Basic SQL Queries

LESSON

Generate a quiz about this unit

Summarize this unit

Have any doubt about this content? Ask!

Learning Objectives

By the end of this lesson, you will be able to:

- Interpret tables by utilizing the `SELECT ... FROM` statement.
- Identify and select `UNIQUE` values from a table.
- Apply techniques to change the format of variables within a table.
- Analyze data in a table and sort it using the `ORDER BY` clause.
- Implement data aggregation in a table using the `GROUP BY` and `HAVING` clauses.
- Demonstrate the ability to filter tables effectively using the `WHERE` clause.
- Utilize various operators such as `IN`, `BETWEEN`, `LIKE`, `AND`, `OR`, `=`, `>=` to manipulate and filter data.
- Construct and apply conditions within SQL queries using the `CASE` statement.
- Create new calculated variables for enhanced data analysis and manipulation.

Read tables with SELECT statement

The most fundamental SQL command is the `SELECT` statement, which retrieves data from a database.

```sql
SELECT * FROM tablename;
```

Copy

Explain this code

This gets every row and column from the table *tablename*. The asterisk means “all the columns”.

If we want to select just a few columns we would do:

```sql
SELECT column1, column2
FROM tablename;
```

Copy

Explain this code

Example:

Suppose we have a table `students` with columns `id`, `name`, and `age`.

Table `students`:

id	name	age
1	Alice	20
2	Bob	22

3	Charlie	21
4	David	23

To fetch all names from the `students` table:

```
SELECT name FROM students;
```

Copy

Explain this code

Result:

name
Alice
Bob
Charlie
David

Select only the UNIQUE values

When querying data from a table, you may get duplicate rows. To remove these duplicate rows (to get unique values in the output), you use the `DISTINCT` clause in the `SELECT` statement.

```
SELECT DISTINCT columnname
FROM tablename;
```

Copy

Explain this code

Example:

Table `students` (same as above).To fetch unique ages from the `students` table:

```
SELECT DISTINCT age FROM students;
```

Copy

Explain this code

Result:

age
20
22
21
23

Aggregate functions

To perform calculations on a set of values and return a single value, MySQL provides various aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.

```
SELECT AGGREGATE_FUNCTION(columnname)
FROM tablename;
```

Copy

Example:

Table `students` (same as above).

To find the average age of students:

```
SELECT AVG(age) FROM students;
```

Copy

Explain this code

Result:

AVG(age)
21.5

ORDER BY

When you use the `SELECT` statement to query data from a table, the order of rows in the result set is unspecified. To sort the rows in the result set, you add the `ORDER BY` clause in the `SELECT` statement.

You can also specify the direction of the order by adding the command `ASC` or `DESC`.

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

Copy

Explain this code

The `ORDER BY` clause in SQL allows you to sort the result set based on multiple columns. This is especially useful when you want to have a primary sort criterion and then a secondary sort criterion (and so on) for rows that have the same value in the primary sort column.

To sort the result set based on multiple columns:

```
SELECT column1, column2, ...
FROM tablename
ORDER BY columnname1 [ASC|DESC], columnname2 [ASC|DESC], ...;
```

Copy

Explain this code

When using multiple columns in the `ORDER BY` clause:

1. The result set is first sorted by the first column.
2. If there are rows with duplicate values in the first column, the result set sorts those rows based on the second column, and so forth.

Example:

Table `students`: same as before.

id	name	age
1	Alice	20
2	Bob	22
3	Charlie	21
4	David	23

```sql
SELECT name FROM students ORDER BY age DESC;
```

Copy

Explain this code

In this case, the output will be sorted by age, so from the oldest to the youngest.

Result:

name
David
Bob
Charlie
Alice

Example:

Suppose we have the `students` table, and we want to order the students primarily by their age and then, for students of the same age, alphabetically by name.

```sql
SELECT name, age
FROM students
ORDER BY age DESC, name ASC;
```

Copy

Explain this code

In this example:

1. The result set is first sorted in descending order by age.
2. For students who have the same age, their names are then sorted in ascending alphabetical order.

GROUP BY

The `GROUP BY` clause is an essential aspect of SQL, especially when you're working with aggregate functions.

The `GROUP BY` clause groups rows that have the same values in specified columns into aggregate data, such as the sum, average, minimum, maximum, etc. It's often used with aggregate functions to perform an operation on each group of rows.

```sql
SELECT column1, AGGREGATE_FUNCTION(column2)
FROM tablename
GROUP BY column1;
```

Copy

Explain this code

Key Points:

1. **Grouping**: When using `GROUP BY`, the data is divided into groups based on the values in the specified column(s). Each group represents one or more rows of the table.
2. **Aggregate Functions**: Once the data is grouped, you can use an aggregate function (like `SUM`, `AVG`, `COUNT`, etc.) to perform a calculation on each group. The result is a single value per group.
3. **Multiple Columns**: You can group by more than one column by listing multiple column names separated by commas. This creates groups based on the unique combinations of values in the specified columns.

Example:

Table: sales

product	region	amount_sold	year
Laptop	East	100	2022
Phone	West	150	2022
Laptop	West	80	2022
Laptop	North	90	2021
Phone	South	70	2021

If we want to determine the total sales amount for each product:

```sql
SELECT product, SUM(amount_sold) as total_sold
FROM sales
GROUP BY product;
```

Copy

Explain this code

Resulting Table:

product	total_sold
Laptop	270
Phone	220

Example:

To ascertain the total sales amount for each product in every region:

```sql
SELECT product, region, SUM(amount_sold) as total_sold
FROM sales
GROUP BY product, region;
```

Copy

Explain this code

Resulting Table:

product	region	total_sold
Laptop	East	100
Laptop	West	80
Laptop	North	90
Phone	West	150
Phone	South	70

Note that for the data we have this example doesn’t make much sense, but this is how we would do it if we wanted to group by product and region.

WHERE clause for filtering data

The `WHERE` clause is a fundamental part of SQL that allows you to filter records based on specific conditions, ensuring that only certain rows are selected or affected by a SQL statement.

```
FROM tablename
WHERE condition;
```

Explain this code

Key Points:

1. **Conditions**: The condition in the `WHERE` clause can involve comparisons using operators like `=`, `<>` or `!=`, `<`, `>`, `<=`, `>=`.
2. **Logical Operators**: You can combine multiple conditions using logical operators like `AND`, `OR`, and `NOT`.
3. **Functions and Expressions**: The `WHERE` clause can also contain functions, arithmetic calculations, and other expressions.

Example:

Using the same `sales` table with columns `product`, `region`, `amount_sold`, and `year`:

Table: sales

product	region	amount_sold	year
Laptop	East	100	2022
Phone	West	150	2022
Laptop	West	80	2022
Laptop	North	90	2021
Phone	South	70	2021

To fetch all sales records for the product “Laptop”:

Copy

```
SELECT *
FROM sales
WHERE product = 'Laptop';
```

Explain this code

Resulting Table:

product	region	amount_sold	year
Laptop	East	100	2022
Laptop	West	80	2022
Laptop	North	90	2021

Example

To fetch sales records for the product “Laptop” in the region “West” during the year 2022:

Copy

```
SELECT *
FROM sales
WHERE product = 'Laptop' AND region = 'West' AND year = 2022;
```

Explain this code

Resulting Table:

Laptop	West	80	2022

When combined with other SQL clauses, the order of operation is essential:

1. The `WHERE` clause filters the rows on the given condition.
2. If there's a `GROUP BY` clause, the filtered rows are then grouped.
3. Aggregate functions are applied.
4. The `HAVING` clause filters the groups (if present) - we will explain this below.
5. Finally, the `ORDER BY` clause sorts the result set.

For instance, using the `sales` table:

To get the total sales amount for each product in 2022:

```
SELECT product, SUM(amount_sold) as total_sold
FROM sales
WHERE year = 2022
GROUP BY product;
```

Copy

Explain this code

In this example, the `WHERE` clause first filters out the records not from the year 2022. The remaining records are then grouped by product, and the total sales amount for each product is computed.

Resulting Table:

product	total_sold
Laptop	180
Phone	150

HAVING clause

The `HAVING` clause is a special clause in SQL that allows you to filter the result sets of a query after they have been grouped by a `GROUP BY` clause. It's specifically used to filter aggregate results.

While the `WHERE` clause filters rows before they're aggregated, the `HAVING` clause filters after the aggregation.

```
SELECT column1, AGGREGATE_FUNCTION(column2)
FROM tablename
GROUP BY column1
HAVING condition;
```

Copy

Explain this code

Key Points:

1. **Post-Aggregation Filtering**: The primary purpose of the `HAVING` clause is to allow filtering after the `GROUP BY` operation. This is particularly useful when you want to include/exclude certain groups based on an aggregate criterion.

2. **Works with Aggregate Functions**: Conditions in the `HAVING` clause often involve aggregate functions like `SUM()`, `AVG()`, `COUNT()`, etc.

3. **Not a Substitute for WHERE**: It's important to note that `HAVING` isn't a replacement for the `WHERE` clause. Instead, they serve different purposes. If you need to filter individual rows before aggregation, use the `WHERE` clause.

Example

Using the previously mentioned `sales` table with columns `product`, `region`, `amount_sold`, and `year`:

```sql
SELECT product, SUM(amount_sold) as total_sold
FROM sales
GROUP BY product
HAVING total_sold > 100;
```

Copy

Explain this code

In this example:

- The `GROUP BY` clause groups the sales by `product`.
- The `SUM(amount_sold)` calculates the total amount sold for each product.
- The `HAVING` clause then filters out groups (i.e., products) that have a total sold amount of 100 or less.

Resulting Table:

product	total_sold
Laptop	270
Tablet	220

Combining with WHERE:

You can use both `WHERE` and `HAVING` in the same query. Remember:

- `WHERE` filters rows before aggregation.
- `HAVING` filters groups after aggregation.

For instance:

To get the total sales amount for each product in 2022, but only include those products with a total sold amount greater than 150:

```sql
SELECT product, SUM(amount_sold) as total_sold
FROM sales
WHERE year = 2022
GROUP BY product
HAVING total_sold > 150;
```

Copy

Explain this code

In this example, the `WHERE` clause first filters out the records not from the year 2022. The remaining records are then grouped by product, the total sales amount for each product is computed, and finally, the `HAVING` clause filters out products that don’t meet the sales threshold.

Resulting Table:

product	total_sold
Laptop	180

LIMIT Clause

To limit the number of rows returned:

```sql
SELECT column1, column2, ...
FROM tablename
LIMIT number;
```

Copy

Explain this code

Example:

To fetch the first three sales:

```
SELECT * FROM sales LIMIT 3;
```

Copy

Explain this code

Resulting Table:

product	region	amount_sold	year
Laptop	East	100	2022
Phone	West	150	2022
Laptop	West	80	2022

CASE Expression in SQL

The `CASE` expression provides a way to introduce conditional decision-making logic directly within SQL queries. It's akin to "if-then-else" logic in other programming languages.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE result_else
END AS alias_name
```

Copy

Explain this code

- **WHEN**: Specifies a condition to check.
- **THEN**: Provides the result to be returned if the condition is true.
- **ELSE**: (optional) Specifies the result to return if none of the `WHEN` conditions are met.
- **END**: Closes the `CASE` statement.
- **AS alias_name**: Gives a name to the new column formed as a result of the `CASE` expression.

Example with the `sales` table:

Imagine we want to categorize products based on their `amount_sold`:

- High Sales: `amount_sold` greater than 200
- Moderate Sales: `amount_sold` between 100 and 200
- Low Sales: `amount_sold` less than 100

Using the `sales` table with columns `product`, `region`, `amount_sold`, and `year`:

```
SELECT product, region, amount_sold, year,
CASE
    WHEN amount_sold > 200 THEN 'High Sales'
    WHEN amount_sold >= 100 AND amount_sold <= 200 THEN 'Moderate Sales'
    ELSE 'Low Sales'
END AS sales_category
FROM sales;
```

Copy

Explain this code

- Depending on the value, it categorizes the sales as either ‘High Sales’, ‘Moderate Sales’, or ‘Low Sales’.
- The result of this categorization is displayed in a new column named `sales_category`.

Given the `sales` table:

product	region	amount_sold	year
Laptop	East	100	2022
Phone	West	150	2022
Laptop	West	80	2022
Laptop	North	90	2021
Phone	South	70	2021

When executing the provided SQL query to categorize products based on their sales amount:

Resulting Table:

product	region	amount_sold	year	sales_category
Laptop	East	100	2022	Moderate Sales
Phone	West	150	2022	Moderate Sales
Laptop	West	80	2022	Low Sales
Laptop	North	90	2021	Low Sales
Phone	South	70	2021	Low Sales

FORMAT Function in SQL

The `FORMAT` function is primarily used to format numbers and dates to make them more readable or to adhere to specific local or business conventions.

1. Formatting Numbers

In MySQL, the `FORMAT` function primarily formats numbers by grouping thousands and controlling the number of decimal places displayed.

Syntax:

```
FORMAT(number, decimal_places, [locale])
```

Copy

Explain this code

- **number**: The number to be formatted.
- **decimal_places**: The number of decimal places to be returned.
- **locale** (optional): The locale to be used for formatting (e.g., ‘en_US’).

Example:

To format the number 1234567.89 to two decimal places with a comma as the thousand separator:

```
SELECT FORMAT(1234567.89, 2);  -- Returns '1,234,567.89'
```

Copy

Explain this code

2. Formatting Dates

Syntax:

```
DATE_FORMAT(date, format_specifiers)
```

Copy

Explain this code

- **date**: The date value to be formatted.
- **format_specifiers**: The format pattern using specific specifiers (e.g., `%Y` for year, `%m` for month, `%d` for day).

Example:

Assuming a table `sales` has a `sale_date` column:

Table: sales with sale_date

product	region	amount_sold	sale_date
Laptop	East	100	2022-06-15
Phone	West	150	2022-09-20

If we want to format the date to display in the format "Month Day, Year", we can use:

```
SELECT DATE_FORMAT(sale_date, '%M %d, %Y') AS formatted_date
FROM sales;
```

Copy

Explain this code

Resulting Table:

formatted_date
June 15, 2022
September 20, 2022

Considerations for Other Databases:

- **SQL Server**: The `FORMAT` function can be used for both numbers and dates. For example, `FORMAT(some_date, 'yyyy-MM-dd')` to format a date.
- **Oracle**: Oracle uses the `TO_CHAR` function for both number and date formatting. For instance, `TO_CHAR(some_date, 'YYYY-MM-DD')` for dates.
- **PostgreSQL**: PostgreSQL uses the `TO_CHAR` function similarly to Oracle for date and number formatting.

SQL Operators

SQL operators are symbols or keywords used to perform operations on data in SQL queries. They help manipulate and compare data, filter rows, and perform calculations. Here are some common types of SQL operators:

Arithmetic Operators:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` or `MOD` Modulus (division remainder)

Comparison Operators:

- `=` Equal to
- `!=` or `<>` Not equal to
- `<` Less than
- `>` Greater than
- `<=` Less than or equal to
- `>=` Greater than or equal to

Logical Operators:

- AND: Logical AND
- OR: Logical OR
- NOT: Logical NOT

Concatenation Operator:

- `||` or `CONCAT()`: Concatenates strings together

Pattern Matching Operators:

- `LIKE`: Matches a pattern in text (supports wildcards)
- `IN`: Checks if a value exists in a list of values

NULL Comparison Operators:

- `IS NULL`: Checks if a value is NULL
- `IS NOT NULL`: Checks if a value is not NULL

Between Operator:

- `BETWEEN`: Checks if a value falls within a range

Order of Writing and Executing SQL Clauses

Order of Writing SQL Clauses

The order in which you write SQL clauses and the order in which they are executed by the database engine are two different things. This distinction is important for understanding how SQL processes your requests.

When constructing an SQL statement, the clauses need to be written in a specific order:

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`
7. `LIMIT`

For example:

Copy

```
FROM tablename
WHERE condition_on_column
GROUP BY column1
HAVING condition_on_aggregated_column
ORDER BY column1
LIMIT number;
```

Explain this code

Order of Execution of SQL Clauses

The SQL engine does not execute clauses in the order they are written. Instead, it follows a specific sequence to process the data:

1. `FROM`: It identifies the table(s) from which data is to be retrieved.
2. `WHERE`: Filters the rows based on the condition specified. It acts on the result of the `FROM` clause.
3. `GROUP BY`: It groups rows that have the same values in specified columns into aggregate data.
4. `HAVING`: Filters the groups based on a condition. It acts on the result of the `GROUP BY` clause.
5. `SELECT`: Selects columns to be displayed in the final result. If there are any aggregate functions, they are computed here.
6. `ORDER BY`: Sorts the result set based on one or more columns.
7. `LIMIT`: Limits the number of rows to be returned in the result set.

Why This Matters:

Understanding the order of execution is crucial for several reasons:

- **Predictability**: Knowing how SQL processes the data helps you predict the outcome of your queries. For instance, the `WHERE` clause filters rows before they are grouped by the `GROUP BY` clause.
- **Performance**: Optimizing the `WHERE` clause can significantly reduce the number of rows that need to be grouped, aggregated, or sorted in subsequent steps, leading to faster query performance.
- **Debugging**: If your query isn't returning what you expect, understanding the order of execution can help pinpoint where things are going awry.

Summary

This lesson introduces fundamental SQL commands and concepts essential for data retrieval and manipulation. You have seen how to use SELECT statements, aggregate functions, and the GROUP BY clause for organizing data, along with the WHERE and HAVING clauses for filtering data. The lesson also covers formatting numbers and dates with functions like FORMAT and DATE_FORMAT in MySQL. Additionally, you have learned various SQL operators for arithmetic, comparison, and logical operations. Finally you have seen that the order of writing and executing SQL clauses is crucial for optimized query performance and effective debugging.

Any doubt? Ask our AI Chatbot!

✓

Lesson Completed

Mark as not completed

How would you rate the content of this unit?

😞 😕 😐 😊 😍