SQL refresher

Introduction

SQL (Structured Query Language) is a standardized programming language specifically designed for managing and manipulating databases. SQL is used to communicate with a database to perform various tasks like querying data, updating records, deleting data, and creating or modifying database structures (like tables).

SQL is the backbone of most database systems, including popular ones like MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. Its simplicity and powerful capabilities make it an essential tool for anyone working with data.

Here are some key points about SQL:

- **Declarative Language**: SQL is a declarative language, meaning you specify *what* you want to do, and the database management system (DBMS) decides *how* to do it.
- **Data Manipulation**: With SQL, you can retrieve, insert, update, and delete data in a database.
- **Data Definition**: SQL also allows you to define the structure of your data by creating or altering tables, views, and other database objects.

- **Data Control**: SQL provides commands to control access to data, ensuring only authorized users can perform certain actions.

Introduction to DBMS

A **Database Management System (DBMS)** is software that allows users to store, retrieve, update, and manage data in a structured and efficient manner. The primary purpose of a DBMS is to provide an organized way to handle large amounts of information and ensure data consistency, security, and accessibility.

Key functions of a DBMS include:

- **Data Storage**: The DBMS provides a structured way to store data in tables, rows, and columns.
- **Data Retrieval**: Users can retrieve data using queries (written in SQL) to get specific information from the database.
- **Data Integrity**: The DBMS ensures that the data remains accurate and consistent through rules and constraints.
- **Data Security**: The DBMS controls access to the data, ensuring that only authorized users can perform certain operations.
- **Data Backup and Recovery**: The DBMS provides tools to back up data and recover it in case of failures or errors.

DBMS simplifies data management by abstracting the complexities of storing and retrieving data, allowing users to focus on higher-level tasks like analyzing and making decisions based on that data.

SQL Keywords

SQL keywords are reserved words that have special meaning in SQL queries and commands. They are used to define the structure of SQL statements and specify the operations you want to perform on the database. Here are some common SQL keywords:

- `SELECT`: Retrieves data from a database.

  ```
  SELECT column1, column2 FROM table_name;
  ```

- `INSERT`: Adds new records to a table.

```
INSERT INTO table_name (column1, column2) VALUES (value
1, value2);
```

- `UPDATE` : Modifies existing records in a table.

```
UPDATE table_name SET column1 = value1 WHERE condition;
```

- `DELETE` : Removes records from a table.

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

- `CREATE` : Defines new database objects like tables or views.

```
CREATE TABLE table_name (column1 datatype, column2 datat
ype);
```

- `ALTER` : Changes the structure of an existing database object.

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

- `DROP` : Deletes database objects like tables or views.

```
DROP TABLE table_name;
```

- `WHERE` : Filters records based on a specified condition.

```
SELECT * FROM table_name WHERE condition;
```

- `JOIN` : Combines rows from two or more tables based on a related column.

```
SELECT * FROM table1 JOIN table2 ON table1.column = tabl
e2.column;
```

- `GROUP BY` : Groups rows sharing a property into summary rows.

```
SELECT column, COUNT(*) FROM table_name GROUP BY column;
```

- `ORDER BY`: Sorts the result set by one or more columns.

  ```
  SELECT * FROM table_name ORDER BY column ASC|DESC;
  ```

SQL Data Types

Data types define the kind of data that can be stored in a column of a table. Here are some common SQL data types:

- `INT`: Integer numbers.

  ```
  age INT;
  ```

- `VARCHAR(n)`: Variable-length string, where `n` specifies the maximum length.

  ```
  name VARCHAR(50);
  ```

- `CHAR(n)`: Fixed-length string, where `n` specifies the length.

  ```
  code CHAR(5);
  ```

- `TEXT`: Large text data.

  ```
  description TEXT;
  ```

- `DATE`: Date values.

  ```
  birth_date DATE;
  ```

- `DATETIME`: Date and time values.

  ```
  created_at DATETIME;
  ```

- `FLOAT`: Floating-point numbers.

  ```
  price FLOAT;
  ```

- `BOOLEAN`: Boolean values (true/false).

```
is_active BOOLEAN;
```

SQL Operators

SQL operators perform operations on data in SQL statements. Here are some common operators:

- **Arithmetic Operators**: Perform mathematical operations.
  - `+`: Addition
  - : Subtraction
  - : Multiplication
  - `/`: Division

  ```
  SELECT price * 1.1 AS new_price FROM products;
  ```

- **Comparison Operators**: Compare values and return a boolean result.
  - `=`: Equal to
  - `<>` or `!=`: Not equal to
  - `<`: Less than
  - `>`: Greater than
  - `<=`: Less than or equal to
  - `>=`: Greater than or equal to

  ```
  SELECT * FROM employees WHERE salary > 50000;
  ```

- **Logical Operators**: Combine multiple conditions.
  - `AND`: Both conditions must be true
  - `OR`: At least one condition must be true
  - `NOT`: Negates a condition

  ```
  SELECT * FROM orders WHERE status = 'shipped' AND quantity > 10;
  ```

- **IN**: Checks if a value is within a set of values.

  ```
  SELECT * FROM products WHERE category IN ('Electronics',
  'Furniture');
  ```

- **LIKE**: Searches for a specified pattern.

  ```
  SELECT * FROM customers WHERE name LIKE 'J%';
  ```

- **BETWEEN**: Checks if a value is within a range.

  ```
  SELECT * FROM orders WHERE order_date BETWEEN '2024-01-0
  1' AND '2024-12-31';
  ```

DDL (Data Definition Language) Functions in SQL

DDL functions are used to define and modify the structure of database objects such as tables, indexes, and schemas. Here are some key DDL commands:

1. `CREATE TABLE`

Purpose: Creates a new table in the database.

Syntax:

DDL (Data Definition Language) Functions in SQL

DDL functions are used to define and modify the structure of database objects such as tables, indexes, and schemas. Here are some key DDL commands:

1. `CREATE TABLE`

Purpose: Creates a new table in the database.

Syntax:

```
CREATE TABLE table_name (
    column1 datatype [constraint],
    column2 datatype [constraint],
```

```
    ...
);
```

Example:

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

This command creates a table named `employees` with columns for employee ID, first name, last name, and hire date.

2. `ALTER TABLE`

Purpose: Modifies an existing table's structure, such as adding, deleting, or modifying columns.

Syntax (Adding a Column):

```
ALTER TABLE table_name ADD column_name datatype [constrain
t];
```

Syntax (Dropping a Column):

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

Syntax (Modifying a Column):

```
ALTER TABLE table_name MODIFY column_name datatype [constra
int];
```

Example (Adding a Column):

```
ALTER TABLE employees ADD email VARCHAR(100);
```

Example (Dropping a Column):

```
ALTER TABLE employees DROP COLUMN email;
```

Example (Modifying a Column):

```
ALTER TABLE employees MODIFY last_name CHAR(100);
```

These commands add, remove, or change columns in the `employees` table.

3. `DROP TABLE`

Purpose: Deletes an entire table and all of its data from the database.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE employees;
```

This command removes the `employees` table from the database permanently.

4. `TRUNCATE TABLE`

Purpose: Removes all rows from a table but keeps the table structure for future use. It is usually faster than `DELETE` because it does not generate individual row delete operations.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE employees;
```

This command deletes all rows in the `employees` table but retains the table structure and its columns.

Key Differences Between `TRUNCATE TABLE` and `DELETE`

- `TRUNCATE TABLE`:
    - Removes all rows quickly and is less resource-intensive.
    - Does not generate individual row delete operations.
    - Cannot be rolled back in some databases.
    - Does not fire triggers.
- `DELETE`:
    - Removes rows one at a time and can be slower for large tables.
    - Each delete operation can be rolled back if a transaction is used.
    - Can fire triggers defined for the table.

DML (Data Manipulation Language) Commands in SQL

DML commands are used to manage and manipulate data within tables. They allow you to insert, update, delete, and retrieve data. Here's a breakdown of the key DML commands:

1. `INSERT`

Purpose: Adds new records to a table.

Syntax:

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

Example:

```
INSERT INTO employees (employee_id, first_name, last_name,
hire_date)
VALUES (1, 'John', 'Doe', '2024-08-12');
```

2. `UPDATE`

Purpose: Modifies existing records in a table.

Syntax:

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

Example:

```
UPDATE employees
SET last_name = 'Smith'
WHERE employee_id = 1;
```

3. `DELETE`

Purpose: Removes records from a table.

Syntax:

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

Example:

```
DELETE FROM employees
WHERE employee_id = 1;
```

Query Clauses for Data Retrieval

1. `JOIN`

Purpose: Combines rows from two or more tables based on a related column.

Types of Joins:

- `INNER JOIN`: Returns rows with matching values in both tables.

  ```
  SELECT employees.first_name, departments.department_name
  FROM employees
  INNER JOIN departments ON employees.department_id = depa
  rtments.department_id;
  ```

- `LEFT JOIN` (or `LEFT OUTER JOIN`): Returns all rows from the left table and matched rows from the right table.

  ```
  SELECT employees.first_name, departments.department_name
  FROM employees
  LEFT JOIN departments ON employees.department_id = depar
  tments.department_id;
  ```

- `RIGHT JOIN` (or `RIGHT OUTER JOIN`): Returns all rows from the right table and matched rows from the left table.

  ```
  SELECT employees.first_name, departments.department_name
  FROM employees
  RIGHT JOIN departments ON employees.department_id = depa
  rtments.department_id;
  ```

- `FULL JOIN` (or `FULL OUTER JOIN`): Returns all rows when there is a match in one of the tables.

  ```
  SELECT employees.first_name, departments.department_name
  FROM employees
  FULL JOIN departments ON employees.department_id = depar
  tments.department_id;
  ```

2. `GROUP BY`

Purpose: Groups rows that have the same values into summary rows.

Syntax:

```
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1;
```

Example:

```
SELECT department_id, COUNT(*)
FROM employees
```

```
GROUP BY department_id;
```

This query counts the number of employees in each department.

3. `ORDER BY`

Purpose: Sorts the result set by one or more columns.

Syntax:

```
SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC];
```

Example:

```
SELECT first_name, hire_date
FROM employees
ORDER BY hire_date DESC;
```

This query sorts employees by their hire date in descending order.

4. `HAVING`

Purpose: Filters groups based on a specified condition, used in conjunction with `GROUP BY`.

Syntax:

```
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1
HAVING COUNT(*) > value;
```

Example:

```
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 10;
```

This query shows departments with more than 10 employees.

Aggregate Queries in SQL

Aggregate queries use aggregate functions to perform calculations on multiple rows of data and return a single result. These functions are used in conjunction with the `GROUP BY` clause to group rows into summary rows and with the `HAVING` clause to filter those groups. Here's a look at some common aggregate functions and their usage:

Common Aggregate Functions

1. `COUNT()`
   - **Purpose:** Returns the number of rows that match a specified condition.
   - **Syntax:**

     ```
     SELECT COUNT(column_name) FROM table_name;
     ```

   - **Example:**
     This query returns the total number of employees.

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

2. `SUM()`
   - **Purpose:** Returns the total sum of a numeric column.
   - **Syntax:**

     ```
     SELECT SUM(column_name) FROM table_name;
     ```

   - **Example:**
     This query returns the total salary of all employees.

     ```
     SELECT SUM(salary) FROM employees;
     ```

3. `AVG()`
   - **Purpose:** Returns the average value of a numeric column.

- **Syntax:**

```
SELECT AVG(column_name) FROM table_name;
```

- **Example:**
  This query returns the average salary of employees.

```
SELECT AVG(salary) FROM employees;
```

4. `MIN()`

- **Purpose:** Returns the minimum value in a column.
- **Syntax:**

```
SELECT MIN(column_name) FROM table_name;
```

- **Example:**
  This query returns the lowest salary among employees.

```
SELECT MIN(salary) FROM employees;
```

5. `MAX()`

- **Purpose:** Returns the maximum value in a column.
- **Syntax:**

```
SELECT MAX(column_name) FROM table_name;
```

- **Example:**
  This query returns the highest salary among employees.

```
SELECT MAX(salary) FROM employees;
```

Using Aggregate Functions with `GROUP BY`

Aggregate functions are often used with the `GROUP BY` clause to calculate summary statistics for each group.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example:

```
SELECT department_id, AVG(salary) AS average_salary
FROM employees
GROUP BY department_id;
```

This query calculates the average salary for each department.

Filtering Groups with `HAVING`

The `HAVING` clause is used to filter the results of aggregate functions, similar to how the `WHERE` clause filters rows.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING aggregate_function(column2) condition;
```

Example:

```
SELECT department_id, COUNT(*) AS num_employees
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 10;
```

This query finds departments with more than 10 employees.

Constraints and Assertions

Constraints and **assertions** in SQL and database management systems are used to enforce rules and ensure the integrity of the data within the database. They help maintain data consistency and enforce business rules.

Constraints

Constraints are rules applied to database tables to enforce data integrity and consistency. They are defined when creating or altering tables and ensure that data adheres to specific requirements. Here are the main types of constraints:

1. **Primary Key Constraint**
    - Ensures that each row in a table has a unique identifier and that no null values are allowed in the primary key column(s).
    - **Syntax**:

    ```
    CREATE TABLE employees (
        employee_id INT PRIMARY KEY,
        employee_name VARCHAR(100)
    );
    ```

2. **Foreign Key Constraint**
    - Ensures that the value in a column (or a set of columns) matches the value in a column in another table, establishing a relationship between the two tables.
    - **Syntax**:

    ```
    CREATE TABLE orders (
        order_id INT PRIMARY KEY,
        employee_id INT,
        FOREIGN KEY (employee_id) REFERENCES employees(em
    ployee_id)
    );
    ```

3. **Unique Constraint**
    - Ensures that all values in a column (or a set of columns) are unique across the table.
    - **Syntax**:

    ```
    CREATE TABLE employees (
        employee_id INT PRIMARY KEY,
    ```

```
    email VARCHAR(100) UNIQUE
);
```

4. **Check Constraint**
   - Ensures that all values in a column satisfy a specific condition.
   - **Syntax**:

   ```
   CREATE TABLE employees (
       employee_id INT PRIMARY KEY,
       salary DECIMAL(10, 2),
       CHECK (salary > 0)
   );
   ```

5. **Not Null Constraint**
   - Ensures that a column cannot contain null values.
   - **Syntax**:

   ```
   CREATE TABLE employees (
       employee_id INT PRIMARY KEY,
       employee_name VARCHAR(100) NOT NULL
   );
   ```

6. **Default Constraint**
   - Provides a default value for a column when no value is specified during an insert operation.
   - **Syntax**:

   ```
   CREATE TABLE employees (
       employee_id INT PRIMARY KEY,
       hire_date DATE DEFAULT CURRENT_DATE
   );
   ```

Assertions

Assertions are a type of constraint that is more complex and used to enforce business rules at a higher level. Unlike column-level constraints, assertions can involve multiple tables and more complex conditions.

Characteristics of Assertions:

1. **Complex Conditions**: Assertions can involve complex conditions and multiple tables.
2. **Global Constraints**: They are applied at the database level and ensure that certain conditions hold true across the entire database.
3. **Declarative Syntax**: Assertions are usually declared using SQL's declarative syntax.

Syntax of Assertions:

In SQL, the creation and management of assertions are not supported in all RDBMS implementations directly as they are part of the SQL standard. For those databases that support them, the syntax can be:

```
CREATE ASSERTION assertion_name
CHECK (condition);
```

Example of an Assertion:

Suppose we want to ensure that the total salary paid to employees in a department does not exceed a certain limit:

```
CREATE ASSERTION max_salary_check
CHECK (
    NOT EXISTS (
        SELECT department_id
        FROM employees
        GROUP BY department_id
        HAVING SUM(salary) > 1000000
    )
);
```

This assertion checks that no department's total salary exceeds 1,000,000. If such a condition is met, the assertion is violated, and the database system will prevent the operation that causes the violation.

Constraints vs. Assertions

- **Scope**:
    - **Constraints**: Typically applied to columns or rows within a single table or across related tables.
    - **Assertions**: Applied across the entire database and can involve complex conditions and multiple tables.
- **Complexity**:
    - **Constraints**: Generally simpler and enforceable directly through SQL commands.
    - **Assertions**: More complex and are used for enforcing higher-level business rules.
- **Support**:
    - **Constraints**: Widely supported in most RDBMS implementations.
    - **Assertions**: Not always supported in all RDBMS implementations and may require custom triggers or other mechanisms for enforcement.

Views in SQL

A **view** in SQL is a virtual table that provides a way to present data from one or more tables. Unlike a regular table, a view does not store data itself but displays data dynamically based on a query defined when the view is created. Views are useful for simplifying complex queries, providing a layer of security, and presenting data in a specific format.

Creating Views

To create a view, you use the `CREATE VIEW` statement. This statement defines the view's name and the query that determines what data the view will display.

Syntax

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

Example

Create a view to display employee details from the `employees` table where the department is 'Sales':

```
CREATE VIEW SalesEmployees AS
SELECT employee_id, employee_name, salary
FROM employees
WHERE department = 'Sales';
```

Now, `SalesEmployees` can be queried just like a table:

```
SELECT * FROM SalesEmployees;
```

Modifying Views

To modify an existing view, you use the `CREATE OR REPLACE VIEW` statement. This statement allows you to redefine the view with a new query or change its structure.

Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE new_condition;
```

Example

Suppose we want to modify the `SalesEmployees` view to also include the hire date:

```
CREATE OR REPLACE VIEW SalesEmployees AS
SELECT employee_id, employee_name, salary, hire_date
FROM employees
WHERE department = 'Sales';
```

Dropping Views

To remove a view from the database, you use the `DROP VIEW` statement. This statement deletes the view definition but does not affect the underlying tables or data.

Syntax

```
DROP VIEW view_name;
```

Example

To drop the `SalesEmployees` view:

```
DROP VIEW SalesEmployees;
```

Additional Notes

1. **Views vs. Tables**:
   - **Tables**: Store data physically.
   - **Views**: Are virtual and do not store data but represent data from one or more tables.
2. **Updating Views**:
   - Some views are updatable, meaning you can perform `INSERT`, `UPDATE`, or `DELETE` operations on them if the view is based on a single table and meets certain criteria.
   - Not all views are updatable. Views that involve joins, aggregations, or complex queries may not support direct updates.
3. **Materialized Views**:
   - Some databases support materialized views, which store the result of the view query physically. These views need to be refreshed periodically to reflect the latest data.
4. **Security**:
   - Views can be used to restrict access to specific data by creating views that include only a subset of columns or rows from the underlying tables.

Joins

Sample Data

`employees` Table

employee_id	employee_name	department_id
1	Alice	101
2	Bob	102
3	Charlie	NULL
4	David	104

`departments` Table

department_id	department_name
101	HR
102	IT
103	Finance
104	Marketing

1. Inner Join

SQL Query

```
SELECT employees.employee_id, employees.employee_name, depa
rtments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

Result

employee_id	employee_name	department_name
1	Alice	HR
2	Bob	IT
4	David	Marketing

2. Left Join (or Left Outer Join)

SQL Query

```
SELECT employees.employee_id, employees.employee_name, depa
rtments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

Result

employee_id	employee_name	department_name
1	Alice	HR
2	Bob	IT
3	Charlie	NULL
4	David	Marketing

3. Right Join (or Right Outer Join)

SQL Query

```
SELECT employees.employee_id, employees.employee_name, depa
rtments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

Result

employee_id	employee_name	department_name
1	Alice	HR
2	Bob	IT
4	David	Marketing
NULL	NULL	Finance

4. Full Outer Join

SQL Query

```
SELECT employees.employee_id, employees.employee_name, depa
rtments.department_name
```

```
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.department_id;
```

Result

employee_id	employee_name	department_name
1	Alice	HR
2	Bob	IT
3	Charlie	NULL
4	David	Marketing
NULL	NULL	Finance

5. Self Join

SQL Query

Assuming there's a `manager_id` column in the `employees` table to indicate the manager:

```
SELECT e1.employee_id AS Employee1, e2.employee_id AS Emplo
yee2
FROM employees e1
INNER JOIN employees e2
ON e1.manager_id = e2.manager_id
AND e1.employee_id <> e2.employee_id;
```

Result

Employee1	Employee2
Alice	Bob

6. Cross Join

SQL Query

```
SELECT employees.employee_id, employees.employee_name, depa
rtments.department_name
```

```
FROM employees
CROSS JOIN departments;
```

Result

employee_id	employee_name	department_name
1	Alice	HR
1	Alice	IT
1	Alice	Finance
1	Alice	Marketing
2	Bob	HR
2	Bob	IT
2	Bob	Finance
2	Bob	Marketing
3	Charlie	HR
3	Charlie	IT
3	Charlie	Finance
3	Charlie	Marketing
4	David	HR
4	David	IT
4	David	Finance
4	David	Marketing

Advanced Function

Certainly! Here are the examples and results for each SQL advanced function:

1. String Functions

CONCAT

Example

```
SELECT CONCAT('Hello', ' ', 'World') AS greeting;
```

Result

greeting
Hello World

LENGTH

Example

```
SELECT LENGTH('Hello World') AS length;
```

Result

length
11

SUBSTRING

Example

```
SELECT SUBSTRING('Hello World' FROM 1 FOR 5) AS part;
```

Result

part
Hello

REPLACE

Example

```
SELECT REPLACE('Hello World', 'World', 'SQL') AS new_strin
g;
```

Result

new_string
Hello SQL

UPPER

Example

```
SELECT UPPER('Hello World') AS uppercase_string;
```

Result

uppercase_string
HELLO WORLD

LOWER

Example

```
SELECT LOWER('Hello World') AS lowercase_string;
```

Result

lowercase_string
hello world

---

2. Date and Time Functions

TIMESTAMP

Example

```
SELECT TIMESTAMP '2024-08-12 15:30:00' AS current_timestamp;
```

Result

current_timestamp
2024-08-12 15:30:00

DATEPART

Example

```
SELECT DATEPART(year, '2024-08-12') AS year;
```

Result

year
2024

DATEADD

Example

```
SELECT DATEADD(day, 10, '2024-08-12') AS new_date;
```

Result

new_date
2024-08-22

---

3. Mathematical Functions

FLOOR

Example

```
SELECT FLOOR(123.456) AS floored_value;
```

Result

floored_value
123

ABS

Example

```
SELECT ABS(-123.456) AS absolute_value;
```

Result

absolute_value
123.456

MOD

Example

```
SELECT MOD(10, 3) AS remainder;
```

Result

remainder
1

ROUND

Example

```
SELECT ROUND(123.4567, 2) AS rounded_value;
```

Result

rounded_value
123.46

CEILING

Example

```
SELECT CEILING(123.456) AS ceiling_value;
```

Result

ceiling_value
124

4. Conditional Functions

CASE

Example

```
SELECT employee_id,
       CASE
           WHEN department_id = 101 THEN 'HR'
           WHEN department_id = 102 THEN 'IT'
           ELSE 'Other'
       END AS department_name
FROM employees;
```

Result

employee_id	department_name
1	HR
2	IT
3	Other
4	Other

NULLIF

Example

```
SELECT NULLIF(100, 100) AS result;
```

Result

result
NULL

COALESCE

Example

```
SELECT COALESCE(NULL, NULL, 'default', 'value') AS result;
```

Result

result
default

Subqueries

Subqueries are queries nested inside another query and can be categorized into two main types: nested subqueries and correlated subqueries. Here's a detailed explanation of both:

1. Nested Subqueries

Definition: A nested subquery is a query embedded within another SQL query. The subquery is executed once, and its result is used by the outer query.

Characteristics:

- The subquery is executed independently of the outer query.
- It can be used in SELECT, WHERE, and HAVING clauses.
- Often used to retrieve a single value or a set of values that the outer query will use.

Example:

Consider a database with two tables: `employees` and `departments`.

Tables:

- **employees**: `employee_id`, `name`, `department_id`, `salary`
- **departments**: `department_id`, `department_name`

To find the names of employees who have the highest salary in their department:

```
SELECT name
FROM employees
WHERE salary = (
    SELECT MAX(salary)
    FROM employees e2
    WHERE e2.department_id = employees.department_id
);
```

Explanation:

- The inner query retrieves the maximum salary for each department.
- The outer query selects the names of employees whose salary matches this maximum.

2. Correlated Subqueries

Definition: A correlated subquery is a subquery that references columns from the outer query. It is executed repeatedly, once for each row processed by the outer query.

Characteristics:

- The subquery depends on the outer query for its values.
- Typically used in the WHERE clause of the outer query.
- Can be used to compare each row in the outer query to the results of the subquery.

Example:

Using the same `employees` and `departments` tables, to find employees whose salary is higher than the average salary of their department:

```
SELECT name
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e2.department_id = e1.department_id
);
```

Explanation:

- The inner query calculates the average salary for each department using `e1.department_id` from the outer query.
- The outer query retrieves the names of employees whose salary is greater than the average salary of their respective department.

Indexes

Indexes are critical components in database management systems (DBMS) that enhance the speed and efficiency of data retrieval operations. They work similarly to an index in a book, allowing the database to find specific rows of data quickly without scanning the entire table.

What is an Index?

An index is a database object that improves the speed of data retrieval operations on a database table. It maintains a sorted list of values from one or more columns of the table and helps quickly locate rows based on those values.

Types of Indexes

1. **Single-Column Index**:
    - **Definition**: An index created on a single column of a table.
    - **Usage**: Speeds up queries that filter or sort data based on this column.
    - **Example**: Creating an index on the `employee_id` column of the `employees` table.

    ```
    CREATE INDEX idx_employee_id ON employees (employee_id);
    ```

2. **Composite Index (Multi-Column Index)**:
    - **Definition**: An index created on multiple columns of a table.
    - **Usage**: Optimizes queries that filter or sort based on multiple columns.
    - **Example**: Creating an index on `department_id` and `salary` columns.

    ```
    CREATE INDEX idx_dept_salary ON employees (department_i
    d, salary);
    ```

3. **Unique Index**:
    - **Definition**: Ensures that the values in the indexed column(s) are unique across the table.
    - **Usage**: Enforces uniqueness for primary keys or other unique constraints.
    - **Example**: Creating a unique index on the `email` column to ensure no two employees have the same email address.

    ```
    CREATE UNIQUE INDEX idx_unique_email ON employees (emai
    l);
    ```

4. **Full-Text Index**:

- **Definition**: Used for searching large text data, allowing for efficient full-text searches.
- **Usage**: Improves performance for queries involving text searches or pattern matching.
- **Example**: Creating a full-text index on a `description` column.

```
CREATE FULLTEXT INDEX idx_fulltext_desc ON products (des
cription);
```

5. **Bitmap Index**:
   - **Definition**: Uses bitmap representations to quickly handle low-cardinality columns (columns with few distinct values).
   - **Usage**: Efficient for queries with multiple conditions on categorical data.
   - **Example**: Creating a bitmap index on a `gender` column with only 'M' and 'F' values.

```
CREATE BITMAP INDEX idx_gender ON employees (gender);
```

6. **Clustered Index**:
   - **Definition**: Determines the physical order of data in the table. Each table can have only one clustered index.
   - **Usage**: Optimizes range queries and sorting operations.
   - **Example**: Setting a clustered index on the `employee_id` column.

```
CREATE CLUSTERED INDEX idx_clustered_empid ON employees
(employee_id);
```

7. **Non-Clustered Index**:
   - **Definition**: A separate structure from the table that points to the rows in the table. A table can have multiple non-clustered indexes.
   - **Usage**: Improves performance for queries that use columns not covered by the clustered index.
   - **Example**: Creating a non-clustered index on the `last_name` column.

```
CREATE NONCLUSTERED INDEX idx_last_name ON employees (la
st_name);
```

Advantages of Indexes

1. **Speed**: Indexes significantly improve query performance by reducing the amount of data scanned.
2. **Efficiency**: They speed up search operations, sorting, and filtering, which can enhance overall query execution time.
3. **Optimization**: Helps with optimizing JOIN operations and WHERE clause conditions.

Disadvantages of Indexes

1. **Storage Overhead**: Indexes consume additional disk space.
2. **Maintenance Cost**: Indexes require maintenance, especially when data is inserted, updated, or deleted. This can impact write performance.
3. **Complexity**: Managing and optimizing indexes can add complexity to database administration.

Example

Consider a `customers` table with columns `customer_id`, `name`, and `email`. To improve the performance of queries that search by `email`, you can create an index:

```
CREATE INDEX idx_email ON customers (email);
```

With this index, queries like:

```
SELECT * FROM customers WHERE email = 'example@example.co
m';
```

will perform faster as the database can quickly locate the row based on the indexed `email` column, rather than scanning the entire table.

Managing Indexes

1. Creating Indexes

Purpose: To speed up data retrieval operations by creating a structured data path to quickly access rows.

Considerations:

- **Which Columns to Index**: Index columns that are frequently used in `WHERE` clauses, join conditions, or sorting.
- **Type of Index**: Choose between single-column, composite, unique, or other types based on query needs.

Example:

```
CREATE INDEX idx_customer_name ON customers (name);
```

2. Monitoring Index Usage

Purpose: To ensure indexes are being used effectively and efficiently.

Methods:

- **Database Performance Tools**: Use built-in tools to monitor index usage and performance metrics.
- **Query Execution Plans**: Analyze execution plans to check if indexes are being used.

Example (in MySQL):

```
EXPLAIN SELECT * FROM customers WHERE name = 'John Doe';
```

3. Rebuilding and Reorganizing Indexes

Purpose: To optimize performance by defragmenting indexes and improving their efficiency.

Methods:

- **Rebuild**: Drop and recreate the index to defragment it.
- **Reorganize**: Compact and reorganize the index without dropping it.

Example (in SQL Server):

```
-- Rebuild Index
ALTER INDEX idx_customer_name ON customers REBUILD;
```

```
-- Reorganize Index
ALTER INDEX idx_customer_name ON customers REORGANIZE;
```

4. Dropping Indexes

Purpose: To remove indexes that are no longer needed, freeing up resources and reducing overhead.

Example:

```
DROP INDEX idx_customer_name ON customers;
```

5. Index Maintenance

Purpose: Regular maintenance to ensure indexes remain effective over time.

Methods:

- **Monitoring Performance**: Regularly check the performance impact of indexes.
- **Updating Statistics**: Ensure that database statistics are up-to-date for the optimizer to make informed decisions.

Query Optimization

1. Understanding Query Execution Plans

Purpose: To analyze how the database executes a query and where optimizations can be made.

Example (in MySQL):

```
EXPLAIN SELECT * FROM orders WHERE order_date = '2024-01-0
1';
```

2. Using Indexes Effectively

Purpose: To ensure queries utilize indexes to avoid full table scans.

Methods:

- **Index Coverage**: Ensure indexes cover columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses.
- **Composite Indexes**: Use composite indexes for queries involving multiple columns.

3. Query Refactoring

Purpose: To rewrite queries for better performance.

Methods:

- *Avoid SELECT ***: Specify only the columns needed.
- **Use Proper Joins**: Use appropriate join types and ensure they are indexed.

Example:

```
-- Instead of
SELECT * FROM orders WHERE customer_id = 123;

-- Use
SELECT order_id, order_date FROM orders WHERE customer_id =
123;
```

4. Analyzing and Optimizing Joins

Purpose: To ensure joins are efficient.

Methods:

- **Index Join Columns**: Index columns used in joins.
- **Check Join Order**: Analyze the order of joins for optimal performance.

5. Optimizing Subqueries

Purpose: To ensure subqueries do not degrade performance.

Methods:

- **Use Joins Instead of Subqueries**: Where applicable, use joins instead of subqueries for better performance.

Example:

```
-- Subquery
SELECT name FROM customers WHERE id IN (SELECT customer_id
FROM orders WHERE amount > 100);

-- Join
SELECT c.name FROM customers c JOIN orders o ON c.id = o.cu
stomer_id WHERE o.amount > 100;
```

6. Managing Query Caching

Purpose: To reduce query execution time by caching query results.

Methods:

- **Enable Query Cache**: In some databases, enable and configure query caching settings.

Example (in MySQL):

```
-- Check if query cache is enabled
SHOW VARIABLES LIKE 'query_cache_type';

-- Enable query cache
SET GLOBAL query_cache_size = 1048576;
```

7. Using Database Tools

Purpose: To leverage built-in tools for query optimization.

Methods:

- **Database Optimizer**: Use built-in database optimizers and tuning advisors to suggest improvements.

Example (in SQL Server):

```
-- Use Database Engine Tuning Advisor
EXEC sp_db_tuningadvisor @database_id = DB_ID('your_databas
e');
```

Transactions

SQL transactions are essential for ensuring data integrity and consistency in database operations. They allow multiple operations to be executed as a single unit, ensuring that either all operations are completed successfully or none are, thus preserving the integrity of the database. Here's an overview of SQL transactions, including the concepts of `BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, ACID properties, and transaction isolation levels.

1. SQL Transactions

Definition: A transaction is a sequence of one or more SQL operations executed as a single unit. Transactions ensure that the database remains in a consistent state, even in the event of errors or system failures.

BEGIN

Purpose: Marks the start of a transaction.

Example:

```
BEGIN;
```

COMMIT

Purpose: Finalizes a transaction, making all changes made during the transaction permanent.

Example:

```
COMMIT;
```

Explanation: After a `COMMIT`, all changes made during the transaction are saved to the database, and the transaction is completed successfully.

ROLLBACK

Purpose: Undoes all changes made during the transaction, reverting the database to its previous state.

Example:

```
ROLLBACK;
```

Explanation: After a `ROLLBACK`, any changes made during the transaction are discarded, and the database returns to the state it was in before the transaction began.

SAVEPOINT

Purpose: Sets a point within a transaction to which you can later roll back, allowing partial rollback within a transaction.

Example:

```
SAVEPOINT savepoint_name;
```

To Rollback to a Savepoint:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

Explanation: `SAVEPOINT` creates a named point in the transaction. If a rollback is needed, you can roll back to this savepoint without affecting the entire transaction.

2. ACID Properties

ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These properties ensure reliable processing of database transactions.

- **Atomicity**: Ensures that all operations within a transaction are completed successfully; if not, the transaction is aborted and the database is left unchanged.
- **Consistency**: Guarantees that a transaction brings the database from one valid state to another, maintaining database integrity.
- **Isolation**: Ensures that transactions are executed independently of one another, meaning the operations of one transaction are not visible to others until the transaction is complete.
- **Durability**: Ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures.

3. Transaction Isolation Levels

Isolation levels define the degree to which the operations in one transaction are isolated from those in other transactions. SQL provides several isolation levels, each balancing between consistency and performance:

- **Read Uncommitted**: Allows transactions to read data from uncommitted changes made by other transactions. This is the lowest level of isolation and can lead to dirty reads.

  Example:

  ```
  SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
  ```

- **Read Committed**: Ensures that a transaction can only read data that has been committed by other transactions. This prevents dirty reads but allows non-repeatable reads.

  Example:

  ```
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  ```

- **Repeatable Read**: Ensures that if a transaction reads a value, it will see the same value if it reads it again, preventing dirty reads and non-repeatable reads. However, it can still be affected by phantom reads.

  Example:

  ```
  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  ```

- **Serializable**: The highest level of isolation, where transactions are executed in a way that ensures complete isolation from other transactions. It prevents dirty reads, non-repeatable reads, and phantom reads but can have performance overhead.

  Example:

  ```
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

Data Integrity and Security

Data Integrity

Data Integrity refers to the accuracy and consistency of data within a database. It is maintained through constraints and rules that enforce the correctness and validity of the data.

1. Data Integrity Constraints

- **Primary Key Constraint**:
  - **Purpose**: Ensures that each row in a table has a unique identifier.
  - **Example**:

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(50)
);
```

- **Foreign Key Constraint**:
  - **Purpose**: Maintains referential integrity between tables by ensuring that a value in one table corresponds to a valid value in another table.
  - **Example**:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    emp_id INT,
    FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
);
```

- **Unique Constraint**:
  - **Purpose**: Ensures that all values in a column or a set of columns are unique.
  - **Example**:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    email VARCHAR(255) UNIQUE
);
```

- **Check Constraint**:
  - **Purpose**: Validates the data in a column against a specified condition.
  - **Example**:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    price DECIMAL(10, 2),
    CHECK (price > 0)
);
```

- **Not Null Constraint**:
    - **Purpose**: Ensures that a column cannot have a NULL value.
    - **Example**:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);
```

Data Security

Data Security focuses on protecting data from unauthorized access and ensuring it is only accessible to those with appropriate permissions.

1. GRANT and REVOKE

- **GRANT**:
    - **Purpose**: Provides specific privileges to users or roles.
    - **Example**:

```
-- Grant SELECT and INSERT permissions on the employe
es table to user 'john'
GRANT SELECT, INSERT ON employees TO 'john';
```

- **REVOKE**:
    - **Purpose**: Removes specific privileges from users or roles.
    - **Example**:

```
-- Revoke INSERT permission on the employees table fr
om user 'john'
REVOKE INSERT ON employees FROM 'john';
```

Database Security Best Practices

- **Authentication and Authorization**
    - **Authentication**: Verify the identity of users accessing the database.
        - **Use strong passwords** and enforce password policies.
        - **Implement multi-factor authentication** (MFA) where possible.
    - **Authorization**: Grant users appropriate permissions based on their roles.
        - **Follow the principle of least privilege**: Grant only the minimum necessary permissions.
        - **Use roles and groups** to manage permissions more efficiently.
- **Data Encryption**
    - **Encrypt Data at Rest**: Protect stored data from unauthorized access.
        - **Example**: Use database features or third-party tools to encrypt database files.
    - **Encrypt Data in Transit**: Protect data as it moves between clients and the server.
        - **Example**: Use SSL/TLS to encrypt connections.
- **Regular Backups**
    - **Create Regular Backups**: Ensure data can be restored in case of loss or corruption.
    - **Example**: Schedule automated backups and store them securely.
- **Patch Management**
    - **Keep Software Updated**: Apply patches and updates to fix vulnerabilities and improve security.
    - **Example**: Regularly update database software and related systems.
- **Monitoring and Auditing**
    - **Monitor Access and Usage**: Track database access and changes to detect suspicious activity.
    - **Example**: Use database auditing features to log and review user actions.
- **Database Configuration**

    - **Secure Database Configuration**: Adjust settings to enhance security.
    - **Example**: Disable unused features and services, and configure proper access controls.
- **SQL Injection Prevention**
    - **Use Parameterized Queries**: Prevent SQL injection attacks by using prepared statements.
    - **Example**:

      ```
      -- Using a parameterized query in an application
      PREPARE stmt FROM 'SELECT * FROM users WHERE email =
      ?';
      EXECUTE stmt USING @email;
      ```
- **Access Controls**
    - **Implement Strong Access Controls**: Control who can access the database and what actions they can perform.
    - **Example**: Set up network firewalls and IP restrictions to limit access to the database server.
- **Data Masking**
    - **Use Data Masking**: Hide sensitive data in non-production environments.
    - **Example**: Mask customer data when creating test databases.

Functions and Procedures

Stored Procedures

Stored Procedures are a set of SQL statements that are stored in the database and can be executed as a single unit. They allow for modular programming by encapsulating logic into reusable procedures.

Key Features of Stored Procedures

1. **Encapsulation**: Stored procedures allow encapsulation of SQL queries and logic, making them reusable and easier to manage.

2. **Performance**: They can improve performance by reducing the amount of information sent between applications and the database and by allowing the database to optimize execution plans.
3. **Security**: Stored procedures help enhance security by controlling access to the data. Users can be granted permission to execute a stored procedure without needing direct access to the underlying tables.
4. **Error Handling**: They can include error handling mechanisms, which allows for more robust and controlled execution.

Example of a Stored Procedure

Creating a Stored Procedure:

```
CREATE PROCEDURE GetEmployeeDetails (IN emp_id INT)
BEGIN
    SELECT * FROM employees WHERE id = emp_id;
END;
```

Executing the Stored Procedure:

```
CALL GetEmployeeDetails(101);
```

Modifying a Stored Procedure:

```
ALTER PROCEDURE GetEmployeeDetails (IN emp_id INT)
BEGIN
    SELECT name, position FROM employees WHERE id = emp_id;
END;
```

Dropping a Stored Procedure:

```
DROP PROCEDURE GetEmployeeDetails;
```

Functions

Functions are similar to stored procedures but are designed to return a single value. They are used to perform calculations or transformations and return a result.

Key Features of Functions

1. **Return Value**: Functions return a single value, which can be used in SQL queries or expressions.
2. **Usage in Queries**: Functions can be used directly in SQL queries, SELECT statements, or as part of expressions.
3. **Encapsulation**: Like stored procedures, functions encapsulate logic for reuse.
4. **Side Effects**: Functions should generally avoid making changes to the database (e.g., inserting or updating records).

Example of a Function

Creating a Function:

```
CREATE FUNCTION CalculateDiscount (price DECIMAL(10,2)) RET
URNS DECIMAL(10,2)
BEGIN
    RETURN price * 0.1; -- Example discount calculation
END;
```

Using the Function in a Query:

```
SELECT product_name, CalculateDiscount(price) AS discount
FROM products;
```

Modifying a Function:

```
ALTER FUNCTION CalculateDiscount (price DECIMAL(10,2)) RETU
RNS DECIMAL(10,2)
BEGIN
    RETURN price * 0.15; -- Adjust discount rate
END;
```

Dropping a Function:

```
DROP FUNCTION CalculateDiscount;
```

Comparison

- **Stored Procedures**:
    - Can perform complex operations, including modifying data.
    - Can have multiple statements.
    - Do not necessarily return a value but can return result sets.
- **Functions**:
    - Return a single value.
    - Typically used in SQL queries.
    - Are designed to perform calculations or transformations.

Performance Optimizations

1. Query Analysis Techniques

1.1. Indexes

Indexes are database objects that speed up the retrieval of rows by providing quick access to data based on the values in one or more columns.

- **Purpose**: Indexes improve query performance by reducing the amount of data the database engine needs to scan.
- **Types of Indexes**:
    - **Single-Column Index**: An index on a single column.
    - **Composite Index**: An index on multiple columns.
    - **Unique Index**: Ensures that the values in the indexed column(s) are unique.

Example:

```
-- Creating an index on the 'email' column of the 'users' t
able
CREATE INDEX idx_users_email ON users(email);
```

Usage:
Indexes are automatically used by the database optimizer for SELECT

statements. However, they can slow down INSERT, UPDATE, and DELETE operations due to the overhead of maintaining the index.

1.2. Optimizing Joins

Joins combine rows from two or more tables based on a related column. Proper optimization of joins is crucial for performance.

- **Use Appropriate Join Types**:
    - **INNER JOIN**: Returns rows where there is a match in both tables.
    - **LEFT JOIN**: Returns all rows from the left table and matched rows from the right table.
    - **RIGHT JOIN**: Returns all rows from the right table and matched rows from the left table.
    - **FULL OUTER JOIN**: Returns rows with matches in either table.
    - **CROSS JOIN**: Returns the Cartesian product of both tables.
- **Optimize Join Conditions**: Ensure that join conditions use indexed columns.

Example:

```
-- Optimizing an INNER JOIN by ensuring 'employee_id' is in
dexed
SELECT e.name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_
id;
```

Tip: Avoid using complex joins with multiple tables unless necessary, as they can significantly impact performance.

1.3. Reducing Subqueries

Subqueries are queries nested inside other queries. While subqueries can be useful, they can sometimes be less efficient than joins.

- **Avoid Correlated Subqueries**: Correlated subqueries are evaluated for each row in the outer query, which can be slow.
- **Use Joins Instead**: Often, using joins can be more efficient than subqueries.

Example:

```
-- Replacing a correlated subquery with a JOIN
-- Original with subquery:
SELECT name
FROM employees
WHERE department_id IN (SELECT department_id FROM departmen
ts WHERE location = 'New York');

-- Optimized with JOIN:
SELECT e.name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_
id
WHERE d.location = 'New York';
```

1.4. Selective Projection

Selective Projection involves selecting only the columns needed for the query, rather than all columns.

- **Purpose**: Reducing the number of columns retrieved minimizes the amount of data processed and transferred.

Example:

```
-- Select only necessary columns
SELECT name, salary
FROM employees;
```

- *Avoid SELECT ***: Selecting all columns can result in fetching more data than required, impacting performance.

2. Additional Optimization Techniques

2.1. Query Execution Plans

Query Execution Plans provide insight into how the database engine executes a query. Analyzing execution plans helps identify bottlenecks and areas for improvement.

- **Use Database Tools**: Most database systems offer tools to view and analyze execution plans (e.g., `EXPLAIN` in MySQL or `EXPLAIN ANALYZE` in PostgreSQL).

2.2. Partitioning

Partitioning divides large tables into smaller, more manageable pieces while maintaining them as a single logical table.

- **Types of Partitioning**:
    - **Range Partitioning**: Based on a range of values (e.g., date ranges).
    - **List Partitioning**: Based on a list of values.
    - **Hash Partitioning**: Based on a hash function.

Example:

```
-- Creating range partitions based on the 'order_date'
CREATE TABLE orders (
    order_id INT,
    order_date DATE,
    ...
) PARTITION BY RANGE (order_date) (
    PARTITION p0 VALUES LESS THAN (2021-01-01),
    PARTITION p1 VALUES LESS THAN (2022-01-01),
    ...
);
```

2.3. Caching

Caching stores the results of frequent queries in memory to reduce the need to execute the same query multiple times.

- **Database Caching**: Many databases offer built-in caching mechanisms.
- **Application-Level Caching**: Use caching mechanisms in your application to store results.

Triggers in SQL

Triggers are special types of stored procedures in SQL that automatically execute or "fire" in response to certain events on a table or view. They are used to maintain data integrity, enforce business rules, or automate tasks within the database.

Types of Triggers

1. `BEFORE` **Trigger**

    - **Purpose:** Executes before an `INSERT`, `UPDATE`, or `DELETE` operation on a table.
    - **Use Case:** Validate or modify data before it is written to the database.

    Syntax:

    ```
    CREATE TRIGGER trigger_name
    BEFORE INSERT | UPDATE | DELETE ON table_name
    FOR EACH ROW
    BEGIN
        -- trigger logic
    END;
    ```

    Example:

    ```
    CREATE TRIGGER check_salary
    BEFORE INSERT ON employees
    FOR EACH ROW
    BEGIN
        IF NEW.salary < 0 THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Sala
    ry cannot be negative';
        END IF;
    END;
    ```

    This trigger prevents insertion of employees with a negative salary.

2. `AFTER` **Trigger**

    - **Purpose:** Executes after an `INSERT`, `UPDATE`, or `DELETE` operation on a table.

- **Use Case:** Update other tables or logs after a change has been committed.

Syntax:

```
CREATE TRIGGER trigger_name
AFTER INSERT | UPDATE | DELETE ON table_name
FOR EACH ROW
BEGIN
    -- trigger logic
END;
```

Example:

```
CREATE TRIGGER log_salary_change
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_audit (employee_id, old_salary, n
ew_salary, change_date)
    VALUES (OLD.employee_id, OLD.salary, NEW.salary, NOW
());
END;
```

This trigger logs salary changes in an audit table after an update.

3. `INSTEAD OF` **Trigger**

- **Purpose:** Executes in place of an `INSERT`, `UPDATE`, or `DELETE` operation on a table.
- **Use Case:** Customize or redirect data manipulation operations.

Syntax:

```
CREATE TRIGGER trigger_name
INSTEAD OF INSERT | UPDATE | DELETE ON table_name
FOR EACH ROW
BEGIN
    -- trigger logic
END;
```

Example:

```
CREATE TRIGGER replace_insert
INSTEAD OF INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employees_archive (employee_id, first_na
me, last_name, hire_date)
    VALUES (NEW.employee_id, NEW.first_name, NEW.last_na
me, NEW.hire_date);
END;
```

This trigger inserts data into an archive table instead of the main employees table.

Components of a Trigger

- **Trigger Name:** A unique name for the trigger within the database.
- **Trigger Event:** The database event that activates the trigger (e.g., `INSERT`, `UPDATE`, `DELETE`).
- **Trigger Timing:** Specifies whether the trigger should run `BEFORE` or `AFTER` the event.
- **Trigger Action:** The code that executes in response to the event, typically written in SQL.

Benefits of Using Triggers

- **Data Integrity:** Ensure data follows certain rules and constraints.
- **Automated Actions:** Automatically perform operations in response to changes in data.
- **Auditing and Logging:** Track changes and maintain logs for auditing purposes.
- **Complex Business Rules:** Enforce complex business rules directly in the database.

Considerations

- **Performance Impact:** Triggers can affect database performance, especially if they execute complex logic or are triggered frequently.
- **Debugging:** Triggers can make debugging more challenging, as they operate automatically and their effects may not be immediately visible.
- **Recursive Triggers:** Be cautious with triggers that might call other triggers, leading to recursive actions and potential performance issues.