SQL Cookbook

Everyone can cook delicious recipes

300+ recipes

By Hernando Abella

ALUNA PUBLISHING HOUSE

Thank you for trusting our Publishing House. If you have the opportunity to evaluate our work and give us a comment on Amazon, we will appreciate it very much!

Table of contents

Introduction

Purpose and Structure of the Cookbook

Welcome to the SQL Cookbook, a comprehensive guide to solving common SQL problems and challenges. This cookbook is designed to provide practical solutions and recipes for developers, data analysts, and database administrators who work with SQL databases on a daily basis.

The primary purpose of this cookbook is to serve as a quick reference and problem-solving guide for SQL practitioners. It covers a wide range of topics, from basic SQL queries to advanced techniques such as window functions and performance optimization.

The structure of the cookbook is organized into various chapters, each focusing on a specific aspect of SQL development. Within each chapter, you'll find recipes that address specific tasks or challenges, along with explanations and examples to help you understand the concepts.

How to Use the Cookbook

This cookbook is designed to be used as a reference guide whenever you encounter a SQL-related problem or need guidance on a particular task. You can either browse through the table of contents to find relevant topics or use the index to search for specific keywords or queries.

Each recipe in the cookbook follows a consistent format:

- **Problem Statement:** A brief description of the problem or task at hand.
- **Solution:** The SQL query or technique used to solve the problem.
- **Explanation:** An explanation of how the solution works and why it's effective.
- **Example:** One or more examples demonstrating the solution in action.

Tips for Efficient SQL Querying

Before diving into the recipes, here are some general tips to help you write efficient SQL queries:

Use indexes to improve query performance, especially for columns frequently used in **WHERE** clauses or **JOIN** conditions.

Minimize the use of wildcard characters (%) in LIKE queries, as they can lead to inefficient full table scans.

Avoid using **SELECT *** to retrieve all columns from a table. Instead, specify only the columns you need.

Use parameterized queries or prepared statements to prevent SQL injection attacks and improve security.

Regularly monitor and optimize your database schema, indexes, and query execution plans for better performance.

By following these tips and leveraging the recipes in this cookbook, you'll be better equipped to tackle SQL challenges and become a more proficient SQL developer.

Retrieving Data

Retrieving Individual Columns

Problem: You want to retrieve specific columns from a table rather than fetching all columns.

Solution:

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

Explanation: The SELECT statement is used to retrieve data from a database. By specifying the column names separated by commas after the SELECT keyword, you can selectively retrieve only the columns you need from the specified table.

Example: Consider a table named employees with columns employee_id, first_name, last_name, email, and hire_date. To retrieve only the first_name and last_name columns from the employees table:

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

Output:

```
| first_name | last_name |
|------------|-----------|
| John       | Doe       |
| Jane       | Smith     |
| Michael    | Johnson   |
| ...        | ...       |
```

This query will return only the first_name and last_name columns for all rows in the employees table.

Retrieving Multiple Columns

Problem: You want to retrieve multiple columns from a table in a single query.

Solution:

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

Explanation: The SELECT statement allows you to specify multiple column names separated by commas after the SELECT keyword. This enables you to retrieve multiple columns from the specified table in a single query.

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

To retrieve the product_id, product_name, and price columns from the products table:

```
SELECT product_id, product_name, price
FROM products;
```

Output:

```
| product_id | product_name   | price |
|------------|----------------|-------|
| 1          | Laptop         | 1200  |
| 2          | Smartphone     | 800   |
| 3          | T-shirt        | 20    |
| 4          | Headphones     | 150   |
| 5          | Jeans          | 50    |
| ...        | ...            | ...   |
```

This query will return the product_id, product_name, and price columns for all rows in the products table.

Retrieving All Columns

Problem: You want to retrieve all columns from a table in a single query.

Solution:

```
SELECT *
FROM table_name;
```

Explanation:

- The asterisk (*) wildcard character in the SELECT statement represents all columns in the specified table. Using SELECT * allows you to retrieve all columns from the table without specifying each column individually.

Example: Consider a table named employees with columns employee_id, first_name, last_name, email, and hire_date.

To retrieve all columns from the employees table:

```
SELECT *

FROM employees;
```

Output:

```
| employee_id | first_name | last_name | email               | hire_date  |
|-------------|------------|-----------|---------------------|------------|
| 1           | John       | Doe       | john@example.com    | 2021-01-15 |
| 2           | Jane       | Smith     | jane@example.com    | 2021-02-20 |
| 3           | Michael    | Johnson   | michael@example.com | 2021-03-10 |
| 4           | Emily      | Brown     | emily@example.com   | 2021-04-05 |
| ...         | ...        | ...       | ...                 | ...        |
```

This query will return all columns for all rows in the employees table.

Retrieving Distinct Rows

Problem: You want to retrieve only the distinct rows from a table, eliminating duplicate rows.

Solution:

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

Explanation:

- The SELECT DISTINCT statement retrieves only unique rows from the specified columns in the table. Duplicate rows are eliminated, and only one instance of each distinct row is returned in the result set.

Example: Consider a table named orders with columns order_id, customer_id, and order_date. To retrieve only the distinct customer IDs from the orders table:

```
SELECT DISTINCT customer_id
FROM orders;
```

Output:

```
| customer_id |
|-------------|
| 1001        |
| 1002        |
| 1003        |
| ...         |
```

This query will return only the unique customer IDs from the orders table, eliminating any duplicate entries.

Limiting Results

Problem: You want to limit the number of rows returned by a query to a specified number.

Solution:

```
SELECT column1, column2, ...
FROM table_name
LIMIT number_of_rows;
```

Explanation:

- The LIMIT clause restricts the number of rows returned by the query to the specified value. It is typically used in conjunction with the SELECT statement to limit the result set to a manageable size.

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

To retrieve the first 10 rows from the products table:

```
SELECT product_id, product_name, price
FROM products
LIMIT 10;
```

Output:

```
| product_id | product_name   | price |
|------------|----------------|-------|
| 1          | Laptop         | 1200  |
| 2          | Smartphone     | 800   |
| 3          | T-shirt        | 20    |
| 4          | Headphones     | 150   |
| 5          | Jeans          | 50    |
```

This query will return only the first 10 rows from the products table.

Using Comments

Problem:

You want to add comments to your SQL queries to improve readability and provide explanatory notes.

Solution:

```
-- Single-line comment
SELECT column1, column2
FROM table_name; -- Inline comment

/*
   Multi-line comment
   This query retrieves data from the specified columns
in the table.
*/
SELECT column1, column2
FROM table_name;
```

Explanation:

- **Single-line Comment:** Starts with -- and extends to the end of the line. Useful for adding brief comments on a single line.
- **Inline Comment:** Placed after a statement on the same line, following --. Useful for adding comments inline with code.
- **Multi-line Comment:** Enclosed between /* and */. Can span multiple lines and is useful for longer comments or explanations.

SELECT Clause Ordering

Problem: You want to retrieve data from a table and order the results based on one or more columns.

Solution:

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

Explanation:

- The ORDER BY clause is used in conjunction with the SELECT statement to sort the result set based on specified columns. By default, sorting is done in ascending order (ASC), but you can specify descending order (DESC) if needed.

Example:

```
-- Retrieve employee names ordered by last name in
ascending order
SELECT first_name, last_name
FROM employees
ORDER BY last_name ASC;

-- Retrieve products ordered by price in descending
order, then by product name in ascending order
SELECT product_name, price
FROM products
ORDER BY price DESC, product_name ASC;
```

Output: The output will display the selected columns from the table, ordered according to the specified criteria.

```
| first_name | last_name |
|------------|-----------|
| John       | Doe       |
| Jane       | Smith     |
| ...        | ...       |

| product_name  | price |
|---------------|-------|
| Laptop        | 1200  |
| Smartphone    | 800   |
| ...           | ...   |
```

Sorting Data

Sort by One Column

Problem: You want to retrieve data from a table and sort the results based on one specific column.

Solution:

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

Explanation:

- The ORDER BY clause in conjunction with the SELECT statement is used to sort the result set based on the values of a specific column. By default, sorting is done in ascending order (ASC), but you can specify descending order (DESC) if needed.

Example: Consider a table named employees with columns employee_id, first_name, last_name, and hire_date. To retrieve employee data sorted by their hire date:

```
SELECT employee_id, first_name, last_name, hire_date
FROM employees
ORDER BY hire_date;
```

Output: The output will display the selected columns from the employees table, sorted based on the hire_date column.

```
| employee_id | first_name | last_name | hire_date  |
|-------------|------------|-----------|------------|
| 102         | John       | Doe       | 2021-01-15 |
| 105         | Jane       | Smith     | 2021-02-20 |
| 103         | Michael    | Johnson   | 2021-03-10 |
| ...         | ...        | ...       | ...        |
```

Sort by Multiple Columns

Problem: You want to retrieve data from a table and sort the results based on multiple columns.

Solution:

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

Explanation:

- The ORDER BY clause allows you to sort the result set based on multiple columns. You can specify multiple columns in the order you want them to be sorted. Sorting is done in ascending order (ASC) by default, but you can specify descending order (DESC) if needed.

Example: Consider a table named employees with columns department_id, last_name, and hire_date. To retrieve employee data sorted first by department ID in ascending order and then by hire date in

Descending order:

```
SELECT department_id, last_name, hire_date
FROM employees
ORDER BY department_id ASC, hire_date DESC;
```

Output: The output will display the selected columns from the employees table, sorted first by department_id in ascending order, and then by hire_date in descending order.

```
| department_id | last_name | hire_date  |
|---------------|-----------|------------|
| 10            | Smith     | 2021-03-10 |
| 10            | Johnson   | 2021-02-20 |
| 20            | Doe       | 2021-01-15 |
| ...           | ...       | ...        |
```

Sort by Column Position

Sorting by column position refers to ordering the results based on the position of columns rather than their names.

Problem Statement: You want to retrieve data from a table and sort the results based on the position of columns in the result set.

Solution:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY 1 [ASC | DESC], 2 [ASC | DESC], ...;
```

Explanation:

- Instead of specifying column names in the ORDER BY clause, you can use the positional notation to refer to columns by their position in the result set. The first column has position 1, the second column has position 2, and so on. Sorting is done in ascending order (ASC) by default, but you can specify descending order (DESC) if needed.

Example: Consider a table named employees with columns department_id, last_name, and hire_date. To retrieve employee data and sort it by the second column in ascending order and the third column in descending order:

```
SELECT department_id, last_name, hire_date
FROM employees
ORDER BY 2 ASC, 3 DESC;
```

Output: The output will display the selected columns from the employees table, sorted by the second column (last_name) in ascending order and the third column (hire_date) in descending order.

department_id	last_name	hire_date
20	Doe	2021-01-15
10	Johnson	2021-02-20
10	Smith	2021-03-10
...	...	...

Specifying Sort Direction

When specifying the sort direction in SQL, you can use either ASC (ascending) or DESC (descending) after each column in the ORDER BY clause.

Problem: You want to retrieve data from a table and specify the sort direction for each column in the result set.

Solution:

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

Explanation:

- The ASC keyword specifies ascending order (the default if not specified), while the DESC keyword specifies descending order for sorting. You can use these keywords after each column name in the ORDER BY clause to control the sort direction.

Example: Consider a table named employees with columns last_name, first_name, and hire_date. To retrieve employee data and sort it by last name in ascending order and hire date in descending order:

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

Output: The output will display the selected columns from the employees table, sorted by last name in ascending order and hire date in descending order.

```
| last_name | first_name | hire_date  |
|-----------|------------|------------|
| Doe       | John       | 2021-01-15 |
| Johnson   | Michael    | 2021-03-10 |
| Smith     | Jane       | 2021-02-20 |
| ...       | ...        | ...        |
```

Filtering Data

Checking for Matches

Filtering data in SQL involves checking for matches based on specified criteria using the WHERE clause.

Problem: You want to retrieve data from a table based on specified criteria, checking for matches in one or more columns.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Explanation:

- The WHERE clause is used to filter rows based on a specified condition. Only rows that meet the condition will be included in the result set. The condition can involve comparisons, logical operators, and functions to check for matches in one or more columns.

Example: Consider a table named employees with columns last_name, department, and salary.

To retrieve employee data for those working in the Sales department:

```
SELECT last_name, department, salary
FROM employees
WHERE department = 'Sales';
```

Output: The output will display the selected columns from the employees table for employees working in the Sales department.

```
| last_name | department | salary |
|-----------|------------|--------|
| Smith     | Sales      | 50000  |
| Johnson   | Sales      | 48000  |
| ...       | ...        | ...    |
```

Checking for Nonmatches

To check for non-matches in SQL, you can use the NOT operator in conjunction with comparison operators in the WHERE clause.

Problem: You want to retrieve data from a table based on specified criteria, checking for non-matches in one or more columns.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

Explanation:

- The NOT operator is used to negate a condition in the WHERE clause. It returns true if the condition is false and false if the condition is true. By using the NOT operator, you can filter rows that do not meet the specified criteria.

Example: Consider a table named employees with columns last_name, department, and salary. To retrieve employee data for those not working in the Sales department:

```
SELECT last_name, department, salary
FROM employees
WHERE department <> 'Sales';
```

Output: The output will display the selected columns from the employees table for employees not working in the Sales department.

```
| last_name | department | salary |
|-----------|------------|--------|
| Doe       | HR         | 60000  |
| Brown     | Finance    | 55000  |
| ...       | ...        | ...    |
```

Checking for a Range of Values

To check for a range of values in SQL, you can use comparison operators such as greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=) in the WHERE clause.

Problem: You want to retrieve data from a table based on specified criteria, checking for values within a certain range in one or more columns.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE column BETWEEN value1 AND value2;
```

Explanation:

- The BETWEEN operator is used to specify a range of values in the WHERE clause. It checks if a value lies within the specified range, inclusive of the endpoints. You can use it to filter rows based on a range of values in one or more columns.

Example: Consider a table named employees with columns last_name, age, and salary. To retrieve employee data for those aged between 25 and 35:

```
SELECT last_name, age, salary
FROM employees
WHERE age BETWEEN 25 AND 35;
```

Output: The output will display the selected columns from the employees table for employees aged between 25 and 35.

```
| last_name | age | salary |
|-----------|-----|--------|
| Smith     | 28  | 50000  |
| Johnson   | 32  | 48000  |
| ...       | ... | ...    |
```

Checking for No Value

To check for the absence of a value in SQL, you can use the IS NULL or IS NOT NULL operators in the WHERE clause.

Problem: You want to retrieve data from a table based on specified criteria, checking for the absence of a value in one or more columns.

Solution

```
SELECT column1, column2, ...
FROM table_name
WHERE column IS [NOT] NULL;
```

Explanation:

- The IS NULL operator is used to check if a column contains no value (NULL). Conversely, the IS NOT NULL operator is used to check if a column contains any value other than NULL. You can use these operators in the WHERE clause to filter rows based on the presence or absence of values in one or more columns.

Example: Consider a table named customers with columns customer_id, first_name, and last_name. To retrieve customer data for those without a last name:

```
SELECT customer_id, first_name
FROM customers
WHERE last_name IS NULL;
```

Output: The output will display the selected columns from the customers table for customers without a last name.

```
| customer_id | first_name |
|-------------|------------|
| 1           | John       |
| 2           | Jane       |
| ...         | ...        |
```

Using the AND Operator

To filter data using multiple conditions in SQL, you can use the AND operator in the WHERE clause.

Problem: You want to retrieve data from a table based on multiple conditions, where all conditions must be true for a row to be included in the result set.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND ...;
```

Explanation:

- The AND operator is used to combine multiple conditions in the WHERE clause. It requires that all conditions must evaluate to true for a row to be included in the result set. You can use it to filter data based on multiple criteria simultaneously.

Example: Consider a table named employees with columns department, age, and salary. To retrieve employee data for those working in the Sales department and earning more than $50,000:

```
SELECT department, age, salary
FROM employees
WHERE department = 'Sales' AND salary > 50000;
```

Output: The output will display the selected columns from the employees table for employees working in the Sales department and earning more than $50,000.

```
| department | age | salary |
|------------|-----|--------|
| Sales      | 35  | 60000  |
| Sales      | 28  | 52000  |
| ...        | ... | ...    |
```

Using the OR Operator

To filter data using multiple conditions in SQL, where any of the conditions must be true for a row to be included in the result set, you can use the OR operator in the WHERE clause.

Problem: You want to retrieve data from a table based on multiple conditions, where at least one condition must be true for a row to be included in the result set.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR ...;
```

Explanation:

- The OR operator is used to combine multiple conditions in the WHERE clause. It requires that at least one of the conditions must evaluate to true for a row to be included in the result set. You can use it to filter data based on multiple criteria and retrieve rows that meet any of those criteria.

Example: Consider a table named employees with columns department, age, and salary. To retrieve employee data for those working in the Sales department or earning more than $50,000:

```
SELECT department, age, salary
FROM employees
WHERE department = 'Sales' OR salary > 50000;
```

Output: The output will display the selected columns from the employees table for employees working in the Sales department or earning more than $50,000.

department	age	salary
Sales	35	60000
HR	40	55000
...	...	...

Using AND and OR Operators

The AND and OR operators are fundamental in SQL for constructing complex conditions in the WHERE clause.

Problem: You want to retrieve data from a table based on multiple conditions, combining them with AND and OR operators.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND/OR condition2 AND/OR ...;
```

Explanation:

- **AND Operator:** Combines multiple conditions and requires all conditions to be true for a row to be included in the result set.
- **OR Operator:** Combines multiple conditions and requires at least one condition to be true for a row to be included in the result set.

Example: Consider a table named employees with columns department, age, and salary. To retrieve employee data for those working in the Sales department and earning more than $50,000, or those working in the HR department and aged below 30:

```
SELECT department, age, salary
FROM employees
WHERE (department = 'Sales' AND salary > 50000)
    OR (department = 'HR' AND age < 30);
```

Output: The output will display the selected columns from the employees table for employees meeting any of the specified conditions.

```
| department | age | salary |
|------------|-----|--------|
| Sales      | 35  | 60000  |
| HR         | 25  | 48000  |
| ...        | ... | ...    |
```

Using the IN Operator

The IN operator in SQL allows you to specify multiple values in a WHERE clause, making it convenient for filtering data against a list of possible values.

Problem: You want to retrieve data from a table where a particular column matches any value in a specified list.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

Explanation:

- The IN operator allows you to specify a list of values against which a column's value will be compared. If the column's value matches any value in the list, the row will be included in the result set.

Example: Consider a table named employees with columns department and salary. To retrieve employee data for those working in either the Sales or Marketing department:

```
SELECT department, salary
FROM employees
WHERE department IN ('Sales', 'Marketing');
```

Output: The output will display the selected columns from the employees table for employees working in the Sales or Marketing department.

```
| department | salary |
|------------|--------|
| Sales      | 60000  |
| Marketing  | 55000  |
| Sales      | 52000  |
| ...        | ...    |
```

Using the NOT Operator

The NOT operator in SQL is used to negate a condition in the WHERE clause, allowing you to retrieve data that does not meet a specified condition.

Problem: You want to retrieve data from a table where a particular column does not match a specified value or condition.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

Explanation:

- The NOT operator negates a condition in the WHERE clause. It returns true if the condition is false and false if the condition is true. You can use it to filter out rows that do not meet the specified criteria.

Example: Consider a table named employees with columns department and salary. To retrieve employee data for those not working in the Sales department:

```
SELECT department, salary
FROM employees
WHERE NOT department = 'Sales';
```

Output: The output will display the selected columns from the employees table for employees not working in the Sales department.

```
| department | salary |
|------------|--------|
| HR         | 55000  |
| Marketing  | 60000  |
| HR         | 48000  |
| ...        | ...    |
```

Using the LIKE Operator

The LIKE operator in SQL is used to perform pattern matching within string data. It allows you to search for patterns in a column's values using wildcard characters.

Problem: You want to retrieve data from a table where a particular column matches a specific pattern.

Solution:

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE pattern;
```

Explanation:

- The LIKE operator is used to perform pattern matching in SQL. It allows you to search for strings that match a specified pattern. Wildcard characters, such as % and _, can be used to represent unknown parts of the pattern.

Example: Consider a table named employees with a column last_name. To retrieve employee data for those with a last name starting with 'S':

```
SELECT last_name, department
FROM employees
WHERE last_name LIKE 'S%';
```

Output: The output will display the selected columns from the employees table for employees with a last name starting with 'S'.

```
| last_name | department |
|-----------|------------|
| Smith     | Sales      |
| Stewart   | Marketing  |
| ...       | ...        |
```

The WHERE Clause Operators

The WHERE clause in SQL allows you to filter rows based on specified conditions. There are several operators you can use within the WHERE clause to create these conditions. Here's an overview of the most commonly used ones:

Comparison Operators: These operators are used to compare values.

= (Equal to): Matches rows where the value of a column equals a specified value.

<> or != (Not equal to): Matches rows where the value of a column is not equal to a specified value.

< (Less than): Matches rows where the value of a column is less than a specified value.

> (Greater than): Matches rows where the value of a column is greater than a specified value.

<= (Less than or equal to): Matches rows where the value of a column is less than or equal to a specified value.

>= (Greater than or equal to): Matches rows where the value of a column is greater than or equal to a specified value.

Logical Operators: These operators are used to combine multiple conditions.

AND: Matches rows where all specified conditions are true.

OR: Matches rows where at least one of the specified conditions is true.

NOT: Negates a condition, matching rows where the specified condition is false.

Pattern Matching Operators: These operators are used to perform pattern matching within string data.

LIKE: Matches rows where a column value matches a specified pattern using wildcard characters.

IN: Matches rows where a column value is equal to any value in a specified list of values.

BETWEEN: Matches rows where a column value is within a specified range of values.

NULL-related Operators: These operators are used to check for NULL values.

IS NULL: Matches rows where a column value is NULL.

IS NOT NULL: Matches rows where a column value is not NULL.

Calculated fields

Concatenating Fields

You can concatenate fields or strings together to create a calculated field using the concatenation operator (|| in some database systems).

Problem: You want to create a new field in your query result by concatenating values from multiple columns.

Solution:

```
SELECT column1 || ' ' || column2 AS concatenated_field
FROM table_name;
```

Explanation:

- The concatenation operator (||) is used to concatenate strings or columns together. You can concatenate columns, literal strings, or a combination of both to create a new field in your query result.

Example: Consider a table named employees with columns first_name and last_name. To create a new field that concatenates the first name and last name together:

```
SELECT first_name || ' ' || last_name AS full_name
FROM employees;
```

Output: The output will display the concatenated full names of employees.

```
| full_name    |
|--------------|
| John Doe     |
| Jane Smith   |
| ...          |
```

Removing Unwanted Spaces

To remove unwanted spaces from calculated fields in SQL, you can use the TRIM function, which removes leading and trailing spaces from a string.

Problem: You want to create a new field in your query result by concatenating values from multiple columns, and you want to remove any unwanted spaces from the concatenated result.

Solution:

```
SELECT TRIM(column1 || ' ' || column2) AS cleaned_field
FROM table_name;
```

Explanation:

- The TRIM function is used to remove leading and trailing spaces from a string. By applying it to the concatenated result of columns or strings, you can remove any unwanted spaces from the calculated field.

Example: Consider a table named employees with columns first_name and last_name. To create a new field that concatenates the first name and last name together and removes any leading or trailing spaces:

```
SELECT TRIM(first_name || ' ' || last_name) AS
cleaned_full_name
FROM employees;
```

Output: The output will display the cleaned full names of employees with any leading or trailing spaces removed.

```
|cleaned_full_name    |
|--------------------|
| John Doe           |
| Jane Smith         |
| ...                |
```

Using Aliases

Aliases in SQL are used to give a temporary name to a column or expression in the result set of a query. They are particularly useful for making the output more readable or for referencing calculated fields.

Problem:

You want to create calculated fields in your SQL query and give them meaningful names using aliases.

Solution:

```
SELECT expression AS alias_name
FROM table_name;
```

Explanation:

- The AS keyword is used to assign an alias to a column or expression in the SELECT statement.
- Aliases provide a more descriptive name for calculated fields, making the query results easier to understand.

Example:

Consider a table named employees with columns first_name and last_name. To create a calculated field that concatenates the first name and last name together and give it an alias:

```
SELECT first_name || ' ' || last_name AS full_name
FROM employees;
```

Output: The output will display the calculated field with the alias full_name.

```
| full_name   |
|-------------|
| John Doe    |
| Jane Smith  |
| ...         |
```

Mathematical Calculations

To perform mathematical calculations on fields in SQL and create calculated fields, you can use arithmetic operators such as addition (+), subtraction (-), multiplication (*), and division (/).

Problem: You want to perform mathematical calculations on fields in your SQL query to create new calculated fields.

Solution:

```
SELECT column1 + column2 AS sum,
       column1 - column2 AS difference,
       column1 * column2 AS product,
       column1 / column2 AS quotient
FROM table_name;
```

Explanation:

- Arithmetic operators such as addition (+), subtraction (-), multiplication (*), and division (/) can be used to perform mathematical calculations on fields in SQL.
- You can use these operators to create new calculated fields by combining values from existing columns.

Example: Consider a table named sales with columns quantity and unit_price. To calculate the total revenue for each sale:

```
SELECT quantity * unit_price AS total_revenue
FROM sales;
```

Output: The output will display the calculated field total_revenue containing the result of the mathematical calculation for each row.

```
| total_revenue |
|---------------|
| 500           |
| 750           |
| ...           |
```

CASE WHEN Clause

In SQL, the CASE WHEN clause allows you to perform conditional logic and create calculated fields based on different conditions. It's similar to the "if-else" logic in programming languages.

Problem: You want to create a calculated field in your SQL query that varies based on different conditions.

Solution:

```
SELECT
    column1,
    column2,
    CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        ELSE default_result
    END AS calculated_field
FROM
    table_name;
```

Explanation:

- The CASE WHEN clause evaluates each condition in the order specified and returns the result associated with the first condition that evaluates to true.
- It can be used to create conditional logic in SQL queries and generate calculated fields based on various conditions.

Example: Consider a table named employees with a column salary. You want to create a calculated field salary_category based on different salary ranges:

```sql
SELECT
    salary,
    CASE
        WHEN salary < 50000 THEN 'Low'
        WHEN salary >= 50000 AND salary < 100000 THEN
'Medium'
        ELSE 'High'
    END AS salary_category
FROM
   employees;
```

Output: The output will display the original salary column along with the calculated field salary_category based on the specified conditions.

```
| salary | salary_category |
|--------|-----------------|
| 45000  | Low             |
| 75000  | Medium          |
| ...    | ...             |
```

Case Changes

In SQL, you can use various string functions to manipulate the case of strings. To create calculated fields that change the case of values in columns, you can use functions like UPPER, LOWER, INITCAP, or their equivalents in your database system.

Problem Statement:

You want to create a calculated field in your SQL query that changes the case of values in a column.

Solution:

```
SELECT
    column1,
    UPPER(column2) AS upper_case_column,
    LOWER(column3) AS lower_case_column,
    INITCAP(column4) AS initcap_case_column
FROM
    table_name;
```

Explanation:

- String functions like UPPER, LOWER, and INITCAP are used to change the case of strings in SQL.
- You can apply these functions to columns in your SELECT statement to create calculated fields with modified case.

Example: Consider a table named employees with columns first_name and last_name. You want to create calculated fields for the first name and last name in uppercase, lowercase, and initcap (first letter capitalized):

```
SELECT
    UPPER(first_name) AS upper_case_first_name,
    LOWER(last_name) AS lower_case_last_name,
    INITCAP(first_name) AS initcap_first_name,
    INITCAP(last_name) AS initcap_last_name
FROM
    employees;
```

Output: The output will display the original values of first_name and last_name, along with their uppercase, lowercase, and initcap versions.

upper_case_first_name	lower_case_last_name	initcap_first_name	initcap_last_name
JOHN	doe	John	Doe
JANE	smith	Jane	Smith
...	...	...	...

Data Manipulation Functions

Text Manipulation

Text manipulation in SQL involves various string functions to manipulate and transform text data. These functions enable you to perform tasks such as concatenation, substring extraction, trimming, case conversion, and pattern matching.

Problem: You want to manipulate text data in your SQL queries to perform tasks such as concatenation, substring extraction, trimming, case conversion, and pattern matching.

Solution:

```
-- Concatenation
SELECT CONCAT(column1, ' ', column2) AS concatenated_text
FROM table_name;

-- Substring Extraction
SELECT SUBSTRING(column1, start_index, length) AS
extracted_text
FROM table_name;

-- Trimming
SELECT TRIM(column1) AS trimmed_text
FROM table_name;

-- Case Conversion
SELECT UPPER(column1) AS upper_case_text,
       LOWER(column2) AS lower_case_text,
       INITCAP(column3) AS initcap_case_text
FROM table_name;

-- Pattern Matching
SELECT column1
FROM table_name
WHERE column1 LIKE '%pattern%';
```

Explanation:

- **Concatenation:** Use CONCAT function or concatenation operator (||) to combine multiple strings or columns.
- **Substring Extraction:** Use SUBSTRING function to extract a substring from a string based on start index and length.
- **Trimming:** Use TRIM function to remove leading and trailing spaces from a string.
- **Case Conversion:** Use UPPER, LOWER, and INITCAP functions to convert text to uppercase, lowercase, or initcap (first letter capitalized).
- **Pattern Matching:** Use LIKE operator with wildcard characters (%) to search for patterns in text data.

Example: Consider a table named products with columns product_name and description. You want to perform text manipulation tasks on these columns:

```
SELECT CONCAT(product_name, ' - ', description) AS product_info,
       SUBSTRING(description, 1, 50) AS short_description,
       TRIM(product_name) AS trimmed_product_name,
       UPPER(product_name) AS upper_case_product_name,
       LOWER(description) AS lower_case_description
FROM products
WHERE product_name LIKE '%apple%';
```

Output: The output will display the manipulated text data based on the specified operations and conditions.

```
| product_info          | short_description    | trimmed_product_name | upper_case_product_name | lower_case_description |
|-----------------------|----------------------|----------------------|-------------------------|------------------------|
| Apple iPhone - ...    | Latest iPhone model  | Apple iPhone         | APPLE IPHONE            | latest iphone model    |
| ...                   | ...                  | ...                  | ...                     | ...                    |
```

Date and Time Manipulation

Date and time manipulation in SQL involves various functions to perform operations such as extracting parts of dates, formatting dates, adding or subtracting intervals from dates, and calculating date differences.

Problem: You want to manipulate date and time data in your SQL queries to perform tasks such as extracting parts of dates, formatting dates, adding or subtracting intervals from dates, and calculating date differences.

Solution:

```
-- Extracting Parts of Dates
SELECT EXTRACT(YEAR FROM date_column) AS year,
       EXTRACT(MONTH FROM date_column) AS month,
       EXTRACT(DAY FROM date_column) AS day
FROM table_name;

-- Formatting Dates
SELECT TO_CHAR(date_column, 'YYYY-MM-DD') AS formatted_date
FROM table_name;

-- Adding or Subtracting Intervals
SELECT date_column + INTERVAL '1' DAY AS next_day,
       date_column - INTERVAL '1' MONTH AS previous_month
FROM table_name;

-- Calculating Date Differences
SELECT DATE_PART('day', date2 - date1) AS days_difference
FROM table_name;
```

Explanation:

- **Extracting Parts of Dates:** Use the EXTRACT function to extract specific parts (e.g., year, month, day) of a date.
- **Formatting Dates:** Use the TO_CHAR function to format dates into custom date formats.
- **Adding or Subtracting Intervals:** Use the INTERVAL keyword to add or subtract intervals (e.g., days, months) from dates.

- **Calculating Date Differences:** Use the DATE_PART function to calculate the difference between two dates in a specified unit (e.g., days).

Example: Consider a table named orders with a column order_date. You want to perform various date and time manipulation tasks on this column:

```
SELECT EXTRACT(YEAR FROM order_date) AS order_year,
       TO_CHAR(order_date, 'YYYY-MM-DD') AS formatted_order_date,
       order_date + INTERVAL '1' DAY AS next_order_date,
       DATE_PART('day', CURRENT_DATE - order_date) AS days_since_order
FROM orders;
```

Output: The output will display the manipulated date and time data based on the specified operations.

```
| order_year | formatted_order_date | next_order_date | days_since_order |
|------------|----------------------|-----------------|------------------|
| 2023       | 2023-01-15           | 2023-01-16      | 10               |
| ...        | ...                  | ...             | ...              |
```

Numeric Manipulation

Numeric manipulation in SQL involves various functions to perform operations such as arithmetic calculations, rounding, and formatting on numeric data. Here's an overview of commonly used numeric manipulation functions in SQL:

Problem: You want to manipulate numeric data in your SQL queries to perform tasks such as arithmetic calculations, rounding, and formatting.

Solution:

```
-- Arithmetic Calculations
SELECT column1 + column2 AS sum,
       column1 - column2 AS difference,
       column1 * column2 AS product,
       column1 / column2 AS quotient
FROM table_name;

-- Rounding
SELECT ROUND(column1, precision) AS rounded_value,
       CEIL(column2) AS ceiling_value,
       FLOOR(column3) AS floor_value
FROM table_name;

-- Formatting
SELECT FORMAT(column1, format_string) AS formatted_value
FROM table_name;
```

Explanation:

- **Arithmetic Calculations:** Use arithmetic operators (+, -, *, /) to perform basic arithmetic calculations on numeric columns.
- **Rounding:** Use ROUND, CEIL, and FLOOR functions to round numeric values to a specified precision or to the nearest integer.
- **Formatting:** Use the FORMAT function to format numeric values according to a specified format string.

Example: Consider a table named sales with columns quantity and unit_price.

You want to perform various numeric manipulation tasks on these columns:

```
SELECT quantity * unit_price AS total_price,
       ROUND(total_price, 2) AS rounded_total_price,
       CEIL(unit_price) AS ceiling_unit_price,
       FLOOR(unit_price) AS floor_unit_price,
       FORMAT(total_price, '#,###.00') AS formatted_total_price
FROM sales;
```

Output: The output will display the manipulated numeric data based on the specified operations.

```
| total_price | rounded_total_price | ceiling_unit_price | floor_unit_price | formatted_total_price |
|-------------|---------------------|--------------------|------------------|-----------------------|
| 100         | 100.00              | 10                 | 9                | 100.00                |
| ...         | ...                 | ...                | ...              | ...                   |
```

Using Aggregate Functions

Aggregate functions in SQL allow you to perform calculations on sets of values and return a single result. These functions are commonly used to summarize data, such as calculating totals, averages, counts, and finding minimum or maximum values within groups.

Here's an overview of commonly used aggregate functions in SQL:

Problem: You want to perform calculations on sets of values in your SQL queries to summarize data and return aggregated results.

Solution:

```sql
-- Calculating Totals
SELECT SUM(column1) AS total_sum
FROM table_name;

-- Calculating Averages
SELECT AVG(column2) AS average_value
FROM table_name;

-- Counting Rows
SELECT COUNT(*) AS row_count
FROM table_name;

-- Finding Maximum and Minimum Values
SELECT MAX(column3) AS max_value,
       MIN(column4) AS min_value
FROM table_name;

-- Grouping Data and Applying Aggregate Functions
SELECT category,
       COUNT(*) AS category_count,
       AVG(price) AS average_price
FROM products
GROUP BY category;
```

Explanation:

- **Calculating Totals:** Use the SUM function to calculate the total of a numeric column.
- **Calculating Averages:** Use the AVG function to calculate the average value of a numeric column.
- **Counting Rows:** Use the COUNT function to count the number of rows in a result set.
- **Finding Maximum and Minimum Values:** Use the MAX and MIN functions to find the maximum and minimum values of a column.
- **Grouping Data and Applying Aggregate Functions:** Use GROUP BY clause to group rows that have the same values into summary rows, and then use aggregate functions to perform calculations on each group.

Example: Consider a table named sales with columns product_id, quantity, and price. You want to calculate various aggregate values based on this data:

```
-- Calculate total sales amount
SELECT SUM(quantity * price) AS total_sales_amount
FROM sales;

-- Calculate average quantity sold
SELECT AVG(quantity) AS average_quantity_sold
FROM sales;

-- Count the number of sales transactions
SELECT COUNT(*) AS total_sales_count
FROM sales;

-- Find the maximum and minimum price of products
SELECT MAX(price) AS max_price,
       MIN(price) AS min_price
FROM products;

-- Calculate average price by product category
SELECT category,
       AVG(price) AS average_price
FROM products
GROUP BY category;
```

Output: The output will display the aggregated results based on the specified aggregate functions and grouping (if any).

total_sales_amount	average_quantity_sold	total_sales_count	max_price	min_price	category	average_price
5000	10	100	100	50	Electronics	75
...	...	...	...	...	...	...

Grouping Data

Grouping by One Column

Grouping data in SQL allows you to aggregate information based on the values in one or more columns. When grouping by one column, you combine rows with the same value in that column and perform aggregate functions on the grouped data.

Problem Statement: You want to group data in your SQL query based on the values in one column and perform aggregate calculations on each group.

Solution:

```
SELECT column1,
       aggregate_function(column2) AS result1,
       aggregate_function(column3) AS result2
FROM table_name
GROUP BY column1;
```

Explanation:

- Use the GROUP BY clause to specify the column by which you want to group the data.
- Apply aggregate functions to other columns in the SELECT statement to calculate summary statistics for each group.
- The aggregate functions will be applied separately to each group defined by the unique values in the specified column.

Example: Consider a table named orders with columns customer_id and total_amount.

You want to calculate the total amount spent by each customer:

```
SELECT customer_id,
       SUM(total_amount) AS total_spent
FROM orders
GROUP BY customer_id;
```

Output: The output will display the total amount spent by each customer, grouped by their unique customer ID.

```
| customer_id | total_spent |
|-------------|-------------|
| 1           | 500         |
| 2           | 750         |
| ...         | ...         |
```

Grouping by Multiple Columns

Grouping data by multiple columns in SQL allows you to further refine the groups by considering combinations of values from multiple columns. This can be useful for more granular analysis and summarization of data.

Problem: You want to group data in your SQL query based on the values in multiple columns and perform aggregate calculations on each group.

Solution:

```
SELECT column1,
       column2,
       aggregate_function(column3) AS result1,
       aggregate_function(column4) AS result2
FROM table_name
GROUP BY column1, column2;
```

Explanation:

- Use the GROUP BY clause to specify multiple columns by which you want to group the data.
- The rows will be grouped based on unique combinations of values from the specified columns.
- Apply aggregate functions to other columns in the SELECT statement to calculate summary statistics for each group.

Example: Consider a table named orders with columns customer_id, product_category, and total_amount. You want to calculate the total amount spent by each customer in each product category:

```
SELECT customer_id,
       product_category,
       SUM(total_amount) AS total_spent
FROM orders
GROUP BY customer_id, product_category;
```

Output: The output will display the total amount spent by each customer in each product category, grouped by their unique customer ID and product category.

customer_id	product_category	total_spent
1	Electronics	300
1	Clothing	200
2	Electronics	500
...	...	...

Grouping by Column Number

In SQL, you can also group data by column number instead of specifying column names directly. This can be useful when you want to group by columns based on their position in the SELECT statement

Problem: You want to group data in your SQL query based on the position of columns in the SELECT statement rather than specifying column names directly.

Solution:

```
SELECT column1,
       column2,
       aggregate_function(column_number) AS result1,
       aggregate_function(column_number) AS result2
FROM table_name
GROUP BY 1, 2;
```

Explanation:

- Use the GROUP BY clause followed by the positions of columns in the SELECT statement to specify the grouping.
- Column numbers start from 1, representing the first column in the SELECT statement, and increment sequentially.
- The rows will be grouped based on unique combinations of values from the specified columns.

Example: Consider a table named orders with columns customer_id, product_category, and total_amount. You want to calculate the total amount spent by each customer in each product category:

```
SELECT customer_id,
       product_category,
       SUM(total_amount) AS total_spent
FROM orders
GROUP BY 1, 2;
```

Output: The output will display the total amount spent by each customer in each product category, grouped by their unique customer ID and product category.

customer_id	product_category	total_spent
1	Electronics	300
1	Clothing	200
2	Electronics	500
...	...	...

Filtering Groups

Problem: Filtering groups in SQL involves applying conditions to the aggregated results of grouped data. This allows you to include or exclude specific groups based on certain criteria.

You want to filter groups in your SQL query based on certain conditions applied to the aggregated results of grouped data.

Solution:

```
SELECT grouping_column,
       aggregate_function(column1) AS result1,
       aggregate_function(column2) AS result2
FROM table_name
GROUP BY grouping_column
HAVING condition;
```

Explanation:

- Use the GROUP BY clause to specify the column by which you want to group the data.
- Apply aggregate functions to other columns in the SELECT statement to calculate summary statistics for each group.
- Use the HAVING clause to specify conditions that filter the groups based on the aggregated results.

Example:

Consider a table named orders with columns customer_id and total_amount. You want to calculate the total amount spent by each customer and filter out customers who have spent less than $1000:

```
SELECT customer_id,
       SUM(total_amount) AS total_spent
FROM orders
GROUP BY customer_id
HAVING SUM(total_amount) >= 1000;
```

Output: The output will display the total amount spent by each customer, but only for customers who have spent $1000 or more.

```
| customer_id | total_spent |
|-------------|-------------|
| 1           | 1500        |
| 2           | 2000        |
| ...         | ...         |
```

Using WHERE and HAVING in One Statement

In SQL, you can use both the WHERE and HAVING clauses in a single query to filter both individual rows and groups based on specific conditions. The WHERE clause is applied before grouping, while the HAVING clause is applied after grouping.

Problem: You want to filter both individual rows and groups in your SQL query based on specific conditions applied before and after grouping.

Solution:

```
SELECT grouping_column,
       aggregate_function(column1) AS result1,
       aggregate_function(column2) AS result2
FROM table_name
WHERE condition1
GROUP BY grouping_column
HAVING condition2;
```

Explanation:

- Use the WHERE clause to filter individual rows based on conditions applied before grouping.
- Use the GROUP BY clause to specify the column by which you want to group the data.
- Apply aggregate functions to other columns in the SELECT statement to calculate summary statistics for each group.
- Use the HAVING clause to filter groups based on conditions applied after grouping.

Example: Consider a table named orders with columns customer_id and total_amount. You want to calculate the total amount spent by each customer and filter out customers who have made more than 5 orders and have spent less than $1000:

```sql
SELECT customer_id,
       COUNT(*) AS order_count,
       SUM(total_amount) AS total_spent
FROM orders
WHERE total_amount >= 1000
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

Output: The output will display the total amount spent by each customer who has made more than 5 orders and has spent $1000 or more.

```
| customer_id | order_count | total_spent |
|-------------|-------------|-------------|
| 1           | 6           | 1500        |
| 2           | 8           | 2000        |
| ...         | ...         | ...         |
```

Grouping and Sorting

Grouping and sorting data in SQL allows you to organize and summarize your results effectively. By combining GROUP BY and ORDER BY clauses, you can group your data based on specific criteria and then sort the grouped results accordingly.

Problem:

You want to group your data in SQL based on specific criteria and then sort the grouped results in a particular order.

Solution:

```
SELECT grouping_column,
       aggregate_function(column1) AS result1,
       aggregate_function(column2) AS result2
FROM table_name
GROUP BY grouping_column
ORDER BY sort_column1 [ASC|DESC], sort_column2 [ASC|DESC], ...;
```

Explanation:

- Use the GROUP BY clause to specify the column by which you want to group the data.
- Apply aggregate functions to other columns in the SELECT statement to calculate summary statistics for each group.
- Use the ORDER BY clause to specify the columns by which you want to sort the grouped results, along with the desired sort order (ascending or descending).

Example: Consider a table named orders with columns customer_id and total_amount. You want to calculate the total amount spent by each customer and sort the results in descending order of total amount:

```
SELECT customer_id,
       SUM(total_amount) AS total_spent
FROM orders
GROUP BY customer_id
ORDER BY total_spent DESC;
```

Output: The output will display the total amount spent by each customer, sorted in descending order of total amount.

```
| customer_id | total_spent |
|-------------|-------------|
| 2           | 2000        |
| 1           | 1500        |
| ...         | ...         |
```

Using PARTITION BY

Problem: In SQL, the PARTITION BY clause is used in conjunction with window functions to divide the result set into partitions to which the function is applied separately. This allows for performing calculations or aggregations within each partition independently.

You want to partition your data in SQL and apply calculations or aggregations within each partition separately.

Solution:

```
SELECT column1,
       column2,
       aggregate_function(column3) OVER (PARTITION BY
partition_column) AS result1,
       aggregate_function(column4) OVER (PARTITION BY
partition_column) AS result2
FROM table_name;
```

Explanation:

- Use the PARTITION BY clause followed by the column by which you want to partition the data.
- Apply aggregate functions or window functions to other columns in the SELECT statement.
- The function will be applied separately to each partition defined by the unique values in the partition column.

Example: Consider a table named orders with columns customer_id, order_date, and total_amount. You want to calculate the cumulative sum of total amount for each customer separately, based on the order date:

```
SELECT customer_id,
       order_date,
       total_amount,
       SUM(total_amount) OVER (PARTITION BY customer_id
ORDER BY order_date) AS cumulative_sum
FROM orders;
```

Output: The output will display the cumulative sum of total amount for each customer separately, based on the order date.

customer_id	order_date	total_amount	cumulative_sum
1	2023-01-01	500	500
1	2023-01-05	1000	1500
2	2023-01-02	750	750
2	2023-01-06	1250	2000
...	...	...	...

Working with Subqueries

Filtering by Subquery

Filtering by subquery in SQL involves using the result of a subquery as a condition to filter rows in the main query. This allows you to dynamically filter data based on the results of another query.

Problem: You want to filter rows in your SQL query based on the results of a subquery.

Solution:

```
SELECT column1, column2
FROM table_name
WHERE column3 IN (SELECT column3 FROM another_table WHERE
condition);
```

Explanation:

- Use a subquery enclosed in parentheses to generate a list of values or records.
- Use the result of the subquery as a condition in the WHERE clause of the main query.
- The main query will return rows where the specified column matches any value from the result of the subquery.

Example:

Consider two tables named products and orders. You want to retrieve products that have been ordered at least once:

```
SELECT product_id, product_name
FROM products
WHERE product_id IN (SELECT DISTINCT product_id FROM
orders);
```

Output: The output will display the product ID and product name of products that have been ordered at least once.

```
| product_id | product_name |
|------------|--------------|
| 1          | Product A    |
| 2          | Product B    |
| ...        | ...          |
```

Subqueries as Calculated Fields

Using subqueries as calculated fields in SQL involves incorporating the result of a subquery as a computed value within the main query. This allows you to derive additional information or perform complex calculations based on the results of another query.

Problem: You want to include the result of a subquery as a calculated field in your SQL query to derive additional information or perform complex calculations.

Solution:

```
SELECT column1,
       (SELECT aggregate_function(column2) FROM
another_table WHERE condition) AS calculated_field,
       column3
FROM table_name;
```

Explanation:

- Use a subquery enclosed in parentheses to perform a separate query within the SELECT statement.
- Include the subquery as a calculated field, specifying the desired aggregation or computation.
- The result of the subquery will be included as a computed value in the result set of the main query.

Example:

Consider a table named orders with columns order_id, customer_id, and order_date. You want to retrieve the total number of orders made by each customer:

```
SELECT customer_id,
       (SELECT COUNT(*) FROM orders WHERE customer_id =
customers.customer_id) AS total_orders,
       AVG(order_date) AS average_order_date
FROM customers
GROUP BY customer_id;
```

Output: The output will display the customer ID, total number of orders made by each customer, and the average order date for each customer.

customer_id	total_orders	average_order_date
1	5	2023-01-15
2	7	2023-01-20
...	...	...

Fully Qualified Column Names

Problem: You want to ensure clarity and avoid ambiguity when referencing columns in SQL queries, especially when using subqueries and dealing with multiple tables.

Solution:

```
SELECT table1.column1,
       (SELECT table2.column2 FROM another_table AS
table2 WHERE condition) AS calculated_field,
       table1.column3
FROM main_table AS table1;
```

Explanation:

- Use the table alias or table name along with the column name to fully qualify the column reference.
- When using subqueries, ensure that the columns referenced in the subquery are fully qualified to avoid ambiguity.
- Fully qualified column names help the database engine understand the context of each column reference, especially in queries involving multiple tables.

Example:

Consider a query involving multiple tables where you want to retrieve the customer name and the total amount spent by each customer:

```
SELECT customers.customer_name,
       (SELECT SUM(order_amount) FROM orders WHERE
orders.customer_id = customers.customer_id) AS
total_spent
FROM customers;
```

Output: The output will display the customer name and the total amount spent by each customer, with fully qualified column names ensuring clarity and avoiding ambiguity.

```
| customer_name | total_spent |
|---------------|-------------|
| John Doe      | 1500        |
| Jane Smith    | 2000        |
| ...           | ...         |
```

Joining Tables

Simple Equijoins

Joining tables in SQL involves combining rows from two or more tables based on a related column between them. Equijoins are the most common type of join, where rows are matched between tables based on equality of values in a specified column.

Problem: You want to retrieve data from multiple tables in SQL and combine them based on a related column with matching values.

Solution:

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

Explanation:

- Use the INNER JOIN clause to combine rows from two tables based on a related column with matching values.
- Specify the columns you want to retrieve from each table in the SELECT statement, prefixing them with the table alias or table name to avoid ambiguity.
- Use the ON keyword to specify the condition for the join, typically involving equality of values in the related columns.

Example

Consider two tables named orders and customers, where orders contain order information and customers contains customer information. You want to retrieve order details along with the corresponding customer information:

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

Output:

The output will display the order ID, order date, and customer name for each order, combining data from both the orders and customers tables based on the customer ID.

```
| order_id | order_date | customer_name |
|----------|------------|---------------|
| 1        | 2023-01-01 | John Doe      |
| 2        | 2023-01-05 | Jane Smith    |
| ...      | ...        | ...           |
```

Joining Tables: Inner Joins

Inner joins in SQL are used to combine rows from two or more tables based on a related column between them. Only the rows with matching values in the specified column(s) from both tables are included in the result set.

Problem: You want to retrieve data from multiple tables in SQL and combine them based on a related column, including only the rows with matching values in both tables.

Solution:

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

Explanation:

- Use the INNER JOIN clause to combine rows from two tables based on a related column with matching values.
- Specify the columns you want to retrieve from each table in the SELECT statement, prefixing them with the table alias or table name to avoid ambiguity.
- Use the ON keyword to specify the condition for the join, typically involving equality of values in the related columns.

Example: Consider two tables named orders and customers, where orders contains order information and customers contains customer information.

You want to retrieve order details along with the corresponding customer information:

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

Output: The output will display the order ID, order date, and customer name for each order, combining data from both the orders and customers tables based on the customer ID.

order_id	order_date	customer_name
1	2023-01-01	John Doe
2	2023-01-05	Jane Smith
...	...	...

Joining Multiple Tables

Joining multiple tables in SQL involves combining rows from more than two tables based on related columns between them. This allows you to retrieve data from multiple sources and create more comprehensive result sets.

Problem: You want to retrieve data from multiple tables in SQL and combine them based on related columns, including rows from more than two tables.

Solution:

```
SELECT t1.column1, t1.column2, t2.column1, t2.column2,
t3.column1, t3.column2
FROM table1 AS t1
INNER JOIN table2 AS t2 ON t1.related_column =
t2.related_column
INNER JOIN table3 AS t3 ON t2.related_column =
t3.related_column;
```

Explanation:

- Use the INNER JOIN clause to combine rows from two tables based on a related column with matching values.
- Specify the columns you want to retrieve from each table in the SELECT statement, prefixing them with the table alias or table name to avoid ambiguity.
- Join additional tables using INNER JOIN clauses, specifying the related columns for each join.

Example:

Consider three tables named orders, customers, and products, where orders contains order information, customers contains customer information, and products contains product information.

You want to retrieve order details along with the corresponding customer and product information:

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

Output: The output will display the order ID, customer name, and product name for each order, combining data from all three tables based on the related columns.

```
| order_id | customer_name | product_name |
|----------|---------------|--------------|
| 1        | John Doe      | Product A    |
| 2        | Jane Smith    | Product B    |
| ...      | ...           | ...          |
```

Using Table Aliases

Using table aliases in SQL is a common practice to simplify query syntax and improve readability, especially when dealing with multiple tables in a query. Table aliases provide shorthand names for tables, making it easier to reference columns and distinguish between them when joining tables or performing other operations.

Problem: You want to join multiple tables in SQL and use shorthand names (aliases) for the tables to improve query readability.

Solution:

```
SELECT t1.column1, t1.column2, t2.column1, t2.column2
FROM table1 AS t1
INNER JOIN table2 AS t2 ON t1.related_column =
t2.related_column;
```

Explanation:

- Use the AS keyword to assign a shorthand name (alias) to each table in the FROM clause.
- Refer to columns from each table using the assigned alias in the SELECT statement and join conditions.
- Table aliases provide a concise and readable way to reference tables, especially when working with multiple tables in a query.

Example:

Consider two tables named orders and customers, where orders contain order information and customers contains customer information.

You want to retrieve order details along with the corresponding customer information:

```
SELECT o.order_id, o.order_date, c.customer_name
FROM orders AS o
INNER JOIN customers AS c ON o.customer_id =
c.customer_id;
```

Output: The output will display the order ID, order date, and customer name for each order, using table aliases for clarity and readability.

order_id	order_date	customer_name
1	2023-01-01	John Doe
2	2023-01-05	Jane Smith
...	...	...

Multiple Uses of the Same Table

Using the same table multiple times in a SQL query, often referred to as self-join or multiple table instances, can be useful for comparing rows within the same table or for modeling hierarchical relationships.

Problem: You want to compare rows within the same table or model hierarchical relationships by using multiple instances of the same table in a SQL query.

Solution:

```
SELECT t1.column1, t2.column2
FROM table_name AS t1
INNER JOIN table_name AS t2 ON t1.related_column =
t2.related_column;
```

Explanation:

- Use table aliases to create multiple instances of the same table in the FROM clause.
- Assign unique aliases to each instance to distinguish between them.
- Join the table instances based on related columns, allowing for comparisons between rows within the same table.

Example:

Consider a table named employees that stores information about employees, including their manager.

You want to retrieve each employee's name along with the name of their manager:

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

Output: The output will display each employee's name along with the name of their manager, utilizing multiple instances of the employees table.

```
| employee_name | manager_name |
|---------------|--------------|
| John Doe      | Jane Smith   |
| Jane Smith    | Jane Smith   |
| ...           | ...          |
```

Select All Table Columns

When joining tables in SQL and you want to select all columns from both tables, you can use the asterisk (*) wildcard in the SELECT statement.

```
SELECT table1.*, table2.*
FROM table1
INNER JOIN table2 ON table1.related_column =
table2.related_column;
```

Explanation:

- Use the asterisk (*) wildcard to select all columns from each table.
- Specify the tables from which you want to select all columns followed by the wildcard.
- Join the tables based on the related column using INNER JOIN or any other appropriate join type.

Example:

Consider two tables named orders and customers.

You want to retrieve all columns from both tables for orders placed by customers:

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

Output: The output will display all columns from both the orders and customers tables for orders placed by customers.

Left Outer Joins

Left outer joins in SQL combine rows from two tables based on a related column, including all rows from the left table and matching rows from the right table. If there are no matching rows in the right table, NULL values are used for the columns from the right table.

```
SELECT *
FROM left_table
LEFT JOIN right_table ON left_table.related_column =
right_table.related_column;
```

Explanation:

- Use the LEFT JOIN clause to perform a left outer join.
- Specify the left table first, followed by LEFT JOIN, and then the right table.
- Define the join condition using ON, typically involving equality of values in the related columns.

Example:

Consider two tables named orders and customers.

You want to retrieve all orders along with the corresponding customer information, including orders without associated customers:

```
SELECT *
FROM orders
LEFT JOIN customers ON orders.customer_id =
customers.customer_id;
```

Output: The output will display all columns from both the orders and customers tables, including all orders and matching customer information. For orders without associated customers, NULL values will be displayed for the customer columns.

Right Outer Joins

Right outer joins in SQL combine rows from two tables based on a related column, including all rows from the right table and matching rows from the left table. If there are no matching rows in the left table, NULL values are used for the columns from the left table.

```
SELECT *
FROM left_table
RIGHT JOIN right_table ON left_table.related_column =
right_table.related_column;
```

Explanation:

- Use the RIGHT JOIN clause to perform a right outer join.
- Specify the right table first, followed by RIGHT JOIN, and then the left table.
- Define the join condition using ON, typically involving equality of values in the related columns.

Example:

Consider two tables named orders and customers. You want to retrieve all customers along with the corresponding order information, including customers without associated orders:

```
SELECT *
FROM orders
RIGHT JOIN customers ON orders.customer_id = customers.customer_id;
```

Output: The output will display all columns from both the orders and customers tables, including all customers and matching order information. For customers without associated orders, NULL values will be displayed for the order columns.

```
| order_id | customer_id | order_date | customer_name |
|----------|-------------|------------|---------------|
| 1        | 101         | 2023-01-01 | John Doe      |
| 2        | 102         | 2023-01-05 | Jane Smith    |
| NULL     | 201         | NULL       | Alice Johnson |
| NULL     | 202         | NULL       | Bob Williams  |
```

Full Outer Joins

Full outer joins in SQL combine rows from two tables based on a related column, including all rows from both tables. If there are no matching rows in one table, NULL values are used for the columns from the other table.

```
SELECT *
FROM left_table
FULL OUTER JOIN right_table ON left_table.related_column
= right_table.related_column;
```

Explanation:

- Use the FULL OUTER JOIN clause to perform a full outer join.
- Specify both tables and the join condition using ON, typically involving equality of values in the related columns.

Example:

Consider two tables named orders and customers.

You want to retrieve all orders and customers, including orders without associated customers and customers without associated orders:

```
SELECT *
FROM orders
FULL OUTER JOIN customers ON orders.customer_id =
customers.customer_id;
```

Output: The output will display all columns from both the orders and customers tables, including all orders and customers. For unmatched rows in either table, NULL values will be displayed for the columns from the other table.

```
| order_id | customer_id | order_date | customer_name |
|----------|-------------|------------|---------------|
| 1        | 101         | 2023-01-01 | John Doe      |
| 2        | 102         | 2023-01-05 | Jane Smith    |
| 3        | NULL        | NULL       | NULL          |
| NULL     | 201         | NULL       | Alice Johnson |
| NULL     | 202         | NULL       | Bob Williams  |
```

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.

```
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:

```
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.

```
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:

```
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.

```
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:

```
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.

```
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:

```
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.

```
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:

```
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.

```
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**:

```
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.

```
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:

```
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.

```
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:

```
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.

```
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:

```
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).

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

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

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

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

```
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).

```
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:

```
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:

```
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.

```
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:

```
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.

```
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:

```
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:

```
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:

```
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.

```
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:

```
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.

Reformatting Retrieved Data

In SQL, you can create views to reformat retrieved data, such as changing the display format of columns or combining multiple columns into a single field. This is achieved by using expressions in the SELECT clause to manipulate the data before presenting it in the view.

```
CREATE VIEW view_name AS
SELECT expression AS new_column_name
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 expressions to manipulate the retrieved data.
- Assign an alias using the AS keyword to the expressions to provide meaningful column names in the view.

Example:

Consider a scenario where you want to create a view named formatted_orders to display order details with a formatted date and total amount:

```
CREATE VIEW formatted_orders AS
SELECT order_id, TO_CHAR(order_date, 'YYYY-MM-DD') AS
formatted_date, CONCAT('$', total_amount) AS
formatted_amount
FROM orders;
```

In this example, the formatted_orders view is created to show order details with the order_date column formatted as 'YYYY-MM-DD' and the total_amount column prefixed with '$'. The view retrieves data from the orders table and applies the specified formatting expressions to present the data in the desired format.

In every line of code, they have woven a story of innovation and creativity. This book has been your compass in the vast world of SQL.

Close this chapter knowing that every challenge overcome is an achievement, and every solution is a step toward mastery.

Your code is the melody that gives life to projects. May they continue creating and programming with passion!

Thank you for allowing me to be part of your journey.

With gratitude,

Hernando Abella

Author of SQL Cookbook

Discover Other Useful Resources at:

www.hernandoabella.com