Joins with Aggregate Functions

Joining tables with aggregate functions in SQL allows you to combine data from multiple tables while performing calculations such as SUM, COUNT, AVG, etc., on the grouped data.

```
SELECT t1.column1, AGGREGATE_FUNCTION(t2.column2)
FROM table1 AS t1
JOIN table2 AS t2 ON t1.related_column =
t2.related_column
GROUP BY t1.column1;
```

Explanation:

- Use the JOIN clause to combine rows from two tables based on a related column.
- Apply aggregate functions (e.g., SUM, COUNT, AVG) to columns from one or both tables to perform calculations on grouped data.
- Group the results by one or more columns using the GROUP BY clause to define the grouping criteria.

Example:

Consider two tables named orders and order_items.

You want to calculate the total quantity of items for each order:

```
SELECT orders.order_id, SUM(order_items.quantity) AS
total_quantity
FROM orders
JOIN order_items ON orders.order_id =
order_items.order_id
GROUP BY orders.order_id;
```

Output: The output will display the order ID and the total quantity of items for each order, calculated using the SUM aggregate function.

```
| order_id | total_quantity |
|----------|----------------|
| 1        | 15             |
| 2        | 10             |
| 3        | 25             |
```

Combining Queries

Using UNION

In SQL, the UNION operator is used to combine the results of two or more SELECT statements into a single result set. The SELECT statements must have the same number of columns and compatible data types in corresponding positions.

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

Explanation:

- Use the UNION keyword to combine the results of multiple SELECT statements.
- Each SELECT statement must have the same number of columns, and the corresponding columns must have compatible data types.
- The results of the UNION operation are distinct rows from the combined result sets of the SELECT statements.
- To include duplicate rows in the result set, you can use UNION ALL instead of UNION.

Example:

Consider two tables named employees and customers.

You want to retrieve the names of employees and customers:

```
SELECT employee_name AS name
FROM employees
UNION
SELECT customer_name AS name
FROM customers;
```

Output:

```
| name           |
|----------------|
| John Doe       |
| Jane Smith     |
| Alice Johnson  |
| Bob Williams   |
```

Using UNION ALL

In SQL, the UNION ALL operator is used to combine the results of two or more SELECT statements into a single result set, including all rows from each SELECT statement. Unlike the UNION operator, UNION ALL does not remove duplicate rows from the result set.

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

Explanation:

- Use the UNION ALL keyword to combine the results of multiple SELECT statements, including all rows from each SELECT statement.
- Each SELECT statement must have the same number of columns, and the corresponding columns must have compatible data types.

Example:

Consider two tables named students1 and students2.

You want to retrieve the names of students from both tables, including duplicate names:

```sql
SELECT student_name AS name
FROM students1
UNION ALL
SELECT student_name AS name
FROM students2;
```

Output:

```
| name         |
|--------------|
| John Doe     |
| Jane Smith   |
| Alice Johnson |
| Bob Williams |
| John Doe     |
| Sarah Brown  |
```

Sorting Combined Query Results

After combining the results of multiple queries using UNION or UNION ALL, you can apply sorting to the combined result set using the ORDER BY clause.

```
SELECT column1, column2, ...
FROM table1
UNION [ALL]
SELECT column1, column2, ...
FROM table2
ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;
```

Explanation:

- Use the ORDER BY clause to specify the columns by which you want to sort the combined result set.
- You can specify multiple columns for sorting, separated by commas.
- Optionally, you can specify the sorting direction for each column: ASC (ascending) or DESC (descending). ASC is the default sorting direction if not specified.

Example:

Consider two tables named students1 and students2. You want to retrieve the names of all students from both tables and sort them alphabetically:

```
SELECT student_name AS name
FROM students1
UNION ALL
SELECT student_name AS name
FROM students2
ORDER BY name ASC;
```

Output:

```
| name           |
|----------------|
| Alice Johnson |
| Bob Williams   |
| Jane Smith     |
| John Doe       |
| John Doe       |
| Sarah Brown    |
```

Inserting Data

Inserting complete rows

When inserting data into a table in SQL, you can specify complete rows to be inserted using the INSERT INTO statement.

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

Explanation:

- Use the INSERT INTO statement followed by the name of the table into which you want to insert data.
- Specify the columns into which you want to insert data, followed by the VALUES keyword.
- Provide the corresponding values for each column in the same order as the columns specified.
- Ensure that the data types of the values match the data types of the columns.

Example:

Consider a table named students with columns student_id, student_name, and age.

You want to insert a new row with the values for all columns:

```sql
INSERT INTO students (student_id, student_name, age)
VALUES (1, 'John Doe', 20);
```

Inserting complete rows 2

When inserting data into a table in SQL, you can specify complete rows to be inserted using the INSERT INTO statement.

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

Explanation:

- Use the INSERT INTO statement followed by the name of the table into which you want to insert data.
- Specify the columns into which you want to insert data, followed by the VALUES keyword.
- Provide the corresponding values for each column in the same order as the columns specified.
- Ensure that the data types of the values match the data types of the columns.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to insert a new row with the values for all columns:

```sql
INSERT INTO employees (employee_id, employee_name,
salary)
VALUES (101, 'Alice Johnson', 50000);
```

This statement will insert a new row into the employees table with the values (101, 'Alice Johnson', 50000) for the employee_id, employee_name, and salary columns, respectively.

Inserting complete rows 3

When inserting data into a table in SQL, you can specify complete rows to be inserted using the INSERT INTO statement.

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

Explanation:

- Use the INSERT INTO statement followed by the name of the table into which you want to insert data.
- Specify the columns into which you want to insert data, followed by the VALUES keyword.
- Provide the corresponding values for each column in the same order as the columns specified.
- Ensure that the data types of the values match the data types of the columns.

Example:

Consider a table named products with columns product_id, product_name, and price.

You want to insert a new row with the values for all columns:

```sql
INSERT INTO products (product_id, product_name, price)
VALUES (101, 'Keyboard', 25.99);
```

This statement will insert a new row into the products table with the values (101, 'Keyboard', 25.99) for the product_id, product_name, and price columns, respectively.

Inserting partial rows

When inserting data into a table in SQL, you can specify partial rows by omitting some columns from the INSERT INTO statement.

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

Explanation:

- Use the INSERT INTO statement followed by the name of the table into which you want to insert data.
- Specify the columns into which you want to insert data, followed by the VALUES keyword.
- Provide the corresponding values for each column in the same order as the columns specified.
- You can omit some columns from the INSERT INTO statement, and the database will use default values or NULL for those columns if they are nullable.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to insert a new row with only the employee_name and salary columns:

```sql
INSERT INTO employees (employee_name, salary)
VALUES ('Alice Johnson', 50000);
```

This statement will insert a new row into the employees table with the values 'Alice Johnson' for the employee_name column and 50000 for the salary column.

The employee_id column may have a default value or be auto-generated by the database.

Inserting retrieved data 1

In SQL, you can insert data retrieved from another table into a target table using the INSERT INTO ... SELECT statement. This allows you to copy data from one table to another or insert specific rows based on conditions.

```
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
[WHERE condition];
```

Explanation:

- Use the INSERT INTO statement followed by the name of the target table into which you want to insert data.
- Specify the columns into which you want to insert data.
- Use the SELECT statement to retrieve data from a source table.
- Optionally, you can include a WHERE clause to specify conditions for selecting rows from the source table.

Example:

Consider a table named students with columns student_id, student_name, and age, and another table named new_students with the same structure.

You want to insert all students from the new_students table into the students table:

```
INSERT INTO students (student_id, student_name, age)
SELECT student_id, student_name, age
FROM new_students;
```

This statement will insert all rows from the new_students table into the students table, copying the values of the student_id, student_name, and age columns.

Inserting retrieved data 2

In SQL, you can insert data retrieved from another table into a target table using the INSERT INTO ... SELECT statement. This allows you to copy data from one table to another or insert specific rows based on conditions.

```sql
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
[WHERE condition];
```

Explanation:

Use the INSERT INTO statement followed by the name of the target table into which you want to insert data.

Specify the columns into which you want to insert data.

Use the SELECT statement to retrieve data from a source table.

Optionally, you can include a WHERE clause to specify conditions for selecting rows from the source table.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary, and another table named new_employees with the same structure.

You want to insert all employees from the new_employees table into the employees table who have a salary greater than **$50,000**:

```sql
INSERT INTO employees (employee_id, employee_name,
salary)
SELECT employee_id, employee_name, salary
FROM new_employees
WHERE salary > 50000;
```

This statement will insert all rows from the new_employees table into the employees table where the salary is greater than $50,000, copying the values of the employee_id, employee_name, and salary columns.

Inserting retrieved data 3

In SQL, you can insert data retrieved from another table into a target table using the INSERT INTO ... SELECT statement. This allows you to copy data from one table to another or insert specific rows based on conditions.

```
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
[WHERE condition];
```

Explanation:

- Use the INSERT INTO statement followed by the name of the target table into which you want to insert data.
- Specify the columns into which you want to insert data.
- Use the SELECT statement to retrieve data from a source table.
- Optionally, you can include a WHERE clause to specify conditions for selecting rows from the source table.

Example:

Consider a table named customers with columns customer_id, customer_name, and city, and another table named new_customers with the same structure.

You want to insert all customers from the new_customers table into the customers table:

```
INSERT INTO customers (customer_id, customer_name, city)
SELECT customer_id, customer_name, city
FROM new_customers;
```

This statement will insert all rows from the new_customers table into the customers table, copying the values of the customer_id, customer_name, and city columns.

Creating a copy of the table 1

In SQL, you can create a copy of an existing table, including its structure and data, using the CREATE TABLE ... AS SELECT statement.

```
CREATE TABLE new_table AS
SELECT *
FROM existing_table;
```

Explanation:

- Use the CREATE TABLE statement followed by the name of the new table you want to create.
- Specify the AS keyword followed by a SELECT statement that retrieves data from the existing table.
- The new table will have the same structure as the existing table, including column names and data types, and will contain the data retrieved by the SELECT statement.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to create a copy of this table named employees_copy:

```
CREATE TABLE employees_copy AS
SELECT *
FROM employees;
```

This statement will create a new table named employees_copy with the same structure and data as the employees table. It will contain all columns (employee_id, employee_name, and salary) and all rows from the employees table.

Creating a copy of the table 2

In SQL, you can create a copy of an existing table, including its structure and data, using the CREATE TABLE ... AS SELECT statement.

```sql
CREATE TABLE new_table AS
SELECT *
FROM existing_table;
```

Explanation:

- Use the CREATE TABLE statement followed by the name of the new table you want to create.
- Specify the AS keyword followed by a SELECT statement that retrieves data from the existing table.
- The new table will have the same structure as the existing table, including column names and data types, and will contain the data retrieved by the SELECT statement.

Example:

Consider a table named products with columns product_id, product_name, and price. You want to create a copy of this table named products_copy:

```sql
CREATE TABLE products_copy AS
SELECT *
FROM products;
```

This statement will create a new table named products_copy with the same structure and data as the products table. It will contain all columns (product_id, product_name, and price) and all rows from the products table.

Updating and Deleting Data

Updating Single Column

In SQL, you can update existing data in a table using the UPDATE statement.

```
UPDATE table_name
SET column_name = new_value
[WHERE condition];
```

Explanation:

- Use the UPDATE statement followed by the name of the table you want to update.
- Specify the column you want to update followed by the SET keyword and the new value you want to assign to that column.
- Optionally, use the WHERE clause to specify conditions that must be met for the update to occur. If omitted, all rows in the table will be updated.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to update the salary of an employee with employee_id 101:

```
UPDATE employees
SET salary = 60000
WHERE employee_id = 101;
```

This statement will update the salary column of the employees table to 60000 for the employee with employee_id equal to 101.

Updating Multiple Columns

In SQL, you can update existing data in a table using the UPDATE statement.

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

Explanation:

- Use the UPDATE statement followed by the name of the table you want to update.
- Specify each column you want to update followed by the SET keyword and the new value you want to assign to that column.
- Optionally, use the WHERE clause to specify conditions that must be met for the update to occur. If omitted, all rows in the table will be updated.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to update both the employee_name and salary columns for an employee with employee_id 101:

```
UPDATE employees
SET employee_name = 'Alice Johnson', salary = 65000
WHERE employee_id = 101;
```

This statement will update the employee_name column to 'Alice Johnson' and the salary column to 65000 for the employee with employee_id equal to 101.

Deleting Column Value

In SQL, you can delete specific column values from existing rows using the UPDATE statement.

```
UPDATE table_name
SET column_name = NULL
[WHERE condition];
```

Explanation:

- Use the UPDATE statement followed by the name of the table you want to update.
- Specify the column you want to delete the value from followed by the SET keyword and set it to NULL.
- Optionally, use the WHERE clause to specify conditions that must be met for the update to occur. If omitted, all rows in the table will be affected.

Example:

Consider a table named employees with columns employee_id, employee_name, and department.

You want to delete the department value for an employee with employee_id 101:

```
UPDATE employees
SET department = NULL
WHERE employee_id = 101;
```

This statement will delete the department value for the employee with employee_id equal to 101, setting it to NULL.

Using Subqueries in Update

In SQL, you can use subqueries within an UPDATE statement to perform updates based on the result of a subquery.

```
UPDATE table_name
SET column_name = (SELECT expression FROM subquery)
[WHERE condition];
```

Explanation:

- Use the UPDATE statement followed by the name of the table you want to update.
- Specify the column you want to update followed by the SET keyword.
- Use a subquery inside parentheses to retrieve the new value for the column.
- Optionally, use the WHERE clause to specify conditions that must be met for the update to occur.

Example:

Consider a table named employees with columns employee_id, employee_name, and department_id, and another table named departments with columns department_id and department_name.

You want to update the employee_name column in the employees table with the department name where each employee works:

```
UPDATE employees
SET employee_name = (SELECT department_name FROM
departments WHERE departments.department_id =
employees.department_id);
```

This statement will update the employee_name column in the employees table with the corresponding department_name where each employee works, based on the department_id relationship between the two tables.

Updating All Rows

In SQL, you can update all rows in a table without specifying a condition by using the UPDATE statement without a WHERE clause.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
```

Explanation:

- Use the UPDATE statement followed by the name of the table you want to update.
- Specify each column you want to update followed by the SET keyword and the new value you want to assign to that column.
- Omit the WHERE clause, which means the update will be applied to all rows in the table.

Example:

Consider a table named employees with columns employee_id, employee_name, and salary.

You want to give all employees a salary increase of 10%:

```
UPDATE employees
SET salary = salary * 1.10;
```

This statement will update the salary column for all rows in the employees table, increasing each employee's salary by 10%.

Deleting Specific Rows

In SQL, you can delete specific rows from a table using the DELETE statement.

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

Explanation:

- Use the DELETE FROM statement followed by the name of the table from which you want to delete rows.
- Use the WHERE clause to specify conditions that identify the rows you want to delete. Only rows that meet the specified condition will be deleted. If you omit the WHERE clause, all rows in the table will be deleted.

Example:

Consider a table named employees with columns employee_id, employee_name, and department.

You want to delete the rows for employees who are no longer with the company:

```sql
DELETE FROM employees
WHERE employment_status = 'terminated';
```

This statement will delete all rows from the employees table where the employment_status is set to 'terminated', effectively removing the records of terminated employees from the table.

Deleting All Rows

In SQL, you can delete all rows from a table by using the DELETE statement without specifying a condition.

```sql
DELETE FROM table_name;
```

Explanation:

- Use the DELETE FROM statement followed by the name of the table from which you want to delete rows.
- Omit the WHERE clause, which means the delete operation will be applied to all rows in the table.

Example:

Consider a table named customers with columns customer_id, customer_name, and city.

You want to clear out all customer records from the customers table:

```sql
DELETE FROM customers;
```

This statement will delete all rows from the customers table, effectively removing all customer records. Use caution when executing such statements, as they permanently remove data from the table. Always make sure to have a backup of your data before performing such operations.

Using Subqueries in Delete

In SQL, you can use subqueries within a DELETE statement to delete rows based on the result of a subquery.

```
DELETE FROM table_name
WHERE column_name IN (SELECT column_name FROM other_table
WHERE condition);
```

Explanation:

- Use the DELETE FROM statement followed by the name of the table you want to delete rows from.
- Use the WHERE clause to specify conditions for deleting rows.
- Use a subquery inside parentheses to retrieve the values to be used in the condition for deletion.

Example:

Consider a table named orders with columns order_id and customer_id, and another table named customers with columns customer_id and status.

You want to delete all orders associated with customers whose status is set to 'inactive':

```
DELETE FROM orders
WHERE customer_id IN (SELECT customer_id FROM customers
WHERE status = 'inactive');
```

This statement will delete all rows from the orders table where the customer_id matches any customer_id from the customers table with a status of 'inactive'.

Field Data Types

In SQL, each field (or column) in a table has a data type that defines the kind of data it can store. Different database management systems (DBMS) support various data types, but some common ones include:

Integer: Represents whole numbers without decimal points (e.g., INT, INTEGER, SMALLINT, BIGINT).

```sql
CREATE TABLE ExampleIntegers (
    id INT,
    age SMALLINT,
    salary BIGINT
);
```

Floating-point: Represents numbers with decimal points (e.g., FLOAT, REAL, DOUBLE, DECIMAL).

```sql
CREATE TABLE ExampleFloatingPoint (
    price FLOAT,
    rate REAL,
    amount DECIMAL(10, 2)
);
```

Character strings: Represents sequences of characters (e.g., CHAR, VARCHAR, TEXT).

```sql
CREATE TABLE ExampleStrings (
    name CHAR(50),
    description VARCHAR(255),
    notes TEXT
);
```

Date and time: Represents dates, times, or both (e.g., DATE, TIME, DATETIME, TIMESTAMP).

```sql
CREATE TABLE ExampleDateTime (
    birthdate DATE,
    appointment_time TIME,
    created_at TIMESTAMP
);
```

Boolean: Represents true/false values (e.g., BOOLEAN, BOOL).

```
CREATE TABLE ExampleBoolean (
    is_active BOOLEAN,
    is_admin BOOL
);
```

Binary: Represents binary data, such as images or documents (e.g., BLOB, BYTEA).

```
CREATE TABLE ExampleBinary (
    image_data BLOB,
    document_data BYTEA
);
```

Each data type has its own characteristics, such as storage size, range of values, and behavior in operations like sorting and comparison. It's essential to choose the appropriate data type for each field based on the nature of the data it will store and the operations it will support.

Creating Tables

Basic Table Creation

In SQL, you can create a table using the CREATE TABLE statement:

```sql
CREATE TABLE table_name (
    column1 datatype1 [constraint1],
    column2 datatype2 [constraint2],
    ...
    [table_constraint]
);
```

Explanation:

- Use the CREATE TABLE statement followed by the name of the table you want to create.
- List the columns you want the table to have, along with their data types.
- Optionally, specify constraints for each column to enforce rules or conditions on the data.
- You can also define table-level constraints that apply to multiple columns or the entire table.

Example:

Consider creating a table named employees with columns for employee ID, name, and salary:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    salary DECIMAL(10, 2)
);
```

This statement will create a table named employees with three columns: employee_id, employee_name, and salary. The employee_id column is defined as an integer and designated as the primary key, ensuring its uniqueness. The employee_name column is defined as a variable-length string, and the salary column is defined as a decimal number with precision and scale specified.

Working with NULL Values

In SQL, NULL represents the absence of a value or an unknown value for a particular data item. When creating tables, you can specify whether a column allows NULL values or not.

```sql
CREATE TABLE table_name (
    column1 datatype1 [NULL | NOT NULL],
    column2 datatype2 [NULL | NOT NULL],
    ...
);
```

Explanation:

- After specifying the data type for each column, you can use the NULL or NOT NULL keywords to indicate whether NULL values are allowed in that column.
- If you don't explicitly specify NULL or NOT NULL, the default behavior depends on the database system you're using.

Example:

Consider a table named students with columns for student ID, name, and date of birth. Let's allow the date of birth to be NULL because not all students may have their birth dates recorded:

```sql
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    date_of_birth DATE NULL
);
```

In this example, the student_id column is designated as the primary key, ensuring uniqueness. The student_name column is specified as NOT NULL, meaning it must have a value for every row. However, the date_of_birth column is allowed to contain NULL values, as indicated by the NULL keyword. This flexibility allows for situations where the date of birth information may not be available for some students.

Specifying Default Values

In SQL, you can specify default values for columns when creating tables. Default values are used when an explicit value is not provided during an INSERT operation.

```
CREATE TABLE table_name (
    column1 datatype1 DEFAULT default_value1,
    column2 datatype2 DEFAULT default_value2,
    ...
);
```

Explanation:

- After specifying the data type for each column, you can use the DEFAULT keyword followed by the default value you want to assign to that column.
- When a new row is inserted into the table and no value is provided for a column with a default value, the default value will be used instead.

Example:

Consider a table named tasks with columns for task ID, description, and priority. Let's specify a default priority value of 'Normal' for new tasks:

```
CREATE TABLE tasks (
    task_id INT PRIMARY KEY,
    description VARCHAR(255) NOT NULL,
    priority VARCHAR(50) DEFAULT 'Normal'
);
```

In this example, if you insert a new row into the tasks table without specifying a value for the priority column, it will default to 'Normal'. However, if you provide a value for the priority column during insertion, that value will be used instead of the default.

Creating Tables with Primary Key

In SQL, a primary key is a column or combination of columns that uniquely identifies each row in a table. When creating a table, you can specify one or more primary key columns using the PRIMARY KEY constraint.

```sql
CREATE TABLE table_name (
    column1 datatype1 PRIMARY KEY,
    column2 datatype2,
    ...
);
```

Explanation:

- After specifying the data type for each column, you can designate one column as the primary key by using the PRIMARY KEY constraint after its definition.
- Each table can have only one primary key, but it can consist of one or multiple columns.
- The primary key constraint ensures that the values in the specified column(s) are unique for each row, and it also automatically creates an index on the primary key column(s) for faster data retrieval.

Example:

Consider a table named employees with columns for employee ID, name, and department. Let's designate the employee_id column as the primary key:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department VARCHAR(50)
);
```

In this example, the employee_id column is designated as the primary key, ensuring that each employee has a unique identifier. Any attempts to insert duplicate values into the employee_id column will result in a constraint violation error.

Creating Tables with Foreign Key

In SQL, a foreign key is a column or combination of columns that establishes a link between data in two tables, enforcing referential integrity. When creating a table, you can specify a foreign key constraint to enforce this relationship.

```
CREATE TABLE child_table (
    column1 datatype1,
    column2 datatype2,
    foreign_key_column datatype,
    FOREIGN KEY (foreign_key_column) REFERENCES
parent_table(parent_column)
);
```

Explanation:

- After defining the columns for the child table, you can specify a foreign key constraint using the FOREIGN KEY keyword followed by the column name(s) that will serve as the foreign key(s).
- The REFERENCES keyword indicates the parent table and the column(s) in the parent table that the foreign key(s) reference.
- This constraint ensures that values in the foreign key column(s) of the child table must exist in the referenced column(s) of the parent table.

Example:

Consider two tables named orders and customers. Each order in the orders table is associated with a customer from the customers table. Let's create the orders table with a foreign key constraint referencing the customer_id column in the customers table:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    total_amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES
customers(customer_id)
);
```

In this example, the customer_id column in the orders table is defined as a foreign key referencing the customer_id column in the customers table. This ensures that every customer_id in the orders table must correspond to an existing customer_id in the customers table, maintaining referential integrity between the two tables.

Updating Tables

Adding a Column

In SQL, you can alter an existing table to add a new column using the ALTER TABLE statement.

```
ALTER TABLE table_name
ADD column_name datatype [constraints];
```

Explanation:

- Use the ALTER TABLE statement followed by the name of the table you want to modify.
- Use the ADD keyword followed by the new column's name and its data type.
- Optionally, you can specify constraints for the new column, such as NOT NULL or DEFAULT values.

Example:

Consider a table named employees with columns for employee ID, name, and department. Let's add a new column named email to store employee email addresses:

```
ALTER TABLE employees
ADD email VARCHAR(255) UNIQUE;
```

In this example, we're adding a new column named email with a data type of VARCHAR(255) to store email addresses. Additionally, we specify the UNIQUE constraint to ensure that each email address is unique across all rows in the employees table.

Deleting a Column

In SQL, you can alter an existing table to delete a column using the ALTER TABLE statement.

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

Explanation:

- Use the ALTER TABLE statement followed by the name of the table you want to modify.
- Use the DROP COLUMN keyword followed by the name of the column you want to delete.

Example:

Consider a table named employees with columns for employee ID, name, department, and email. Let's say we want to remove the email column:

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

In this example, the email column will be removed from the employees table. All data in the email column will be deleted, so use this operation with caution, especially if the column contains important information.

Deleting a Table

In SQL, you can delete an existing table using the DROP TABLE statement.

```
DROP TABLE table_name;
```

Explanation:

- Use the DROP TABLE statement followed by the name of the table you want to delete.
- This statement permanently removes the entire table, including all data and metadata associated with it.

Example:

Consider a table named employees that you want to delete:

```
DROP TABLE employees;
```

In this example, the employees table will be deleted from the database. Be cautious when using the DROP TABLE statement, as it cannot be undone, and all data in the table will be lost. Make sure to back up important data before performing this operation.

Renaming a Table

In SQL, you can rename an existing table using the ALTER TABLE statement.

```
ALTER TABLE current_table_name
RENAME TO new_table_name;
```

Explanation:

- Use the ALTER TABLE statement followed by the current name of the table you want to rename.
- Use the RENAME TO clause followed by the new name you want to assign to the table.

Example:

Consider a table named old_table that you want to rename to new_table:

```
ALTER TABLE old_table
RENAME TO new_table;
```

In this example, the old_table will be renamed to new_table. After renaming the table, all references to the old table name will be updated to use the new name. This operation only changes the name of the table and does not affect its structure or data.

Adding a Primary Key

In SQL, you can add a primary key constraint to an existing table using the ALTER TABLE statement. Here's how you can add a primary key to a table:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column_name);
```

Explanation:

- Use the ALTER TABLE statement followed by the name of the table you want to modify.
- Use the ADD CONSTRAINT keyword followed by the name you want to assign to the primary key constraint.
- Specify PRIMARY KEY to indicate that the constraint is a primary key constraint.
- Inside parentheses, specify the column name(s) that will be part of the primary key.

Example:

Consider a table named employees that already exists, and you want to add a primary key constraint on the employee_id column:

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

In this example, a primary key constraint named pk_employee_id is added to the employees table on the employee_id column. This constraint ensures that each value in the employee_id column is unique and not null, serving as the primary key for the table.

Adding a Foreign Key

In SQL, you can add a foreign key constraint to an existing table using the ALTER TABLE statement. Here's how you can add a foreign key to a table:

```sql
ALTER TABLE child_table
ADD CONSTRAINT constraint_name FOREIGN KEY
(foreign_key_column)
REFERENCES parent_table(parent_column);
```

Explanation:

- Use the ALTER TABLE statement followed by the name of the child table you want to modify.
- Use the ADD CONSTRAINT keyword followed by the name you want to assign to the foreign key constraint.
- Specify FOREIGN KEY to indicate that the constraint is a foreign key constraint.
- Inside parentheses, specify the column name(s) in the child table that will serve as the foreign key(s).
- Use the REFERENCES keyword to specify the parent table and the column(s) in the parent table that the foreign key(s) reference.

Example:

Consider a table named orders that already exists, and you want to add a foreign key constraint on the customer_id column referencing the customer_id column in the customers table:

```sql
ALTER TABLE orders
ADD CONSTRAINT fk_customer_id FOREIGN KEY (customer_id)
REFERENCES customers(customer_id);
```

In this example, a foreign key constraint named fk_customer_id is added to the orders table on the customer_id column. This constraint ensures that every value in the customer_id column of the orders table exists in the customer_id column of the customers table, maintaining referential integrity between the two tables.

Deleting a Foreign Key Constraint

In SQL, you can remove a foreign key constraint from an existing table using the ALTER TABLE statement.

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

Explanation:

- Use the ALTER TABLE statement followed by the name of the table containing the foreign key constraint you want to remove.
- Use the DROP CONSTRAINT keyword followed by the name of the foreign key constraint you want to delete.

Example:

Consider a table named orders that has a foreign key constraint named fk_customer_id referencing the customer_id column in the customers table. To remove this foreign key constraint:

```
ALTER TABLE orders
DROP CONSTRAINT fk_customer_id;
```

In this example, the foreign key constraint named fk_customer_id is removed from the orders table. After executing this statement, there will be no constraint enforcing referential integrity between the customer_id column in the orders table and the customer_id column in the customers table.

Using Views

Creating Views

In SQL, a view is a virtual table generated from the result of a SELECT query. Views provide a way to present data from one or more tables in a structured format without altering the underlying tables.

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

Explanation:

- Use the CREATE VIEW statement followed by the name you want to assign to the view.
- Specify the columns you want the view to contain in the SELECT clause.
- Define the source tables from which the view will retrieve data in the FROM clause.
- Optionally, you can include a WHERE clause to filter the rows returned by the view.

Example:

Consider a scenario where you want to create a view named customer_orders to display the order details of customers who reside in a specific city:

```
CREATE VIEW customer_orders AS
SELECT orders.order_id, orders.order_date,
customers.customer_name
FROM orders
JOIN customers ON orders.customer_id =
customers.customer_id
WHERE customers.city = 'New York';
```

In this example, the customer_orders view is created to show the order ID, order date, and customer name for customers residing in New York. The view retrieves data from the orders and customers tables and applies a condition to filter customers based on their city.

Deleting Views

In SQL, you can delete an existing view using the DROP VIEW statement.

```sql
DROP VIEW view_name;
```

Explanation:

Use the DROP VIEW statement followed by the name of the view you want to delete.

This statement permanently removes the view definition, and the view will no longer be available for querying.

Example:

Consider a scenario where you want to delete a view named customer_orders:

```sql
DROP VIEW customer_orders;
```

In this example, the customer_orders view will be deleted from the database. After executing this statement, the view definition is removed, and attempts to query the view will result in an error because the view no longer exists.

Calculated Fields

In SQL, you can create views with calculated fields, which are derived from existing columns or involve mathematical operations, string concatenation, or other manipulations.

```
CREATE VIEW view_name AS
SELECT column1, column2, ..., expression AS calculated_field
FROM table_name;
```

Explanation:

- Use the CREATE VIEW statement followed by the name you want to assign to the view.
- In the SELECT clause, specify the existing columns you want to include in the view.
- Use expressions to calculate new fields, assigning them an alias using the AS keyword.
- The calculated fields can involve arithmetic operations, string concatenation, or any other supported SQL functions.

Example:

Consider a scenario where you want to create a view named order_totals to display order details along with the total amount for each order:

```
CREATE VIEW order_totals AS
SELECT order_id, order_date, product_name, quantity,
price_per_unit, (quantity * price_per_unit) AS
total_amount
FROM orders
JOIN order_details ON orders.order_id =
order_details.order_id;
```

In this example, the order_totals view is created to show order details along with a calculated field total_amount, which is computed by multiplying the quantity and price_per_unit columns. The view retrieves data from the orders and order_details tables and includes the calculated field total_amount in the output.

Filtering Data

In SQL, you can create views that include filtered data by applying a WHERE clause in the view definition. This allows you to define specific criteria for the rows included in the view.

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

Explanation:

- Use the CREATE VIEW statement followed by the name you want to assign to the view.
- In the SELECT clause, specify the columns you want to include in the view.
- Use the FROM clause to specify the source table(s) from which the view will retrieve data.
- Use the WHERE clause to define the condition(s) for filtering the rows in the view.

Example:

Consider a scenario where you want to create a view named high_value_orders to display orders with a total amount exceeding a certain threshold:

```
CREATE VIEW high_value_orders AS
SELECT order_id, order_date, total_amount
FROM orders
WHERE total_amount > 1000;
```

In this example, the high_value_orders view is created to show order details for orders with a total_amount exceeding $1000. The view retrieves data from the orders table and includes only those rows where the total_amount column meets the specified condition.