DATABASES

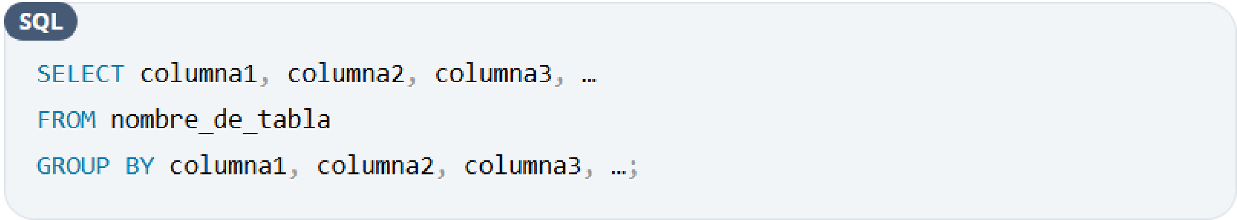
GS WEB APPLICATION DEVELOPMENT

Teacher: Vanesa Maldonado Guerrero
Curs 2025/26

SELECT (GROUP BY)

SQL GROUP BY is a statement used to group rows with identical values. It is used with the SQL command SELECT and follows the WHERE clause. SQL GROUP BY is often used in combination with functions such as SQL AVG() , SQL COUNT()MAX() , or MIN()COUNT () SUM().

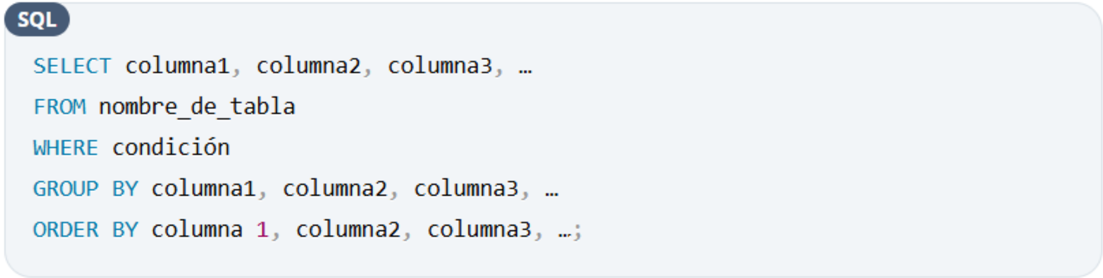
The basic syntax of SQL GROUP BY is as follows:



SQL

```sql
SELECT columna1, columna2, columna3, …
FROM nombre_de_tabla
GROUP BY columna1, columna2, columna3, …;
```

However, the version with a clause **WHERE**, which allows the integration of certain conditions, is much more common:



SQL

```sql
SELECT columna1, columna2, columna3, …
FROM nombre_de_tabla
WHERE condición
GROUP BY columna1, columna2, columna3, …
ORDER BY columna 1, columna2, columna3, …;
```

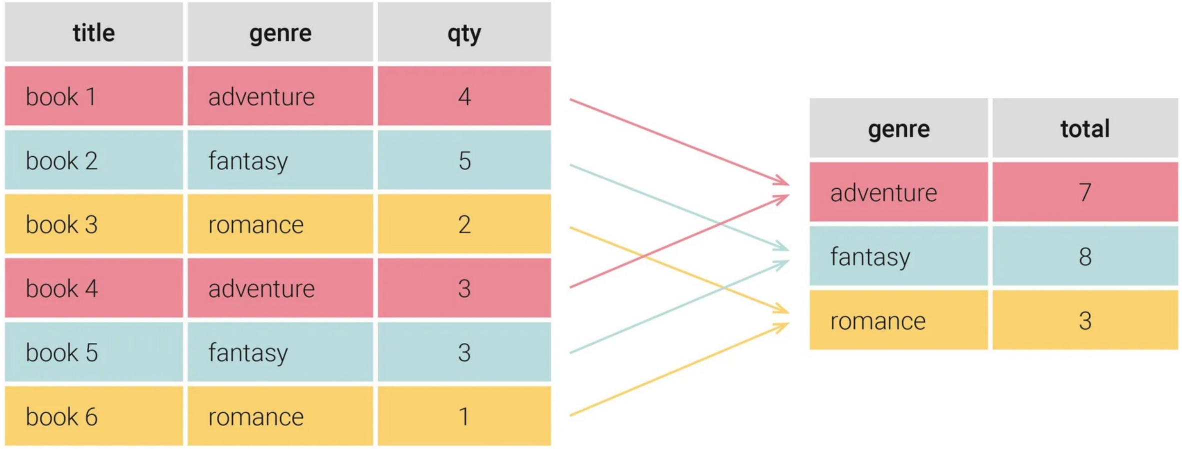
How does SQL GROUP BY work?

Basically, it groups rows with the same value into a group or cube.

Suppose we have a bookstore and we want to know how many books of different genres we have in stock.

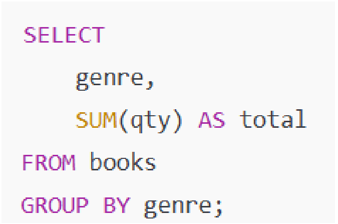
The following visualization shows how the clause **GROUP BY** creates groups from the table data. We want to know the total number of books in each genre; therefore, **GROUP BY** it groups books of the same genre and sums the corresponding quantities.

This creates a results table that lists the genres and their total number of books in our inventory.



title	genre	qty
book 1	adventure	4
book 2	fantasy	5
book 3	romance	2
book 4	adventure	3
book 5	fantasy	3
book 6	romance	1

genre	total
adventure	7
fantasy	8
romance	3



```sql
SELECT
    genre,
    SUM(qty) AS total
FROM books
GROUP BY genre;
```

Using aggregate functions with GROUP BY

GROUP BY It groups rows with the same value into a single cube. We typically want to calculate some statistics for this group of rows, such as the average or total. To do this, SQL provides aggregation functions that combine the values in a given column into a single value for the group.

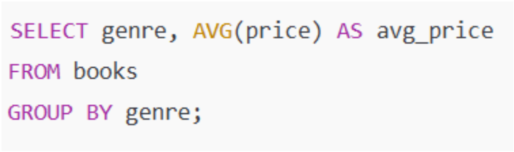
So far, we've only used `grow` **SUM()** as our aggregate function to group book titles in stock. However, this isn't the only aggregate function you can use with `grow` GROUP BY. SQL also offers `grow`.

- **COUNT()** calculate the number of rows in each group.
- **AVG()** find the average value of each group.
- **MIN()** return the minimum value of each group.
- **MAX()** return the maximum value of each group.

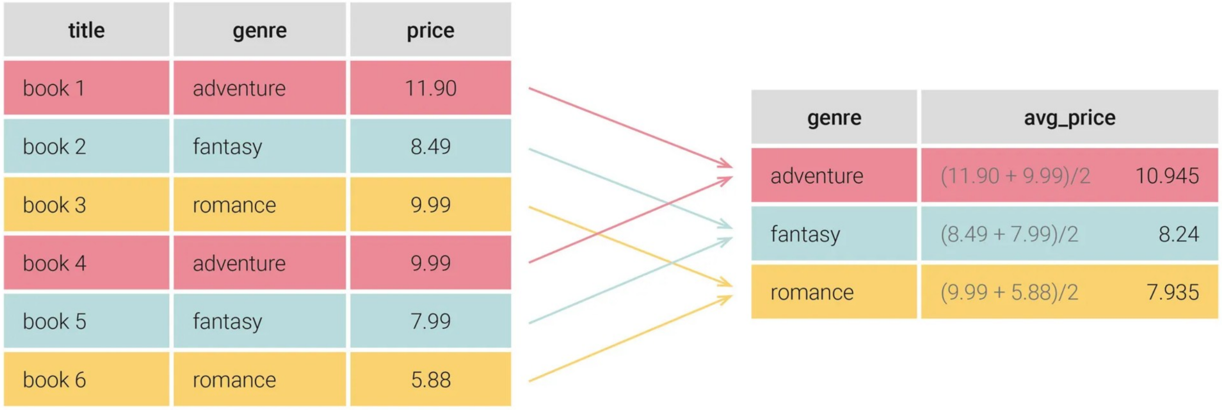
Let's see how the function works **AVG() GROUP BY.**

This time, we want to calculate the average price of books in each genre.

We'll start by visualizing the result we want to obtain:



```
SELECT genre, AVG(price) AS avg_price
FROM books
GROUP BY genre;
```

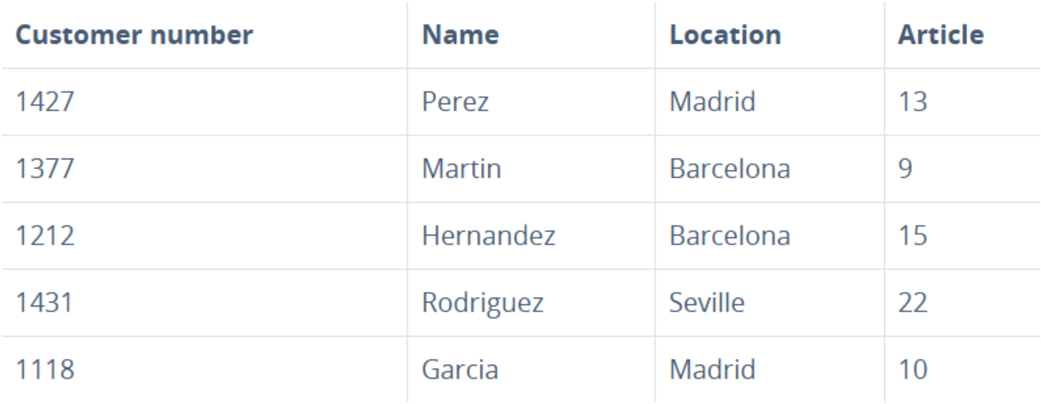


title	genre	price
book 1	adventure	11.90
book 2	fantasy	8.49
book 3	romance	9.99
book 4	adventure	9.99
book 5	fantasy	7.99
book 6	romance	5.88

genre	avg_price	
adventure	(11.90 + 9.99)/2	10.945
fantasy	(8.49 + 7.99)/2	8.24
romance	(9.99 + 5.88)/2	7.935

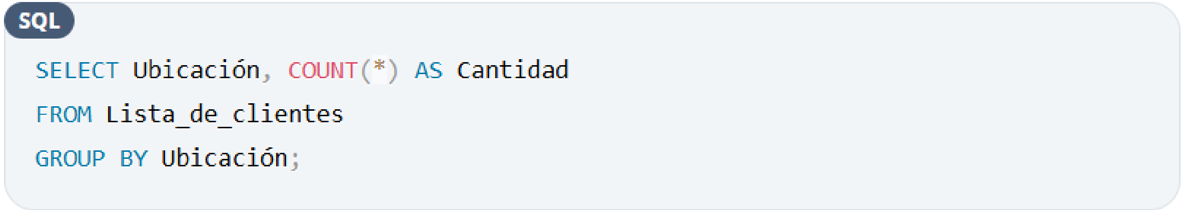
Example of use with COUNT()

This table contains columns for customer number, name, location, and items purchased:



Customer number	Name	Location	Article
1427	Perez	Madrid	13
1377	Martin	Barcelona	9
1212	Hernandez	Barcelona	15
1431	Rodriguez	Seville	22
1118	Garcia	Madrid	10

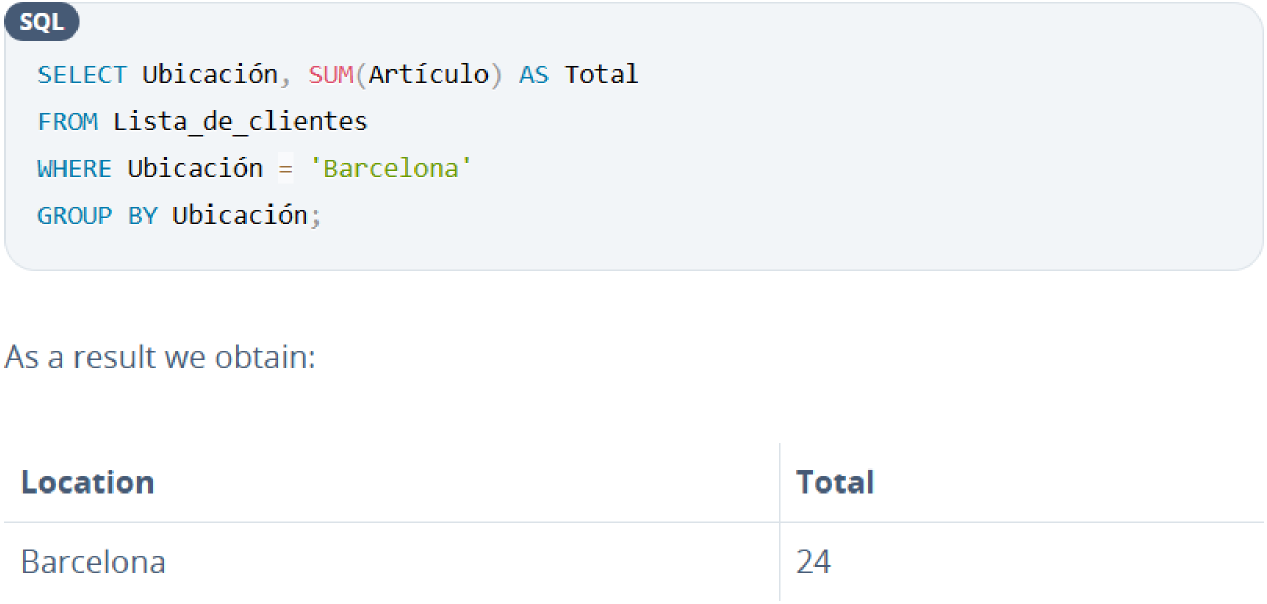
We can now use SQL **GROUP BY** in combination with the function **COUNT()**, for example, to list how many customers come from which cities. Here is the corresponding code:



```sql
SELECT Ubicación, COUNT(*) AS Cantidad
FROM Lista_de_clientes
GROUP BY Ubicación;
```

The sentence along with SUM()

In the following example, we use SQL GROUP BY in combination with SUM()to determine and display how many items were ordered from Barcelona:



SQL

```
SELECT Ubicación, SUM(Artículo) AS Total
FROM Lista_de_clientes
WHERE Ubicación = 'Barcelona'
GROUP BY Ubicación;
```

As a result we obtain:

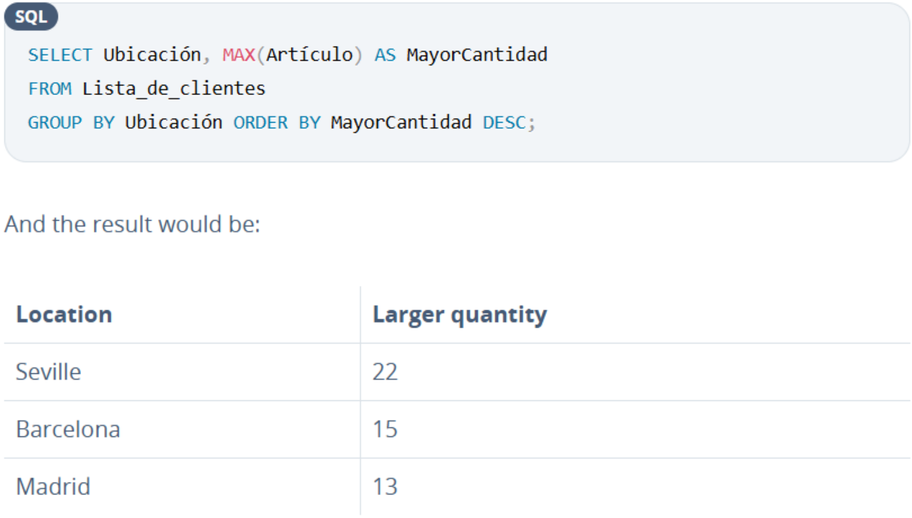
Location	Total
Barcelona	24

Use with ORDER BY

A combination with ` **ORDER BY GROUP BY**` is also possible.

For our table, we order by the highest number of items ordered per customer and by city. We start with the city where a customer has purchased the most items.

The corresponding code, for which we combine SQL GROUP BY with the `GROUP BY` MAX() and `GROUP BY` functions ORDER BY, is as follows:



SQL

```sql
SELECT Ubicación, MAX(Artículo) AS MayorCantidad
FROM Lista_de_clientes
GROUP BY Ubicación ORDER BY MayorCantidad DESC;
```

And the result would be:

Location	Larger quantity
Seville	22
Barcelona	15
Madrid	13

The SQL HAVING Clause

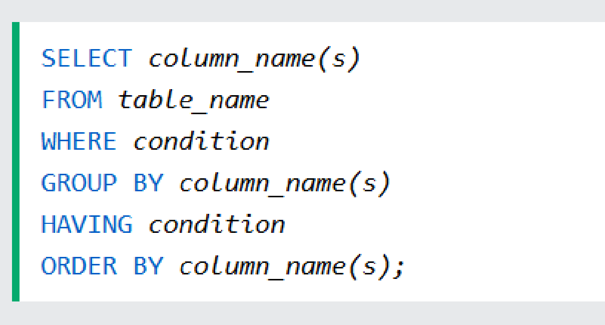
The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions.

The clause **SQL HAVING** is typically used with the clause **GROUP BY** to filter groups of returned rows. Rows are only included in the group when the specified condition is TRUE.

The GROUP BY clause SQL GROUP BY organizes data into groups based on common values; it is most often used to obtain summary information and calculate aggregate statistics.

There can be some confusion between **WHERE** and **HAVING**, but the difference is that clause **WHERE** applies a condition to the entire column, filtering individual rows. It doesn't work with aggregate functions like **SUM() WHERE** and **HAVING AVG().**

On the other hand, the **WHERE** clause **HAVING** places filtering conditions on the groups created by the **WHERE** clause **GROUP BY.** It can be used with aggregate functions.

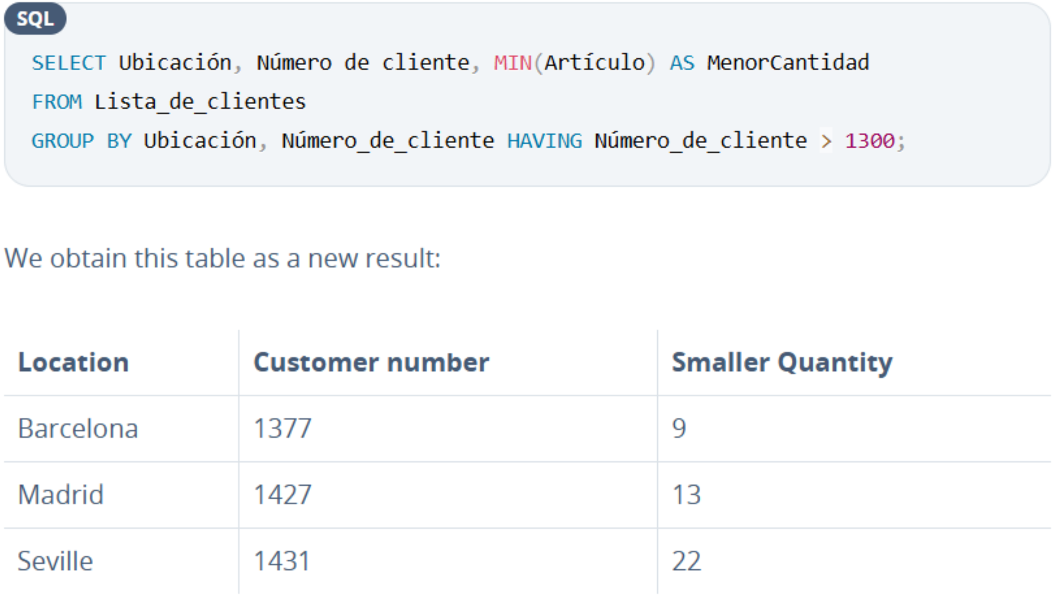


```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Group By in combination with HAVING

You can also combine **SQL GROUP BY** with **SQL HAVING**.

In the following example, we remove from the list those customers whose customer number is less than 1300. We then sort the remaining customers by the number of items they have ordered in ascending order:



SQL

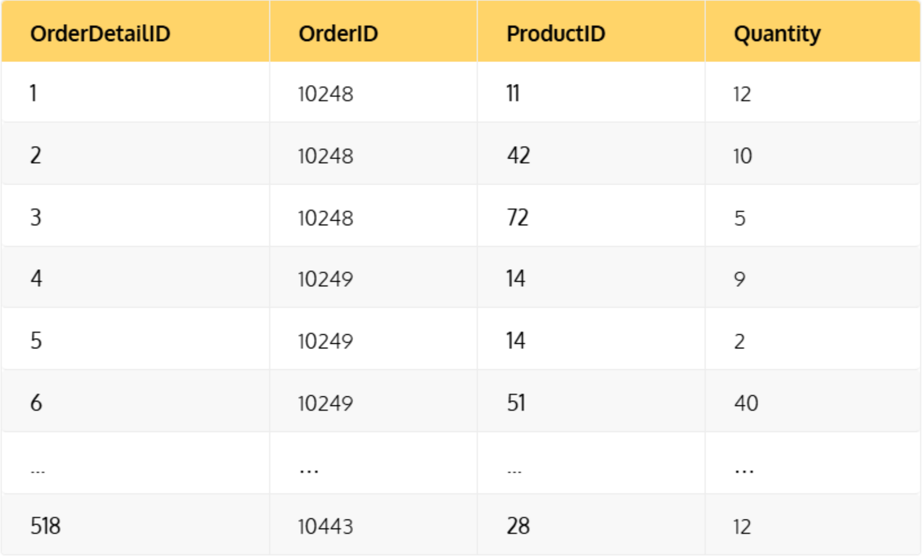
```sql
SELECT Ubicación, Número de cliente, MIN(Artículo) AS MenorCantidad
FROM Lista_de_clientes
GROUP BY Ubicación, Número_de_cliente HAVING Número_de_cliente > 1300;
```

We obtain this table as a new result:

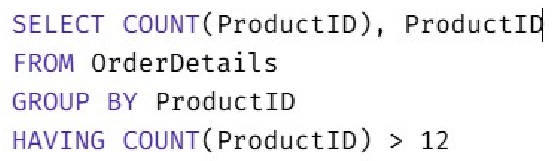
Location	Customer number	Smaller Quantity
Barcelona	1377	9
Madrid	1427	13
Seville	1431	22

Example 1: HAVING with COUNT()

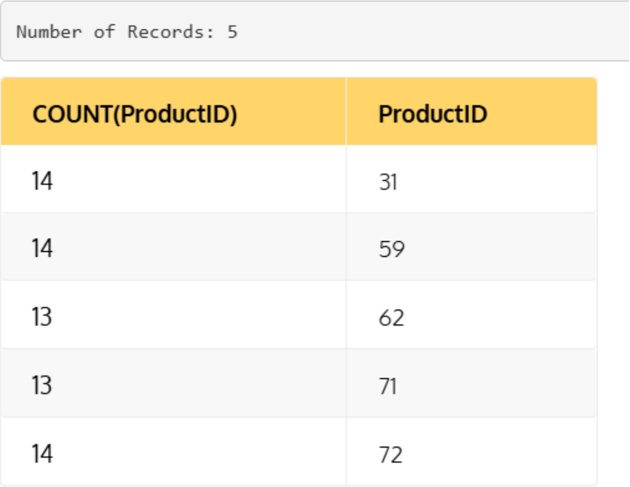
PROBLEM: List all products with more than 12 individual orders placed.



OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	14	2
6	10249	51	40
...	...	...	...
518	10443	28	12



```
SELECT COUNT(ProductID), ProductID
FROM OrderDetails
GROUP BY ProductID
HAVING COUNT(ProductID) > 12
```

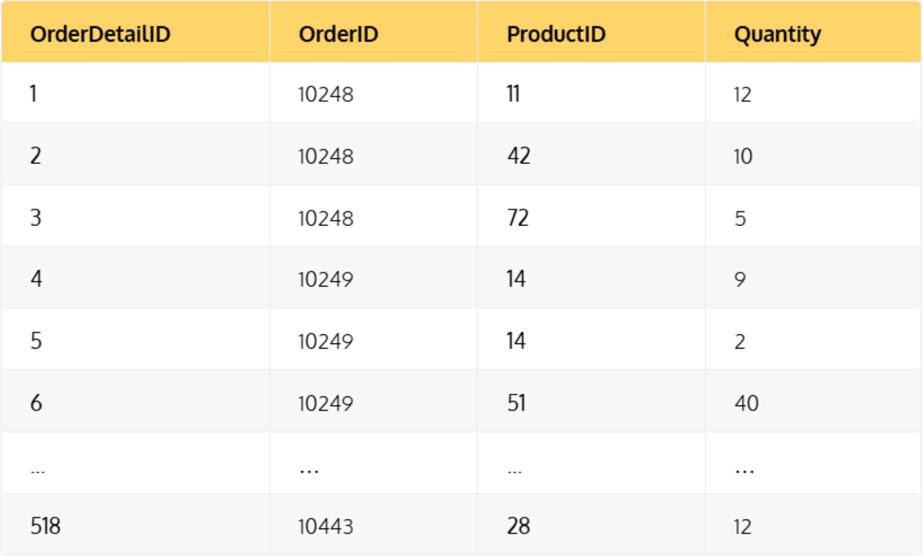


```
Number of Records: 5
```

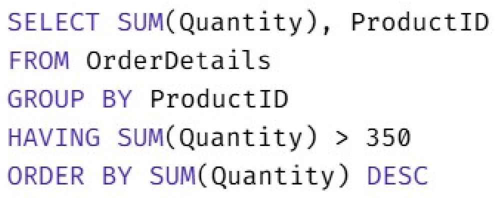
COUNT(ProductID)	ProductID
14	31
14	59
13	62
13	71
14	72

Example 2: HAVING with SUM()

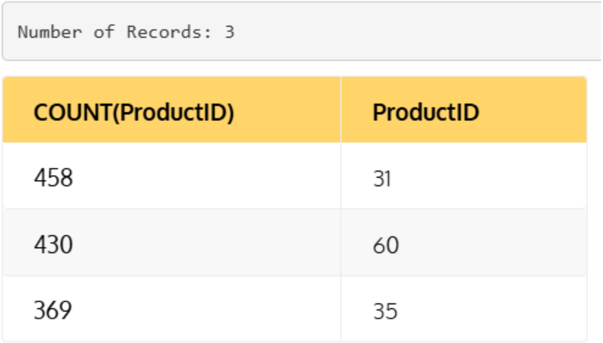
PROBLEM: List all products that have had more than 350 total quantities ordered..



OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	14	2
6	10249	51	40
...	...	...	...
518	10443	28	12



```
SELECT SUM(Quantity), ProductID
FROM OrderDetails
GROUP BY ProductID
HAVING SUM(Quantity) > 350
ORDER BY SUM(Quantity) DESC
```

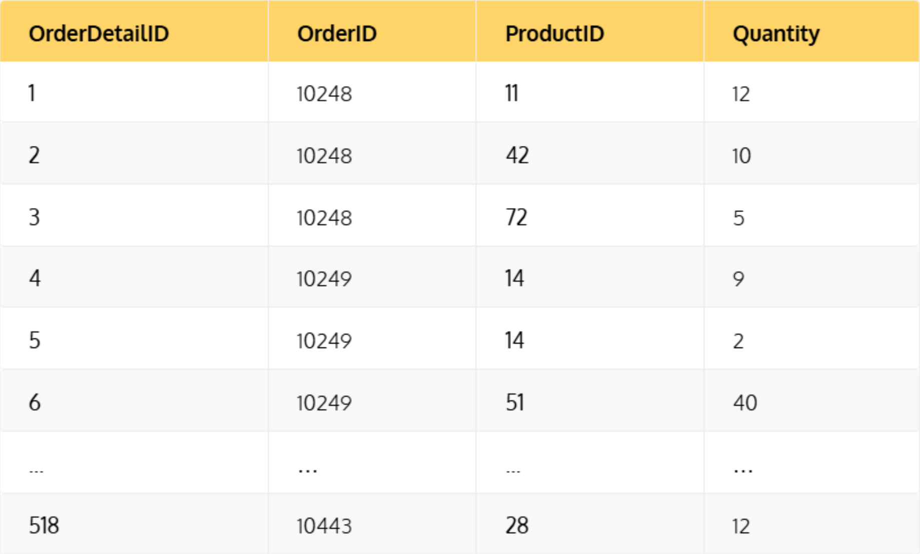


```
Number of Records: 3
```

COUNT(ProductID)	ProductID
458	31
430	60
369	35

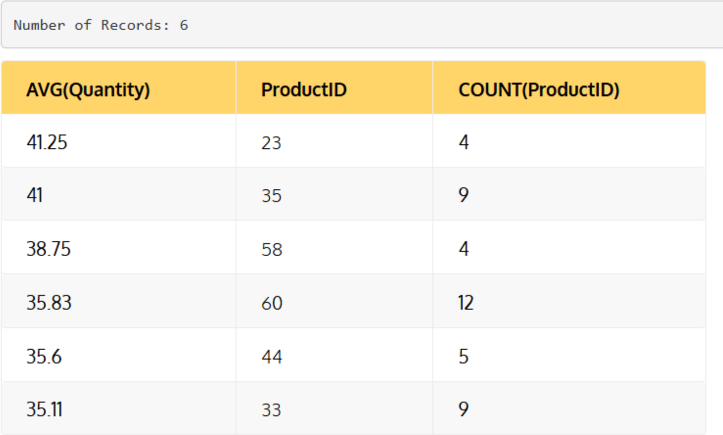
Example 3: HAVING with AVG()

PROBLEM: List all products that have had more than 35 average order quantities and at least 4 orders placed.



OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	14	2
6	10249	51	40
...	...	...	...
518	10443	28	12

```sql
SELECT AVG(Quantity), COUNT(ProductID), ProductID
FROM OrderDetails
GROUP BY ProductID
HAVING AVG(Quantity) > 35 AND COUNT(ProductID) > 3
ORDER BY AVG(Quantity) DESC
```



```
Number of Records: 6
```

AVG(Quantity)	ProductID	COUNT(ProductID)
41.25	23	4
41	35	9
38.75	58	4
35.83	60	12
35.6	44	5
35.11	33	9

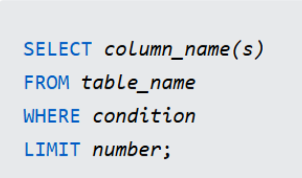
SQL SELECT TOP Clause

The **SELECT TOP** clause is used to specify the number of records to return.

The **SELECT TOP** clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

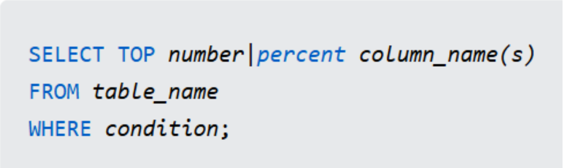
Note: Not all database systems support the SELECT TOP clause.
MySQL supports the **LIMIT** clause to select a limited number of records, while **Oracle** uses **FETCH FIRST *n* ROWS ONLY** and **ROWNUM**.

MySQL Syntax:



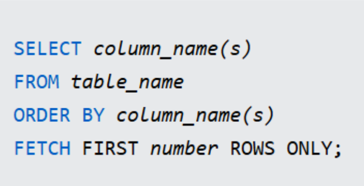
```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

SQL Server / MS Access Syntax:



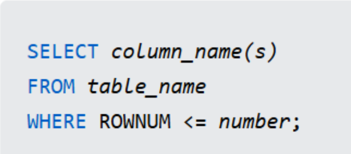
```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

Oracle 12 Syntax:



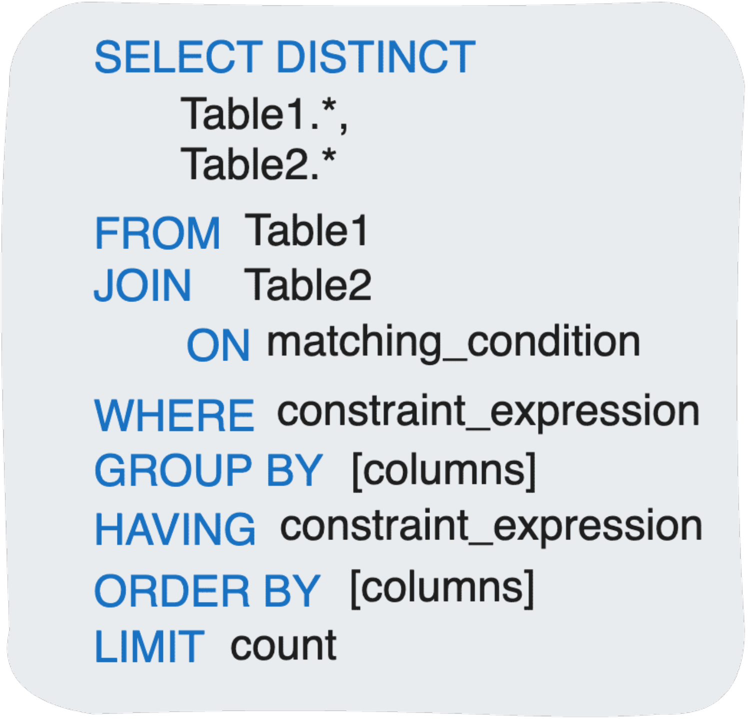
```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s)
FETCH FIRST number ROWS ONLY;
```

Older Oracle Syntax:



```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

SELECT instruction execution order



```
SELECT DISTINCT
        Table1.*,
        Table2.*
FROM  Table1
JOIN    Table2
        ON matching_condition
WHERE  constraint_expression
GROUP BY  [columns]
HAVING  constraint_expression
ORDER BY  [columns]
LIMIT  count
```

Subqueries

Introduction to subqueries:

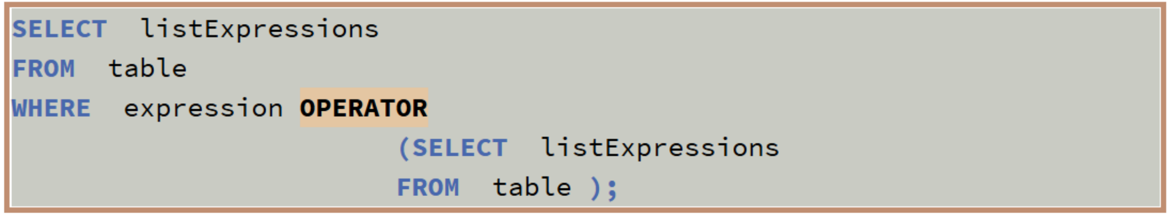
Using **subqueries** is a technique that allows you to use the results of one **SELECT** statement in another **SELECT** statement. It enables you to solve complex queries by using results obtained from previous queries.
A SELECT statement placed inside another SELECT statement is known as a SUBSELECT statement.
This SUBSELECT statement can be placed within WHERE , HAVING , FROM , or JOIN clauses .

Subqueries

Use of simple subqueries

Simple subqueries are those that return a single row. If they also return a single column, they are called scalar subqueries , since they return a single value.

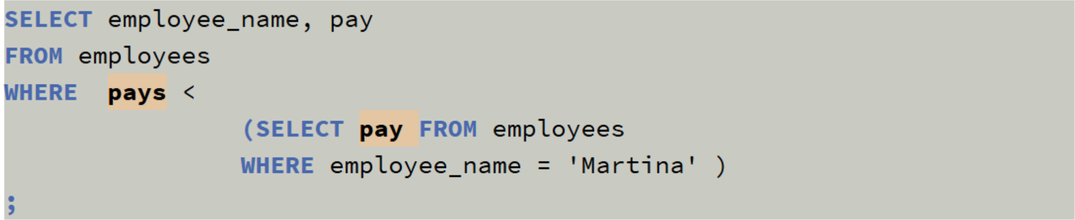
The syntax is:



```
SELECT  listExpressions
FROM  table
WHERE  expression OPERATOR
                        (SELECT  listExpressions
                        FROM  table );
```

The operator can be **>,<,>=,<=,!=, =** or **IN**.

Example:

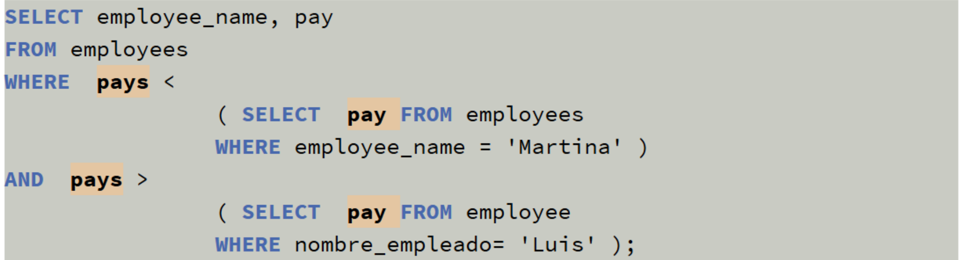


```
SELECT employee_name, pay
FROM employees
WHERE  pays <
               (SELECT pay FROM employees
               WHERE employee_name = 'Martina' )
;
```

This query displays the name and salary of employees whose salary is less than that of employee Martina. For this query to work, the subquery can only return one value (there can only be one employee named Martina).

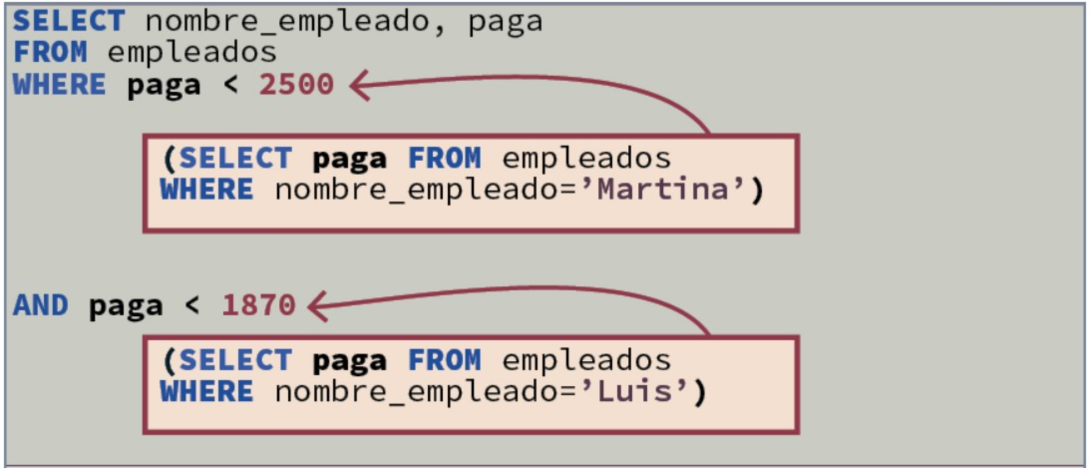
Subqueries

Subqueries can be used as many times as needed:



```
SELECT employee_name, pay
FROM employees
WHERE  pays <
                ( SELECT  pay FROM employees
                WHERE employee_name = 'Martina' )
AND  pays >
                ( SELECT  pay FROM employee
                WHERE nombre_empleado= 'Luis' );
```

Actually, the first thing the database does is calculate the result of the subquery:



```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga < 2500

           (SELECT paga FROM empleados
           WHERE nombre_empleado='Martina')


AND paga < 1870

           (SELECT paga FROM empleados
           WHERE nombre_empleado='Luis')
```

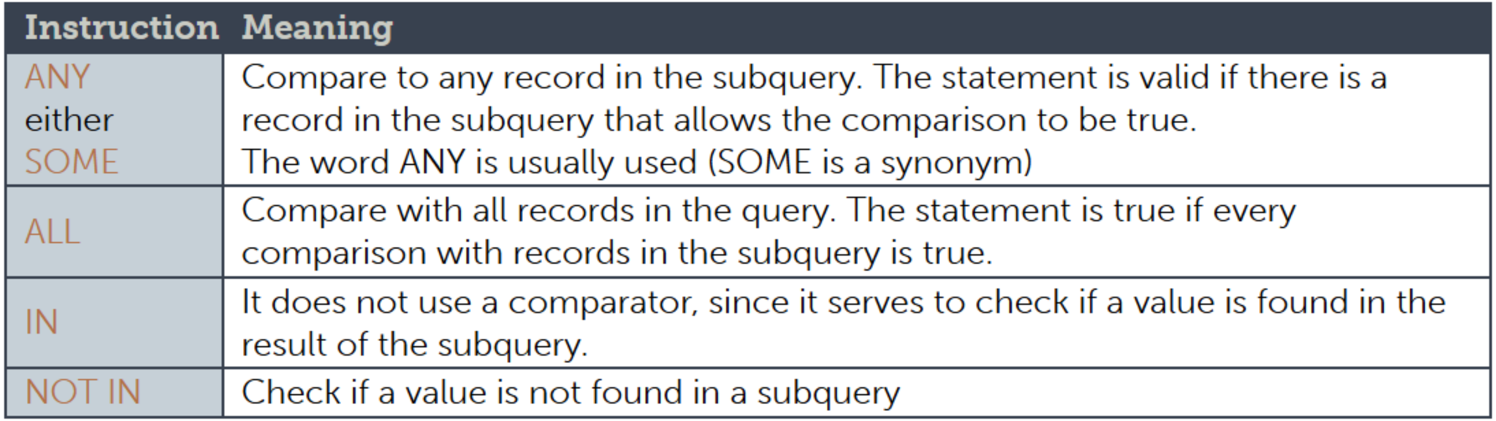
Subqueries

Use of multi-row subqueries

Sometimes you need queries like: *show the salary and name of employees whose salary exceeds that of any employee in the sales department.*

The subquery needed for that result would show all salaries in the sales department. However, we can't use a comparison operator directly because that subquery returns more than one row. The solution is to use special instructions between the operator and the query, which allow the use of multi-row subqueries.

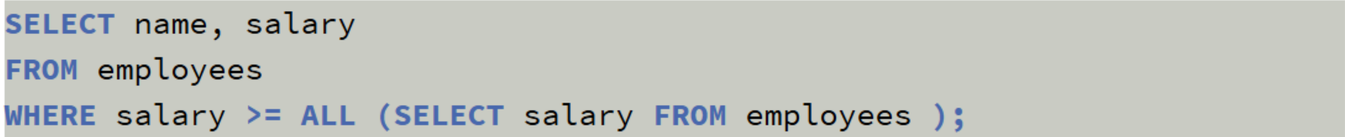
Those instructions are:



Instruction	Meaning
ANY either SOME	Compare to any record in the subquery. The statement is valid if there is a record in the subquery that allows the comparison to be true. The word ANY is usually used (SOME is a synonym)
ALL	Compare with all records in the query. The statement is true if every comparison with records in the subquery is true.
IN	It does not use a comparator, since it serves to check if a value is found in the result of the subquery.
NOT IN	Check if a value is not found in a subquery

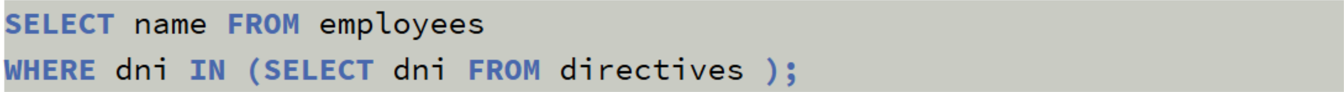
Subqueries

Use of multi-row subqueries: examples



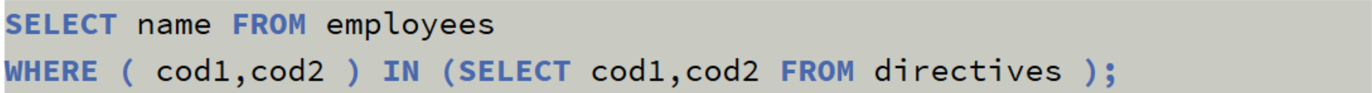
```
SELECT name, salary
FROM employees
WHERE salary >= ALL (SELECT salary FROM employees );
```

The previous query retrieves the highest-paid employee.



```
SELECT name FROM employees
WHERE dni IN (SELECT dni FROM directives );
```

The names of the employees whose ID numbers are in the management table.



```
SELECT name FROM employees
WHERE ( cod1,cod2 ) IN (SELECT cod1,cod2 FROM directives );
```

If you need to compare two columns in an IN query

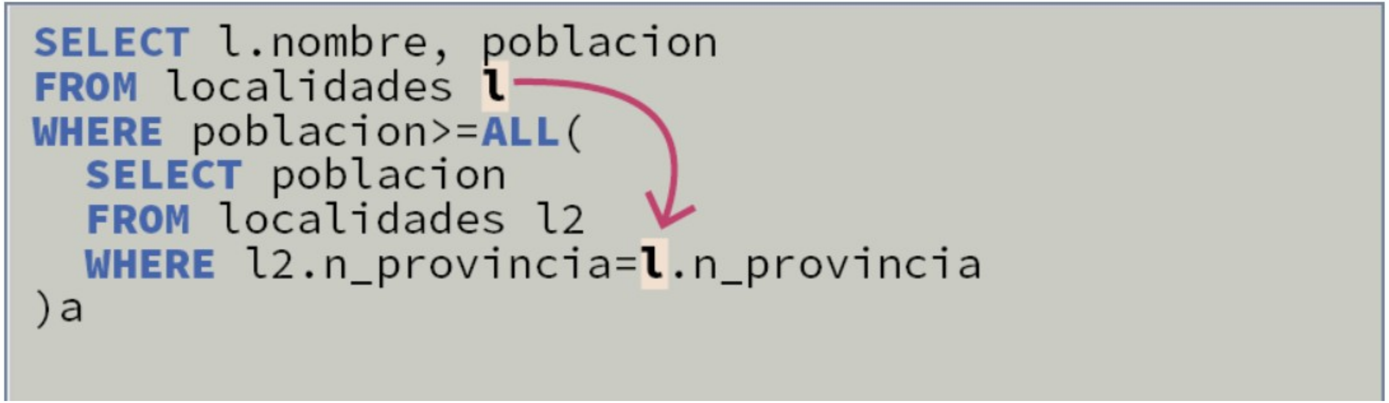
Subqueries

Correlated queries

In subqueries, you may sometimes want to be able to use data from the main query. This is possible by using the alias of the table you want to use from the main query.

For example, suppose we want to retrieve from a geographic database the name and population of the most populated towns in a given province. That is, the towns with the largest population in their province. To do this, we need to compare the population of each town with the total population of all towns in its province. Let's assume the towns table stores the name, population, and province number of each town.

The question would be:

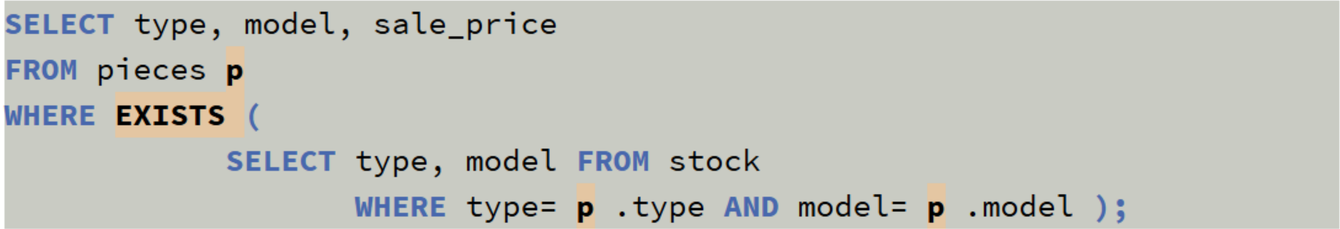


```
SELECT l.nombre, poblacion
FROM localidades l
WHERE poblacion>=ALL(
  SELECT poblacion
  FROM localidades l2
  WHERE l2.n_provincia=l.n_provincia
)a
```

Subqueries

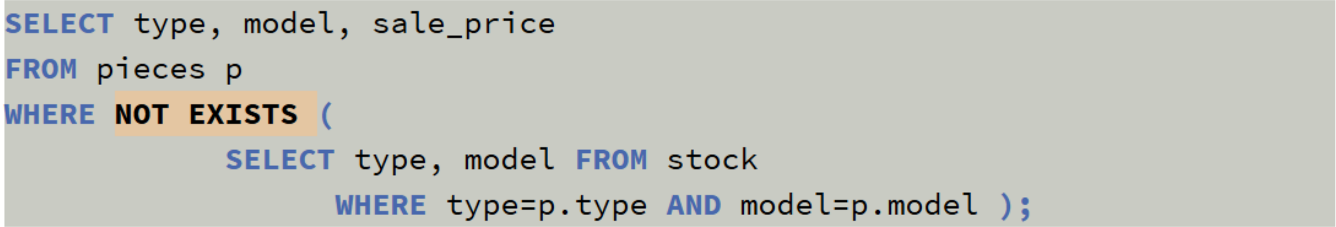
EXISTS queries

This operator returns true if the following query returns a value. Otherwise, it returns false. It is typically used with related queries. Example:



```
SELECT type, model, sale_price
FROM pieces p
WHERE EXISTS (
            SELECT type, model FROM stock
                   WHERE type= p .type AND model= p .model );
```

This query returns the parts that are in the stock table (it is the same as the example discussed in the section on subqueries on multiple values).
The opposite question is:



```
SELECT type, model, sale_price
FROM pieces p
WHERE NOT EXISTS (
            SELECT type, model FROM stock
                  WHERE type=p.type AND model=p.model );
```