DATABASES

GS WEB APPLICATION DEVELOPMENT

Teacher: Vanesa Maldonado Guerrero
Curs 2025/26

Data query (SELECT)

To query database tables, we use the **SELECT** statement . With it, we can query one or more tables. It is undoubtedly the most versatile command in the SQL language .

There are many clauses associated with the **SELECT** statement (**GROUP BY, ORDER, HAVING, UNION**). It is also one of the statements in which database engines most frequently incorporate clauses beyond the standard one, which is the one we will examine here.

Let's start by looking at simple queries, based on a single table. We'll see how to retrieve rows and columns from a table in the order we need.

The result of a **SELECT** query returns a logical table . That is, the results are a data relation, consisting of rows/records with a series of fields/columns, just like any table in the database. However, this table is stored in memory only while it is being used, and then discarded. Each time the query is executed, the result is recalculated.

The basic syntax of a **SELECT** query is as follows (optional values are enclosed in square brackets):

```
1  SELECT [ ALL / DISTINC ] [ * ] / [ListaColumnas_Expresiones] AS [Expresion]
2  FROM Nombre_Tabla_Vista
3  WHERE Condiciones
4  ORDER BY ListaColumnas [ ASC / DESC ]
```

Data query (SELECT)

SELECT

It allows you to select which columns to display and in what order. It's simply the instruction the database interprets as a request for information.

ALL / DISTINCT

ALL is the default value, specifying that the result set can include duplicate rows. It is generally never used.

DISTINCT specifies that the result set can only include unique rows. That is, if a query finds identical records appearing more than once, these are removed. Very useful in many situations.

Field names

You must specify a list of field names from the table that you are interested in and therefore want to return. There will usually be more than one, in which case you separate each name from the others with commas.

You can precede the column names with the table name using the Table.Column format . In addition to column names, you can also include constants, arithmetic expressions, and functions in this list to create dynamically calculated fields.

If we want it to return all the fields in the table, we use the wildcard **“*” (asterisk)**.

Data query (SELECT)

The names provided must exactly match the field names in the table, but if we want them to have different names in our logical results table, we can use:

AS

It allows you to rename columns if used in the SELECT clause, or to rename tables if used in the FROM clause. It's optional. With this, you can create various aliases for columns and tables. We'll see an example shortly.

FROM

This clause allows you to specify the tables or views from which we will obtain the information. For now, we will look at examples for obtaining information from a single table.

As mentioned above, tables can also be renamed using the “AS” instruction.

WHERE

Specifies the filter condition for the returned rows. This is used when you don't want all rows from a table to be returned, but only those that meet certain conditions. This clause is commonly used in most queries.

Conditions

These are logical expressions to be checked against the filter condition. After evaluation, they return TRUE or FALSE for each row, depending on whether the condition is met.

Data query (SELECT)

ORDER BY

Defines the order of the rows in the result set. Specify the field or fields (separated by commas) by which you want to sort the results.

ASC / DESC

ASC is the default value; it specifies that the column indicated in the ORDER BY clause will be sorted in ascending order, that is, from smallest to largest. If DESC is specified instead, it will be sorted in descending order (from largest to smallest).

For example, to sort the results in ascending order by city, and those from the same city in descending order by name, we would use this sorting clause:

```
1  … ORDER BY Ciudad, Nombre DESC …
```

Since we haven't set ASC or DESC to the City column, the default value (which is ASC) will be used.

NOTE: Although at first, if you're not yet used to it, it might seem like it's sorting by both columns in descending order. If that's what you want, you should write … ORDER BY City DESC, Name DESC …

Aggregation functions

Aggregate functions in SQL allow us to perform operations on a set of results, but return a single aggregated value for all of them. In other words, they allow us to obtain averages, maximums, etc., from a set of values.

The basic aggregation functions supported by all data managers are as follows:

- **COUNT**: returns the total number of rows selected by the query.
- **MIN**: returns the minimum value of the field we specify.
- **MAX**: returns the maximum value of the field we specify.
- **SUM**: sums the values in the specified field. Can only be used with numeric columns.
- **AVG**: Returns the average value of the specified field. Can only be used with numeric columns.

All these functions apply to a single column, which we will specify in parentheses, except for the **COUNT** function, which can be applied to a column or indicated with an asterisk (*). The difference between using a column name and an asterisk (*) is that in the first case, null values for that column are not counted, while in the second, they are.

```
1  SELECT COUNT(*) AS TotalFilas, COUNT(ShipRegion) AS FilasNoNulas,
2  MIN(ShippedDate) AS FechaMin, MAX(ShippedDate) AS FechaMax,
3  SUM(Freight) AS PesoTotal, AVG(Freight) PesoPromedio
4  FROM Orders
```

Aggregation functions: example

id_employee	name	department	salary	age
1	Ana	Sales	2500	28
2	Luis	Sales	2700	35
3	Marta	Accounting	3200	41
4	Pedro	Sales	NULL	30
5	Carla	Accounting	3500	38

Counts the number of rows or values:

-- Count all employees (includes rows with NULL salaries)

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

-- Count only employees with a salary (excludes NULL values)

```
SELECT COUNT(salary) AS employees_with_salary
FROM employees;
```

Returns the smallest value in a column:

-- Lowest salary

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

-- Youngest employee

```
SELECT MIN(age) AS youngest_age
FROM employees;
```

Aggregation functions: example

id_employee	name	department	salary	age
1	Ana	Sales	2500	28
2	Luis	Sales	2700	35
3	Marta	Accounting	3200	41
4	Pedro	Sales	NULL	30
5	Carla	Accounting	3500	38

Returns the largest value in a column:

```
-- Highest salary
    SELECT MAX(salary) AS maximum_salary
    FROM employees;

    -- Oldest employee
    SELECT MAX(age) AS oldest_age
    FROM employees;
```

Adds up all numeric values (ignores NULLs):

```
-- Total of all salaries
    SELECT SUM(salary) AS total_salaries
    FROM employees;
```

Returns the average (mean) value of a numeric column:

```
-- Average salary
SELECT AVG(salary) AS average_salary
FROM employees;
```

What are SQL Operators?

SQL operators are one or more symbols or characters that perform a specific action or operation. These are the most important functions of the operators:

- data comparison,
- data filtering and classification;
- arithmetic calculations,
- logical operations,
- pattern comparison,
- dataset analysis,
- checking for NULL values.

What types of SQL Operators exist?

SQL operators can be divided into different types depending on the action and data query to be performed. The following types exist:

- **Comparison operators** perform comparisons between selected values. They allow you to define specific criteria for filtering, sorting, or grouping data.
- **Logical operators** evaluate the truth values of conditions by forming Boolean expressions that result in TRUE TRUE, OR, FALSE OR UNKNOWN. These include SQL AND, SQL OR , and SQL NOT .
- **Arithmetic operators** are used to perform mathematical calculations with numerical data and values. Among other things, it is possible to add, subtract, or divide values in specific columns. Other operations include multiplication, calculating percentages, and finding roots.
- **String operators**, as their name suggests, perform operations on text strings. For example, they allow you to search for exact terms or patterns and substrings in specific columns and perform comparisons. These include SQL LIKE and SQL wildcards such as the percent sign and underscore, which can be used as substitutes in pattern searches.
- **Set operators** perform operations on selected datasets and return results based on defined conditions. They allow you to manipulate datasets, create intersections and differences, or generate unions. Examples include SQL UNION and UNION EXCEPT.

Comparison operators

SQL Operator	Function
`=`	Checks if two values are equal, returns or `TRUE` `FALSE` filters, updates, and creates values that meet the comparison.
`<>` either `!=`	Checks if two values are different, returns either true `TRUE` or false `FALSE`, and filters or compares values in columns or tables.
`<`	Check less than conditions between values
`>`	Check for greater than conditions between values
`<=` either `>=`	Check values for less than or equal to or greater than or equal to
`!<` either `!>`	Check values for not less than or not greater than (not an ISO standard)

Comparison operators: exemples

```sql
-- Employees who work in the Sales department
SELECT name, department
FROM employees
WHERE department = 'Sales';
```

```sql
-- Employees younger than 35 years old
SELECT name, age
FROM employees
WHERE age < 35;
```

```sql
-- Employees NOT in the Sales department
SELECT name, department
FROM employees
WHERE department <> 'Sales';
```

```sql
-- Employees older than 35
SELECT name, age
FROM employees
WHERE age > 35;
```

```sql
-- Employees 35 years old or younger
SELECT name, age
FROM employees
WHERE age <= 35;
```

```sql
-- Employees with salary greater than or equal to 3000
SELECT name, salary
FROM employees
WHERE salary >= 3000;
```

```sql
-- Equivalent to: salary >= 3000
SELECT name, salary
FROM employees
WHERE salary !< 3000;
```

```sql
-- Equivalent to: salary <= 3000
SELECT name, salary
FROM employees
WHERE salary !> 3000;
```

Logical operators

SQL Operator	Function
`AND`	Both values must meet the AND condition.
`OR`	At least one value must satisfy the OR condition
`NOT`	Checks data sets that do not meet a condition or for which a negated condition applies
`ALL`	Check if all selected values meet a given condition
`BETWEEN`	Check if the values are within a specific range
`EXISTS`	Check if there are values in another data set
`IN`	Check if the values are in a list of items or values
`LIKE`	Check if the values match a specific pattern or an exact string of characters
`SOME`	Check if at least one value in a list or column meets a given condition

Logical operators: exemples

```sql
-- Employees from Sales with salary greater than 2500
SELECT name, department, salary
FROM employees
WHERE department = 'Sales' AND salary > 2500;
```

```sql
-- Employees from Sales OR employees older than 40
SELECT name, department, age
FROM employees
WHERE department = 'Sales' OR age > 40;
```

```sql
-- Employees NOT in Sales
SELECT name, department
FROM employees
WHERE NOT department = 'Sales';
```

```sql
-- Employees aged between 30 and 40
SELECT name, age
FROM employees
WHERE age BETWEEN 30 AND 40;
```

```sql
-- Show departments that have at least one employee
SELECT DISTINCT department
FROM employees e1
WHERE EXISTS (
    SELECT 1
    FROM employees e2
    WHERE e1.department = e2.department
);
```

Logical operators: exemples

```sql
-- Employees whose salary is higher than ALL salaries in Sales
SELECT name, salary
FROM employees
WHERE salary > ALL (
    SELECT salary FROM employees WHERE department = 'Sales'
);
```

```sql
-- Employees in Sales or Accounting
SELECT name, department
FROM employees
WHERE department IN ('Sales', 'Accounting');
```

```sql
-- Employees whose names start with 'A'
SELECT name
FROM employees
WHERE name LIKE 'A%';
```

```sql
-- Employees with salary greater than SOME (at least one) salary in Sales
SELECT name, salary
FROM employees
WHERE salary > SOME (
    SELECT salary FROM employees WHERE department = 'Sales'
);
```

Arithmetic operators

SQL Operator	Function
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus, obtains the remainder of the division

Arithmetic operators: exemple

```sql
SELECT
    price,
    quantity,
    price * quantity AS total_cost,
    price % 5 AS remainder_example
FROM products;
```

```sql
SELECT
    order_id,
    amount_paid,
    total_amount,
    total_amount - amount_paid AS amount_due
FROM orders;
```

```sql
SELECT
    product_name,
    price,
    quantity,
    price * quantity AS total_value
FROM products;
```

```sql
SELECT
    department_id,
    SUM(salary) AS total_salary,
    COUNT(*) AS num_employees,
    SUM(salary) / COUNT(*) AS average_salary
FROM employees
GROUP BY department_id;
```

```sql
SELECT
    employee_id,
    employee_id % 2 AS is_even
FROM employees;
```

Chain operators

SQL Operator	Function
`LIKE`	Check if the datasets match a user-defined search pattern
`NOT LIKE`	Excludes datasets that do not match a defined search pattern
`%`	It serves as a wildcard (SQL wildcard) for one or more unknown characters and can enclose substrings when the first and last characters are unknown.
`[…]`	Defines a set of characters to be searched for within square brackets, such as a letter between A and F like [af]
`_`	Replaces a single character in a search pattern

Chain operators: example

```sql
SELECT
    first_name,
    last_name
FROM employees
WHERE first_name LIKE 'A%';
```

```sql
SELECT
    product_name
FROM products
WHERE product_name NOT LIKE '%Pro%';
```

```sql
SELECT
    city
FROM customers
WHERE city LIKE '%land%';
```

```sql
SELECT
    customer_name
FROM customers
WHERE customer_name LIKE '[A-F]%';
```

```sql
SELECT
    product_code
FROM products
WHERE product_code LIKE 'AB_1';
```

Other operators

SQL Operator	Function
`NVL`, `ISNULL`, `IFNULL`	Check if there are NULL values in the datasets to replace them with defined values.
`DISTINCT`	It is also considered a processing or comparison operator and is used in conjunction with `SELECT` to remove duplicate data sets.

Other operators: example

```
SELECT
    employee_id,
    NVL(commission, 0) AS commission_value
FROM employees;
```

```
SELECT
    customer_name,
    IFNULL(address, 'No Address') AS full_address
FROM customers;
```

```
SELECT
    employee_id,
    ISNULL(phone_number, 'Not Available') AS contact_number
FROM employees;
```

```
SELECT DISTINCT country
FROM customers;
```

```
SELECT DISTINCT department_id, job_id
FROM employees;
```

exemples

Mostrar todos los datos de los Clientes de nuestra empresa.

Mostrar apellido, ciudad y región (LastName, city, region) de los empleados de USA (nótese el uso de AS para darle el nombre en español a los campos devueltos).

Mostrar los clientes que no sabemos a qué región pertenecen (o sea, que no tienen asociada ninguna región).

Mostrar las distintas regiones de las que tenemos algún cliente, accediendo sólo a la tabla de clientes.

Mostrar los clientes que pertenecen a las regiones CA, MT o WA, ordenados por región ascendentemente y por nombre descendentemente.

Mostrar los clientes cuyo nombre empieza por la letra “W”.

Mostrar los empleados cuyo código está entre el 2 y el 9

Mostrar los clientes cuya dirección contenga “ki”

Mostrar las Ventas del producto 65 con cantidades entre 5 y 10, o que no tengan descuento

Solutions

- Mostrar todos los datos de los Clientes de nuestra empresa:

```
SELECT * FROM Customers
```

- Mostrar apellido, ciudad y región (LastName, city, region) de los empleados de USA (nótese el uso de AS para darle el nombre en español a los campos devueltos):

```
SELECT E.LastName AS Apellido, City AS Ciudad, Region
FROM Employees AS E
WHERE Country = 'USA'
```

- Mostrar los clientes que no sabemos a qué región pertenecen (o sea, que no tienen asociada ninguna región) :

```
SELECT * FROM Customers WHERE Region IS NULL
```

- Mostrar las distintas regiones de las que tenemos algún cliente, accediendo sólo a la tabla de clientes:

```
SELECT DISTINCT Region FROM Customers WHERE Region IS NOT NULL
```

Solutions

- Mostrar los clientes que pertenecen a las regiones CA, MT o WA, ordenados por región ascendentemente y por nombre descendentemente.

```
SELECT * FROM Customers WHERE Region IN('CA', 'MT', 'WA')
ORDER BY Region, CompanyName DESC
```

- Mostrar los clientes cuyo nombre empieza por la letra "W":

```
SELECT * FROM Customers WHERE CompanyName LIKE 'W%'
```

- Mostrar los empleados cuyo código está entre el 2 y el 9:

```
SELECT * FROM Employees WHERE EmployeeID BETWEEN 2 AND 9
```

- Mostrar los clientes cuya dirección contenga "ki":

```
SELECT * FROM Customers WHERE Address LIKE '%ki%'
```

- Mostrar las Ventas del producto 65 con cantidades entre 5 y 10, o que no tengan descuento:

```
SELECT * FROM [Order Details] WHERE (ProductID = 65 AND Quantity BETWEEN 5 AND 10) OR Discount = 0
```

Nota: En SQL Server, para utilizar nombres de objetos con caracteres especiales se deben poner entre corchetes. Por ejemplo en la consulta anterior [Order Details] se escribe entre corchetes porque lleva un espacio en blanco en su nombre. En otros SGBDR se utilizan comillas dobles (Oracle, por ejemplo: "Order Details") y en otros se usan comillas simples (por ejemplo en MySQL).