Module -3

Built-in Functions in SQL:
Single row multiple row ,
Number,
character,
date,
conversion function,
general functions Grouping/aggregate functions.

Built-in Functions in SQL

SQL provides a wide range of built-in functions that can be used to perform various tasks, such as manipulating data, performing calculations, and formatting output. These functions can be classified into the following categories:

- **Single row functions:** These functions operate on a single row of data and return a single result.
- **Multiple row functions:** These functions operate on a group of rows of data and return a single result for the entire group.
- **Number functions:** These functions perform mathematical operations on numbers.
- **Character functions:** These functions manipulate strings.
- **Date functions:** These functions manipulate dates and times.
- **Conversion functions:** These functions convert data from one type to another.
- **General functions:** These functions perform various tasks, such as comparing values and generating random numbers.

Examples of Built-in Functions

Here are some examples of built-in functions in SQL:

Single row functions:

- **UPPER():** Converts a string to uppercase.
- **LOWER():** Converts a string to lowercase.
- **LENGTH():** Returns the length of a string.
- **SUBSTR():** Extracts a substring from a string.
- **SUM():** Returns the sum of the values in a column.
- **AVG():** Returns the average of the values in a column.
- **MIN():** Returns the minimum value in a column.
- **MAX():** Returns the maximum value in a column.
- **TO_CHAR():** Converts a date or number to a string.
- **TO_DATE():** Converts a string to a date.

Multiple row functions:

- **GROUP BY():** Groups the rows in a table based on the values in a column or columns.
- **HAVING():** Filters the groups of rows returned by the GROUP BY clause.
- **COUNT():** Returns the number of rows in a table or group of rows.
- **RANK():** Returns the rank of a row within a group of rows.
- **ROW_NUMBER():** Returns the row number of a row within a group of rows.
- **LAG():** Returns the value of a column in the previous row of a group of rows.

Number functions:

- **ABS():** Returns the absolute value of a number.
- **ROUND():** Rounds a number to a specified number of decimal places.

- **TRUNC():** Truncates a number to a specified number of decimal places.
- **SQRT():** Returns the square root of a number.
- **POWER():** Returns the power of a number to a specified exponent.

Character functions:

- **CONCAT():** Concatenates two strings.
- **SUBSTR():** Extracts a substring from a string.
- **INSTR():** Returns the position of a substring within a string.
- **TRIM():** Removes whitespace from the beginning and end of a string.
- **REPLACE():** Replaces a substring in a string with another substring.

Date functions:

- **ADD_MONTHS():** Adds a specified number of months to a date.
- **SUB_MONTHS():** Subtracts a specified number of months from a date.
- **LAST_DAY():** Returns the last day of the month for a given date.
- **EXTRACT():** Extracts a part of a date, such as the year, month, or day.
- **TO_DATE():** Converts a string to a date.

Conversion functions:

- **TO_CHAR():** Converts a date or number to a string.
- **TO_DATE():** Converts a string to a date.
- **TO_NUMBER():** Converts a string to a number.
- **TO_TIMESTAMP():** Converts a string to a timestamp.

General functions:

- **IF():** Returns one value if a condition is true and another value if the condition is false.
- **CASE():** Returns a value based on a set of conditions.
- **GREATEST():** Returns the greatest value from a set of values.
- **LEAST():** Returns the least value from a set of values.
- **RAND():** Generates a random number.

Using Built-in Functions

Built-in functions can be used in SQL statements to perform a variety of tasks. For example, you can use the UPPER() function to convert a column of customer names to uppercase, or you can use the SUM() function to calculate the total sales for a given product.

To use a built-in function in a SQL statement, you simply need to specify the function name and the arguments to the function. The arguments to the function can be column names, constants, or other expressions

Grouping data from Tables in SQL:

To group data from tables in SQL, you can use the GROUP BY clause. The GROUP BY clause groups the rows in a table based on the values in a column or columns. You can then use aggregate functions to calculate summary values for each group of rows.

For example, the following SQL statement groups the customers in the customers table by country and calculates the total sales for each country:

```
SELECT country, SUM(sales) AS total_sales
FROM customers
GROUP BY country
ORDER BY total_sales DESC;
```

This statement will return a list of countries, along with the total sales for each country. The countries are ordered by total sales in descending order.

You can also group data from multiple tables in SQL. To do this, you can use a JOIN clause to join the tables together. Once the tables are joined, you can use the GROUP BY clause to group the rows based on the values in a column or columns from any of the tables.

For example, the following SQL statement joins the customers and orders tables and groups the rows by country and product category. The statement then calculates the total sales for each country and product category:

```
SELECT country, product_category, SUM(order_total) AS total_sales
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
GROUP BY country, product_category
ORDER BY total_sales DESC;
```

This statement will return a list of countries and product categories, along with the total sales for each country and product category. The countries and product categories are ordered by total sales in descending order.

Examples

Here are some more examples of how to use the GROUP BY clause to group data from tables in SQL:

SQL

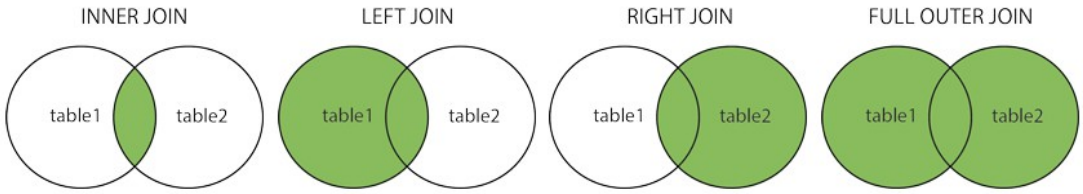
```
# Group the customers in the `customers` table by state and calculate the average customer order value for each state.

SELECT state, AVG(order_total) AS average_order_value
FROM customers
GROUP BY state
ORDER BY average_order_value DESC;

# Group the products in the `products` table by category and calculate the total sales for each category.

SELECT category, SUM(sales) AS total_sales
FROM products
GROUP BY category
ORDER BY total_sales DESC;

# Group the orders in the `orders` table by customer ID and calculate the number of orders placed by each customer.

SELECT customer_id, COUNT(*) AS num_orders
FROM orders
GROUP BY customer_id
ORDER BY num_orders DESC;

# Join the `customers` and `orders` tables and group the rows by country and product category. Calculate the total sales for each country and product category.

SELECT country, product_category,
  SUM(order_total) AS total_sales
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
GROUP BY country, product_category
ORDER BY total_sales DESC;
```

Joins, Joining Multiple Tables, different types of joins:

Joins are a way to combine data from two or more tables based on a common field between them. This can be useful for a variety of tasks.Joins are a powerful tool for combining data from multiple tables in SQL. By using joins, you can access and analyze data in new and powerful ways.

Different Types of Joins

- **INNER JOIN:** This is the most common type of join. It is used to return all rows from both tables where the common field matches in both tables.
-
- SELECT customers.name, orders.product_name
- FROM customers
- INNER JOIN orders ON customers.customer_id = orders.customer_id;
- **LEFT JOIN:** This join is used to return all rows from the left table, even if there is no matching row in the right table. This can be useful for tasks such as finding all customers who have not placed any orders.
-
- SELECT customers.name, orders.product_name
- FROM customers
- LEFT JOIN orders ON customers.customer_id = orders.customer_id;
-
- **RIGHT JOIN:** This join is the opposite of a LEFT JOIN. It is used to return all rows from the right table, even if there is no matching row in the left table. This can be useful for tasks such as finding all products that have not been ordered by any customers.
-
- **FULL JOIN:** This join is used to return all rows from both tables, regardless of whether there is a matching row in the other table. This can be useful for tasks such as generating a complete list of all customers and products.
-



Sub queries, Types of subqueries:
Subqueries are queries that are nested inside of another query. They can be used to perform a variety of tasks, such as:

-

Types of Subqueries

There are two main types of subqueries:

- **Correlated subqueries:** Correlated subqueries reference a column from the outer query. This type of subquery can be inefficient, so it is important to use it carefully.
- **Non-correlated subqueries:** Non-correlated subqueries do not reference any columns from the outer query. This type of subquery is more efficient than correlated subqueries, and it is generally the preferred type of subquery to use.

Here is an example of a correlated subquery:
SELECT *

```
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_date > '2023-10-01');
```

This query will return all customers who have placed an order in the last month. The correlated subquery SELECT customer_id FROM orders WHERE order_date > '2023-10-01' returns a list of customer IDs who have placed an order in the last month. The outer query then filters the results to only include customers who are in this list.

Here is an example of a non-correlated subquery:

```
SELECT *
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders LIMIT 10);
```

This query will return the first 10 customers in the customers table. The non-correlated subquery SELECT customer_id FROM orders LIMIT 10 returns a list of the first 10 customer IDs in the orders table. The outer query then filters the results to only include customers who are in this list.

Single row subquery:
A single-row subquery is a subquery that returns zero or one row to the outer query. Single-row subqueries can be used in the WHERE clause, HAVING clause, or SELECT clause of a SQL statement.

Single-row subqueries are often used to compare the value of a column in the outer query to the value of a column in the subquery. For example, the first query above compares the customer_id column in the outer query to the customer_id column in the subquery to find the customer with the highest order value.

Single-row subqueries can also be used to perform complex calculations. For example, the second query above calculates the average product price and then compares the price of each product to the average price to find all products that are more expensive than the average product price.

Here are some examples of single-row subqueries:

```
# Find the customer with the highest order value.
SELECT *
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders ORDER BY order_total DESC LIMIT 1);

# Find all products that are more expensive than the average product price.
SELECT *
FROM products
WHERE price > (SELECT AVG(price) FROM products);

# Find all orders that were placed before or on the same day as the current date.
SELECT *
FROM orders
WHERE order_date <= CURRENT_DATE;
```

Benefits of Using Single-Row Subqueries

- **1.Improved performance:** Single-row subqueries are often more efficient than multiple-row subqueries because they only return one row to the outer query.
- **2.Simplified code:** Single-row subqueries can make code more readable and easier to maintain.
- **3.Increased flexibility:** Single-row subqueries can be used to perform a variety of tasks that would be difficult or impossible to do without them.

Multiple row subquery:
A multiple-row subquery is a subquery that returns more than one row to the outer query. Multiple-row subqueries can be used in the WHERE clause, HAVING clause, or SELECT clause of a SQL statement.

Multiple-row subqueries are often used to compare the value of a column in the outer query to a list of values returned by the subquery. For example, the first query above compares the customer_id column in the outer query to a list of customer IDs returned by the subquery to find all customers who have placed an order in the last month.

Multiple-row subqueries can also be used to perform complex calculations. For example, the second query above groups the orders by product ID and then counts the number of orders for each product. The subquery then returns the product IDs for all products that have been ordered by more than one customer. The outer query then filters the results to only include products with these product Ids.

Here are some examples of multiple-row subqueries:

```
# Find all customers who have placed an order in the last month.
SELECT *
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_date > '2023-10-01');

# Find all products that have been ordered by more than one customer.
SELECT *
FROM products
WHERE product_id IN (SELECT product_id FROM orders GROUP BY product_id HAVING
COUNT(*) > 1);

# Find all orders that were placed before or on the same day as the current date and have a total value of
more than $100.
SELECT *
FROM orders
WHERE order_date <= CURRENT_DATE AND order_total > 100;
```

Benefits of Using Multiple-Row Subqueries

Multiple-row subqueries can offer a number of benefits, including:

- **1.Increased flexibility:** Multiple-row subqueries can be used to perform a variety of tasks that would be difficult or impossible to do without them.
- **2.Simplified code:** Multiple-row subqueries can make code more readable and easier to maintain.

Top N Analysis:
Top-N analysis is a data analysis technique that identifies the top or bottom N values in a dataset. It is a common technique used in business intelligence and data mining to identify trends, patterns, and outliers.

Top-N analysis can be performed on any type of data, but it is most commonly used on numerical data. For example, you could use top-N analysis to identify the top 10 customers by revenue, the top 5 products by sales, or the top 3 regions by website traffic.

Top-N analysis can be performed using a variety of tools, including SQL databases, data mining software, and spreadsheet programs. Most SQL databases have a built-in function for top-N analysis, such as the TOP() function in Microsoft SQL Server or the LIMIT() clause in MySQL.

Here is an example of a top-N analysis query in SQL:

```
SELECT customer_name, order_total
FROM orders
ORDER BY order_total DESC
LIMIT 10;
```

DML Statements:

DML (Data Manipulation Language) statements are used to insert, update, delete, and merge data in SQL databases.

INSERT statements are used to insert new rows into a table.

Syntax:

```
INSERT INTO table_name (column_name1, column_name2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO customers (name, email, phone_number)
VALUES ('John Doe', 'john.doe@example.com', '123-456-7890');
```

UPDATE statements are used to update existing rows in a table.

Syntax:

```
UPDATE table_name
SET column_name1 = value1, column_name2 = value2, ...
WHERE condition;
```

Example:

```
UPDATE customers
SET email = 'jane.doe@example.com'
WHERE customer_id = 1;
```

DELETE statements are used to delete rows from a table.

Syntax:

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

Example:

```
DELETE FROM customers
WHERE customer_id = 1;
```

MERGE statements are used to insert, update, or delete rows in a table based on a condition.

Syntax:

```
MERGE INTO table_name
USING table2 ON table_name.column_name = table2.column_name
WHEN MATCHED THEN UPDATE SET column_name1 = value1, column_name2 = value2, ...
WHEN NOT MATCHED THEN INSERT (column_name1, column_name2, ...) VALUES (value1, value2, ...);
```

Example:

```
MERGE INTO customers
USING new_customers ON customers.customer_id = new_customers.customer_id
```

```
WHEN MATCHED THEN UPDATE SET name = new_customers.name, email = new_customers.email
WHEN NOT MATCHED THEN INSERT (name, email) VALUES (new_customers.name,
new_customers.email);
```

This MERGE statement will update the name and email address of any existing customers in the customers table that match customers in the new_customers table. It will also insert any new customers from the new_customers table that do not match any existing customers in the customers table.

DML statements are a powerful tool for managing data in SQL databases. By using DML statements, you can insert, update, delete, and merge data to meet the needs of your application.

Oracle data types:

Oracle data types are used to define the type of data that can be stored in a column in an Oracle database table. Oracle provides a variety of data types, including:

- **Numeric data types:** These data types are used to store numeric data, such as integers, decimals, and floating-point numbers. Examples of numeric data types include NUMBER, FLOAT, and DOUBLE.
- **Character data types:** These data types are used to store character data, such as strings and text. Examples of character data types include CHAR, VARCHAR2, and CLOB.
- **Date and time data types:** These data types are used to store date and time data. Examples of date and time data types include DATE, TIME, and TIMESTAMP.
- **Binary data types:** These data types are used to store binary data, such as images and video files. Examples of binary data types include BLOB and BFILE.
- **Specialized data types:** These data types are used to store specialized data, such as spatial data and object-relational data. Examples of specialized data types include SDO_GEOMETRY and XMLType.

Choosing the Right Oracle Data Type

When choosing a data type for a column, it is important to consider the following factors:

- **The type of data that will be stored in the column:** You should choose a data type that is appropriate for the type of data that will be stored in the column. For example, you should use a numeric data type to store numeric data and a character data type to store character data.
- **The size of the data that will be stored in the column:** You should choose a data type that is large enough to store the data that will be stored in the column. For example, if you need to store a long string of text, you should use a CLOB data type.
- **The performance requirements of your application:** You should choose a data type that will support the performance requirements of your application. For example, if your application needs to be able to quickly query large amounts of data, you should choose a data type that is optimized for querying.

Examples of Oracle Data Types

Here are some examples of Oracle data types and how they can be used:

- **NUMBER:** The NUMBER data type is used to store numeric data, such as integers, decimals, and floating-point numbers. For example, the following column definition uses the NUMBER data type to store the customer ID:

```
CREATE TABLE customers (
  customer_id NUMBER(10)
);
```

- **VARCHAR2:** The VARCHAR2 data type is used to store variable-length character data, such as strings and text. For example, the following column definition uses the VARCHAR2 data type to store the customer name:

```
CREATE TABLE customers (
```

```
  customer_name VARCHAR2(255)
);
```

- **DATE:** The DATE data type is used to store date and time data. For example, the following column definition uses the DATE data type to store the customer's birth date:

```
CREATE TABLE customers (
  birth_date DATE
);
```

- **BLOB:** The BLOB data type is used to store binary data, such as images and video files. For example, the following column definition uses the BLOB data type to store the customer's profile picture:

```
CREATE TABLE customers (
  profile_picture BLOB
);
```

TCL Command: commit, rollback and savepoint:

TCL Commands: commit, rollback, and savepoint are used to manage transactions in SQL databases. A transaction is a unit of work that is performed against a database. Transactions are used to ensure that data is consistent and up-to-date.

COMMIT

The COMMIT command is used to permanently save all changes made to a database since the last commit. Once a commit has been issued, the changes cannot be undone.

ROLLBACK

The ROLLBACK command is used to undo all changes made to a database since the last commit or savepoint. If a rollback is issued, the database will be restored to its state before the transaction began.

SAVEPOINT

The SAVEPOINT command is used to create a named point in time in a transaction. By creating a savepoint, you can rollback to that point in time later in the transaction. This can be useful if you need to undo a subset of the changes made in a transaction.

Example

The following example shows how to use the COMMIT, ROLLBACK, and SAVEPOINT commands:

```
BEGIN TRANSACTION;

-- Make some changes to the database.

SAVEPOINT savepoint1;

-- Make some more changes to the database.

IF something_goes_wrong THEN
  ROLLBACK TO SAVEPOINT savepoint1;
ELSE
  COMMIT;
END IF;
```

This example starts a transaction and then makes some changes to the database. Next, it creates a savepoint named savepoint1. Then, it makes some more changes to the database. If something goes wrong, the code will rollback to the savepoint savepoint1. Otherwise, the code will commit the changes.

When to Use TCL Commands

TCL commands are typically used in database applications to manage transactions. Here are some examples of when to use TCL commands:

- To ensure that data is consistent and up-to-date.
- To improve the performance of database applications.
- To undo changes made to a database if something goes wrong.

Conclusion

TCL commands are a powerful tool for managing transactions in database applications. By using TCL commands, database applications can ensure that data is consistent and up-to-date, and they can improve their performance.