column2 VARCHAR(50)
);
```

4.2 Foreign Key (FK)

A **foreign key** is a column (or set of columns) that establishes a link between the data in two tables. It refers to the primary key in another table, ensuring referential integrity.

- **Syntax**:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    EmployeeID INT,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
```

```
);
```

4.3 Unique Constraint

The **unique constraint** ensures that all values in a column (or a combination of columns) are unique across the rows.

- **Syntax**:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100) UNIQUE
);
```

4.4 Not Null Constraint

The **NOT NULL constraint** ensures that a column cannot have a NULL value.

- **Syntax**:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL
);
```

4.5 Check Constraint

The **CHECK constraint** ensures that the values in a column meet a specified condition.

- **Syntax**:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Age INT CHECK (Age >= 18)
);
```

5. Authorization

Authorization refers to granting or restricting access to various database resources based on user roles and permissions. In SQL, this is typically done through the **GRANT** and **REVOKE** commands.

5.1 GRANT Command

The **GRANT** statement is used to give specific privileges to a user or role.

- **Syntax**:

  GRANT privilege_type ON object TO user;

- **Example**: Grant SELECT permission on the Employees table to a user:

  GRANT SELECT ON Employees TO user_name;

5.2 REVOKE Command

The **REVOKE** statement is used to remove specific privileges from a user or role.

- **Syntax**:

  REVOKE privilege_type ON object FROM user;

- **Example**: Revoke SELECT permission from a user:

  REVOKE SELECT ON Employees FROM user_name;

5.3 Roles

A **role** is a collection of privileges that can be assigned to users. Roles allow easier management of permissions for a group of users.

- **Syntax**:

  CREATE ROLE role_name;
  GRANT SELECT, INSERT ON Employees TO role_name;

ODBC (Open Database Connectivity)

ODBC is an open standard API (Application Programming Interface) for accessing database management systems (DBMS). It provides a common interface for

connecting to various databases, which is useful for applications that need to interact with multiple databases.

Key Features of ODBC:

- **Database Independence**: ODBC allows applications to access different databases (e.g., MySQL, SQL Server, Oracle) without having to change the application code.
- **SQL Query Execution**: Applications can execute SQL queries using ODBC functions.
- **Data Retrieval**: It can retrieve results from the database and return them to the application.

ODBC Architecture:

1. **ODBC Driver Manager**: This is responsible for managing communication between the application and the ODBC drivers.
2. **ODBC Drivers**: These are DBMS-specific drivers that communicate directly with the database and translate the application's requests into SQL statements.
3. **Application**: The program using ODBC to interact with the database.

Common ODBC Functions:

1. **SQLConnect()**: Establishes a connection to the database.
2. **SQLExecDirect()**: Executes an SQL statement directly on the database.
3. **SQLFetch()**: Retrieves rows from a result set.
4. **SQLCloseCursor()**: Closes a result set cursor.
5. **SQLDisconnect()**: Disconnects from the database.

Example of ODBC in C:

```
#include <sql.h>
#include <sqlext.h>

SQLHENV henv;
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLRETURN ret;

// Allocate environment
```

```
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER)SQL_OV_ODBC3, 0);

// Allocate connection handle
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

// Connect to the database
SQLConnect(hdbc, (SQLCHAR*)"DSN=MyDatabase", SQL_NTS, NULL, 0,
NULL, 0);

// Execute an SQL query
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
SQLExecDirect(hstmt, (SQLCHAR*)"SELECT * FROM Employees",
SQL_NTS);

// Fetch data
SQLFetch(hstmt);

// Close the connection
SQLDisconnect(hdbc);
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

2. JDBC (Java Database Connectivity)

JDBC is an API provided by Java to connect and execute queries with relational databases. It is similar to ODBC but specifically designed for Java applications. JDBC allows Java applications to interact with databases such as MySQL, Oracle, SQL Server, and more.

Key Features of JDBC:

- **Java-centric**: JDBC is designed to work with Java applications.
- **Database-agnostic**: Like ODBC, JDBC provides a standard way to access databases regardless of the underlying DBMS.
- **Connection Management**: JDBC provides methods for establishing a connection, managing transactions, and closing connections.

- **Exception Handling**: JDBC provides built-in exception handling for database-related errors.

JDBC Architecture:

1. **JDBC Drivers**: JDBC requires specific drivers for each database. These drivers handle communication between Java applications and DBMS.
2. **Connection Interface**: This interface is used to establish and manage a connection to the database.
3. **Statement Interface**: This interface is used to execute SQL queries.
4. **ResultSet Interface**: This interface represents the data retrieved from the database.

Common JDBC Methods:

1. **getConnection()**: Establishes a connection to the database.
2. **createStatement()**: Creates a statement to execute SQL queries.
3. **executeQuery()**: Executes a query and returns a result set.
4. **executeUpdate()**: Executes an SQL statement such as INSERT, UPDATE, or DELETE.
5. **close()**: Closes the connection to the database.

Example of JDBC in Java:

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      try {
         // Load the JDBC driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Establish connection
         Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"username", "password");

         // Create a statement
         Statement stmt = conn.createStatement();

         // Execute query
```

```
            ResultSet rs = stmt.executeQuery("SELECT * FROM Employees");

            // Iterate over the result set
            while (rs.next()) {
                int empId = rs.getInt("EmployeeID");
                String firstName = rs.getString("FirstName");
                String lastName = rs.getString("LastName");
                System.out.println(empId + ": " + firstName + " " + lastName);
            }

            // Close the connection
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Functions and Procedural Constructs in SQL

SQL also allows you to define **functions** and use **procedural constructs** to implement more complex business logic directly within the database. These are essential for encapsulating repetitive logic and performing advanced operations without needing to rely on application code.

3.1 Functions in SQL

A **function** in SQL is a stored routine that can take input parameters, execute a sequence of SQL statements, and return a value. Functions are typically used for computations, transformations, or aggregations.

- **Syntax**:

  ```
  CREATE FUNCTION function_name (parameters)
  RETURNS return_type
  AS
  BEGIN
    -- Function body
    RETURN value;
  END;
  ```

- **Example**: A function to calculate the area of a circle given its radius:

  ```
  CREATE FUNCTION CalculateArea (radius FLOAT)
  RETURNS FLOAT
  AS
  BEGIN
    RETURN 3.14159 * radius * radius;
  END;
  ```

- To call the function:

  ```
  SELECT CalculateArea(5);  -- Returns 78.53975
  ```

3.2 Stored Procedures

A **stored procedure** is similar to a function, but instead of returning a value, it may perform actions such as modifying data, handling business logic, or calling other SQL commands. Stored procedures are often used to encapsulate a series of operations that need to be executed as a batch.

- **Syntax**:

  ```
  CREATE PROCEDURE procedure_name (parameters)
  AS
  BEGIN
    -- Procedure body
  END;
  ```

- **Example**: A stored procedure to insert a new employee record:

  ```
  CREATE PROCEDURE InsertEmployee (
    IN empFirstName VARCHAR(50),
    IN empLastName VARCHAR(50),
    IN empSalary DECIMAL(10, 2)
  )
  AS
  BEGIN
    INSERT INTO Employees (FirstName, LastName, Salary)
    VALUES (empFirstName, empLastName, empSalary);
  END;
  ```

- To execute the stored procedure:

  CALL InsertEmployee('Alice', 'Smith', 60000);

3.3 Procedural Constructs in SQL

SQL supports various **procedural constructs** to control the flow of execution in functions and stored procedures. These constructs include:

1. **IF...ELSE**: Conditional execution based on a boolean expression.
   - **Syntax**:

     ```
     IF condition
     BEGIN
       -- Statements to execute if condition is true
     END
     ELSE
     BEGIN
       -- Statements to execute if condition is false
     END
     ```

2. **LOOP / WHILE**: Loops to repeat actions multiple times.
   - **Syntax**:

     ```
     WHILE condition
     BEGIN
       -- Statements to execute while condition is true
     END
     ```

3. **BEGIN...END**: Grouping multiple statements together into a block.
   - **Example**:

     ```
     BEGIN
       DECLARE @counter INT = 1;
       WHILE @counter <= 5
       BEGIN
         PRINT 'This is loop iteration ' + CAST(@counter AS VARCHAR);
         SET @counter = @counter + 1;
       END
     END
     ```

4. **DECLARE**: Used to declare variables within stored procedures or functions.
    - **Syntax**:

      DECLARE @variable_name datatype;