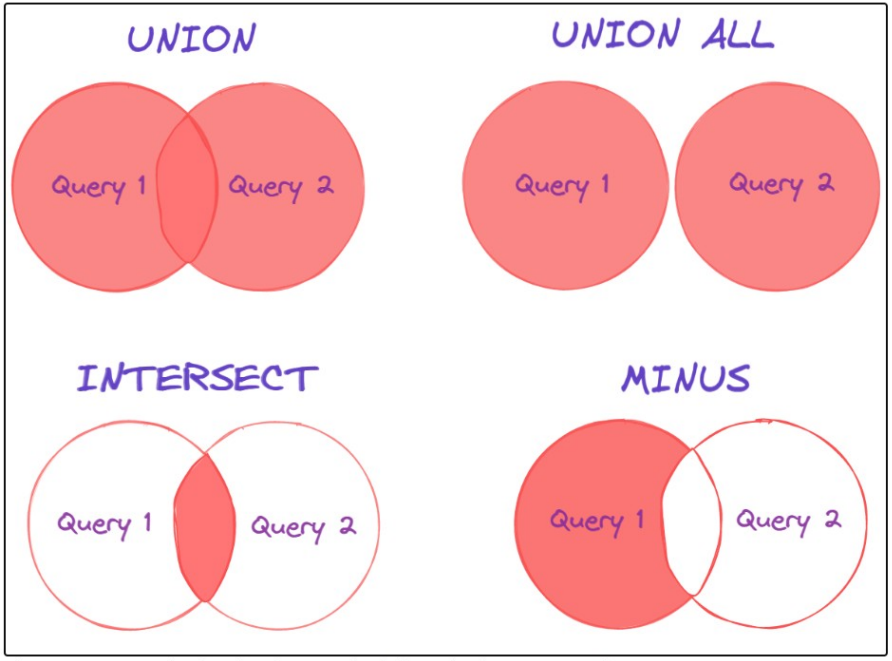
DATABASES

GS WEB APPLICATION DEVELOPMENT

Teacher: Vanesa Maldonado Guerrero
Curs 2025/26

Queries with set operators



Introduction

These queries use at least two SELECT statements whose results can be combined to form a single query.

They are based on mathematical set operators (**union**, **intersection**, and **difference**).

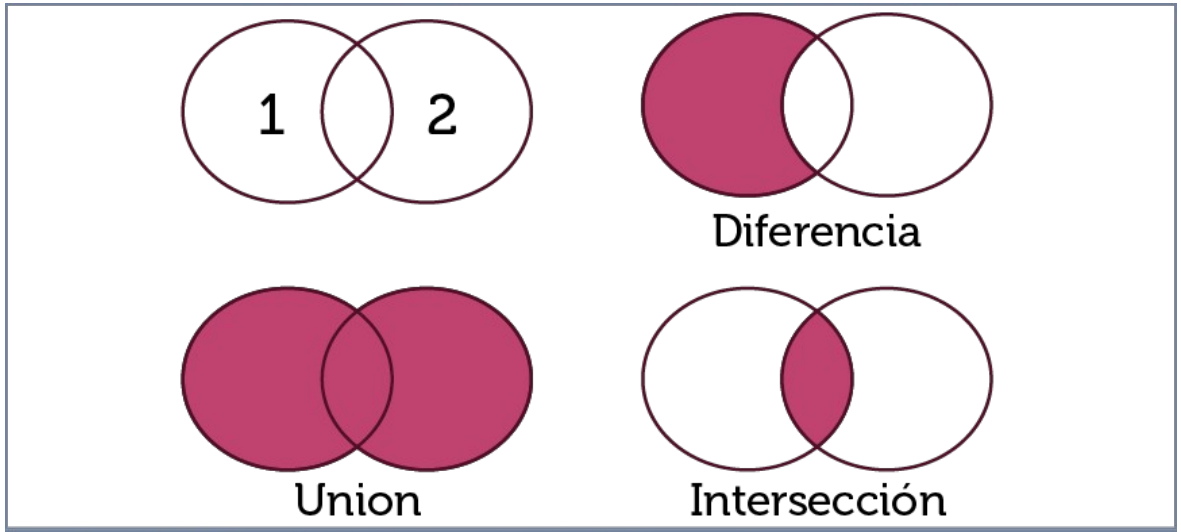


Ilustración 29. Ejemplo de funcionamiento de subconsultas

Introduction

The process is the same as with sets. The most important thing to keep in mind is that in these operations, the number of columns, their type, and their order must be the same in all the queries being combined.

For example:

```
SELECT name FROM people
UNION
SELECT age FROM people;
```

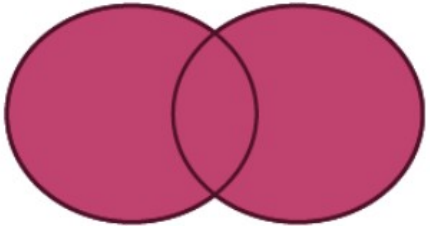
This code would result in an error, since the name columns (usually of type VARCHAR2) and age (usually of type NUMBER) are not of the same type, which would cause an error.

In this other code:

```
SELECT name, age FROM clients
UNION
SELECT age, name FROM partners;
```

Although both queries to be joined have the same columns and the same type, they are not in the same order, so we will have a new error.

Unions



The **UNION** keyword allows you to combine the results of one SELECT statement into another. For this to work, both statements must use the same number and type of columns.

Example:

```
SELECT name FROM provinces
UNION
SELECT name FROM communities
```

The result is a table containing the names of provinces and autonomous communities. In other words, **UNION** returns a query whose result is the rows from both queries.

handling duplicates in joints

The UNION operator combines the results of multiple SELECT statements. However, if there are duplicate entries, it removes them.

Example:

```
SELECT nif, nombre, apellido1, apellido2
FROM clients
UNION
SELECT nif, nombre, apellido1, apellido2
FROM partners;
```

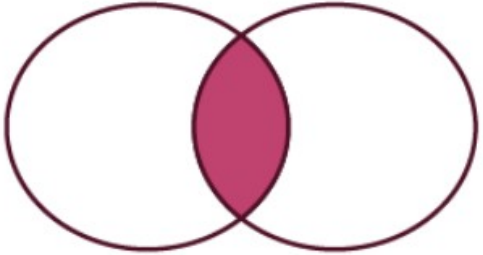
If there is a person who is both a customer and a partner, they would only appear once.

Therefore, SQL provides another operator, **UNION ALL** :

```
SELECT nif, nombre, apellido1, apellido2
FROM clients
UNION ALL
SELECT nif, nombre, apellido1, apellido2
FROM partners;
```

The operation and use are similar, but **UNION ALL** does not remove duplicate data.

Intersections

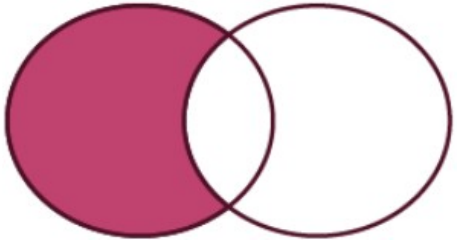


Similarly, the word **INTERSECT** allows you to join two SELECT queries so that the result will be the rows that are present in both queries.

Example: types and models of parts that are only found in warehouses 1 and 2:

```
SELECT type, model FROM stock
WHERE n_almacen=1
INTERSECT
SELECT type, model FROM stock
WHERE n_almacen=2
```

differences



With **MINUS(oracle) o EXCEPT (PostgreSQL, SQL Server y MySQL)**, two SELECT queries are also combined so that the records from the first SELECT that are not present in the second will appear.

Example: types and models of parts that are found in warehouse 1 and not in warehouse 2

```
SELECT type, model FROM stock
WHERE n_almacen=1
MINUS
SELECT type, model FROM stock
WHERE n_almacen=2 ;
```

combination of operations

Multiple nested combinations are possible, allowing for very complex queries. Parentheses are used to clarify which operations to perform first.

Example:

```
(SELECT type, model FROM stocks
WHERE n_almacen=1
INTERSECT
SELECT type, model FROM stock
WHERE n_almacen=2 )
MINUS
SELECT type, model FROM stock
WHERE n_almacen=3;
```

This code returns the types and models of parts that are in warehouses 1 and 2, but not in warehouse number 3.

ordering of combined queries

If we want the results of **UNION** , **INTERSECT** , or **MINUS** operations to be sorted, we must remember that the **ORDER BY** clause must be placed at the end of the statement.

The problem is that the column name can be different in the various SELECT statements used by the query, as in:

```
SELECT lastname1
FROM students
UNION
SELECT lastname2
FROM students;
```

The solution is to use the column names given in the first SELECT statement of the instruction. Example:

```
SELECT lastname1
FROM students
UNION
SELECT lastname2
FROM students
ORDER BY lastname1;
```

This other code wouldn't work:

```
SELECT lastname1
FROM students
UNION
SELECT lastname2
FROM students
ORDER BY lastname2; //error
```

Ordering of combined queries

Furthermore, it's crucial to understand that the ORDER BY clause can only be used at the end of all the combined SELECT statements. The following code is incorrect:

The following code is incorrect:

```
SELECT nif, nombre, apellido1, apellido2
FROM clients
ORDER BY lastname1, lastname2, firstname --Error
UNION ALL
SELECT nif, nombre, apellido1, apellido2
FROM partners;
```

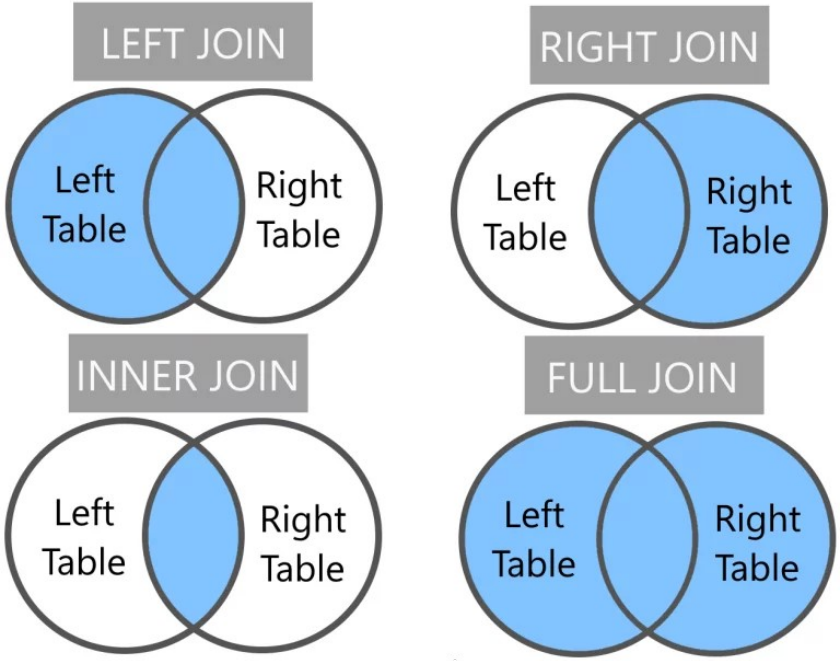
It is also:

```
SELECT nif, nombre, apellido1, apellido2
FROM clients
ORDER BY lastname1, lastname2, firstname --Error
UNION ALL
SELECT nif, nombre, apellido1, apellido2
FROM partners
ORDER BY lastname1, lastname2, firstname;
```

Only one is correct:

```
SELECT nif, nombre, apellido1, apellido2
FROM clients
UNION ALL
SELECT nif, nombre, apellido1, apellido2
FROM partners
ORDER BY lastname1, lastname2, firstname; --Correct
```

Queries on multiple tables. JOIN clause



Retrieve data from multiple tables

Relational databases store their data in multiple tables. Typically, almost any query requires data from several tables simultaneously. This is possible because the data in the tables is linked by columns containing secondary or foreign keys, which allow data from one table to be related to data from another.

As an example, let's look at this table of **departments**:

cod_dep	Department
1	Sales
2	Production
3	Quality
4	Address

On the other hand, we have a table of **employees**:

cod_emp	Name	Last name	cod_dep	age
1	Marisa	Lion	1	54
2	Arthur	Curly	1	58
3	Ann	Ten	2	43
4	Pau	Cabanillas	2	29
5	Luisa	Rodriguez	3	34
6	Anxo	Olivenza	4	21
7	Peter	Anderez		40

The **cod_dep** column in the **employee table** is a secondary key. Through it, we know that Marisa León , for example, is from the Sales department.

Cross product or Cartesian product of tables

Using the tables from the previous slide, if you want to obtain a list of the data for the departments and employees, you could do it this way:

```
SELECT name, surname, department
FROM  departments, employees
ORDER BY name, surname, department;
```

The syntax is correct since, indeed, in the FROM section you can specify several tasks separated by commas. However, the result is confusing:

Name	Last name	Department
Ann	Ten	Quality
Ann	Ten	Address
Ann	Ten	Production
Ann	Ten	Sales
Anxo	Olivenza	Quality
Anxo	Olivenza	Address
Anxo	Olivenza	Production
Anxo	Olivenza	Sales
Arthur	Curly	Quality
Arthur	Curly	Address
Arthur	Curly	Production
Arthur	Curly	Sales
Luisa	Rodriguez	Quality
Luisa	Rodriguez	Address
Luisa	Rodriguez	Production
Luisa	Rodriguez	Sales
Marisa	Lion	Quality
Marisa	Lion	Address
Marisa	Lion	Production
Marisa	Lion	Sales

It appears that all employees work in all departments. This is not true according to the original data.

Ana Díez works in the Sales department, as indicated by her **cod_dep** column.

The reason for this result is that we haven't used the relationship between the tables to link the data from both. Therefore, each row from the first table appears combined with each row from the second. Since the first table (departments) has six rows, and the second (employees) has four, the result is 24 rows.

This method of combining data from multiple tables is known as **cross-multiplication** or **Cartesian multiplication**.

We will usually need to filter that product so that only employee data combined with their department data appears, as is the case here. This is called **joining tables**.

Simple associations

The correct way to perform the above query (associating employees with their departments) would be:

```
SELECT name, surname, department
FROM  departments , employees
WHERE departments.cod_dep = employees.cod_dep
ORDER BY name, surname, department;
```

Both the departments table and the employees table have a column called cod_de p , so it must be distinguished by preceding it with the name of the table to which it belongs.

To avoid repeatedly using the table name, a table alias can be used:

```
SELECT name, surname, department
FROM departments d , employees e
WHERE d .cod_dep= e .cod_dep
ORDER BY name, surname, department;
```

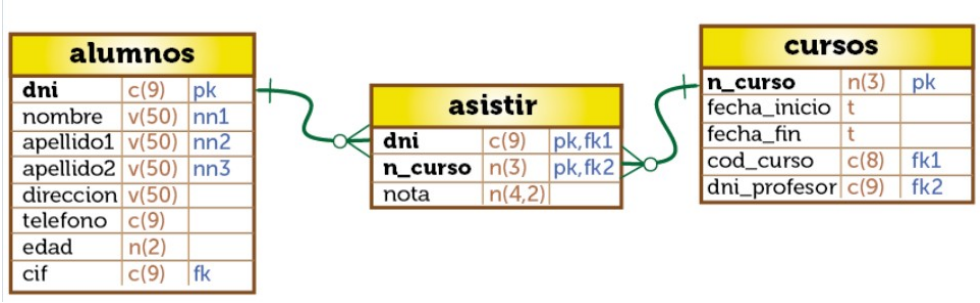
In any case, the result truly shows the employees and the departments they belong to.

Name	Last name	Department
Ann	Ten	Production
Anxo	Olivenza	Address
Arthur	Curly	Sales
Luisa	Rodriguez	Quality
Marisa	Lion	Sales
Pau	Cabanillas	Production

Associate more than one table

It's possible to associate data from more than two tables.

For example, suppose we have these tables, shown from their relational design:



We want to display the students' first and last names along with the course number they are associated with and the start and end dates of those courses.

Although the data we need is in two tables (students and courses), there is no relationship between them. The relationship between these tables is defined by the attend table , so we need to specify that table. The resulting query would be:

```
SELECT name, surname1, surname2,
  c.n_curso, fecha_inicio, fecha_fin
FROM students a, attend s, courses c
WHERE a.dni=s.dni AND c.n_course=s.n_course;
```

SQL syntax 1999

The 1999 version of SQL introduced a new syntax for querying multiple tables. The reason for this was to separate the association conditions from the record selection conditions, thus providing greater clarity to SQL statements.

The complete syntax for queries in SQL 99 format is:

```
SELECT  table1.column1, table1.column2,...
table2.column1, table2.column2,... FROM table1
[ CROSS JOIN table2 ]|
[ NATURAL JOIN table2 ]|
[[ LEFT | RIGHT | FULL OUTER] JOIN table2 USING( column ) ]|
[[ LEFT | RIGHT | FULL OUTER] JOIN table2 ON ( table1.column=table2.column ) ]
```

CROSS JOIN

Using the **CROSS JOIN** option performs a cross product between the specified tables.

Cross products are used for more advanced queries.

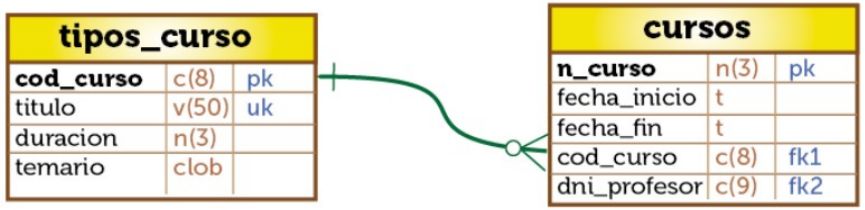
Using the same example seen above, in SQL 99 the cross product would look like this:

```
SELECT name, surname, department
FROM departments
CROSS JOIN employees
ORDER BY name, surname, department;
```

NATURAL JOIN

It establishes an equality relationship between the tables through fields that have the **same name in both tables**. It's not uncommon for the only columns with the same name between two tables to be those that allow you to relate their data.

To better understand the idea, let's look at this example:



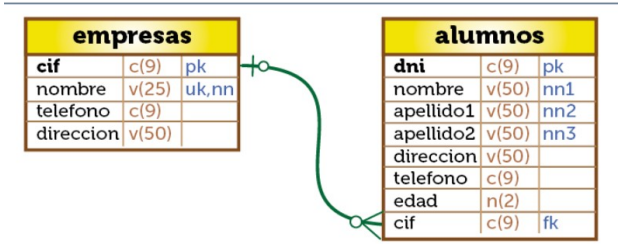
Suppose we want to obtain the title of each course type along with the course numbers associated with that type. We need to use both tables, and we observe that the relationship between the two tables is established by the cod_curso column (present in both tables), and that it is also the only column with the same name in both tables.

In that case, we can use a **NATURAL JOIN** without any problems:

```
SELECT  course_code , title, course_n
FROM course_types t
NATURAL JOIN courses c;
```

NATURAL JOIN

Now, let's look at this other diagram:



If we wanted to obtain the names of the companies and the names of the students, seeing that both tables are related through the column called **cif**, we might be tempted to use this code:

```
SELECT e.name, a.name
FROM companies and
NATURAL JOIN students to;
```

The instruction returns an error for qualifying the name column . The reason is that, in this case, the common column is not only the tax ID number but also the name , phone number , and address.

It is important to remember that **NATURAL JOIN** can only be applied to tables where the only columns that have the same name are those used to relate the tables.

JOIN USING

JOIN USING allows you to associate tables by specifying the columns that relate them, if these have the same name in both tables.

The problem indicated at the previous slide can be solved by **JOIN USING** in this way.

```
SELECT e.name, a.name
FROM companies and
JOIN students to USING (cif);
```

By specifying that the tax identification number (CIF) is the common column, ambiguity arises in the name columns, as it's now necessary to distinguish between company names and student names. The query will function correctly.

Sometimes it's not just one column that's used to establish a relationship, which means specifying all the related columns.

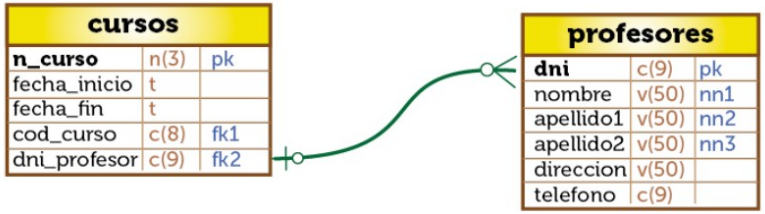
Example:

```
SELECT type, model, storage_number
FROM pieces p
JOIN supplies USING (type,model);
```

JOIN ON

Sometimes tables are related in ways that don't fit any of the previous **JOINs**.

Example:



Suppose we want to obtain the number of each course and the name of the professor who teaches that course. We need to use the courses and professors tables . This involves relating both, and in this case, the column that links them is called **dni** in the professors table and **dni_profesor** in the courses table.

This situation doesn't fit into either a NATURAL JOIN or a JOIN USING. That's what **JOIN ON** is for .

JOIN ON allows you to specify the condition that relates the tables. In this case it would be:

```
SELECT n_curso,nombre
FROM courses c
JOIN professors p ON (dni=dni_profesor);
```

JOIN ON

It should be noted that any **JOIN** can be performed with **JOIN ON** (however, the reverse is not true).

For example, if we have:

```
SELECT e.name, a.name
FROM companies and
JOIN students to USING (cif);
```

We can do an equivalent SELECT like this:

```
SELECT e.name, a.name
FROM companies and
JOIN students ON (e.cif=a.cif);
```

JOIN ON does require qualifying the related columns even if they have the same name.

JOIN ON

The main advantage of **JOIN USING** syntax is the separation between statements about relationships between tables (usually referring to primary keys and related foreign keys) and the conditions that allow you to select rows that meet a specific condition.

For example, in:

```
SELECT p.nombre nombre, p.apellido1, p.apellido2, e.nombre empresa
FROM people p
JOIN companies and USING (cif)
WHERE e.province= 'Palencia' ;
```

The **JOIN** clause links the companies and students tables using the tax ID number (CIF) column present in both tables. This allows us to see the name of the company where each person works. The **WHERE** clause restricts this information so that only people working for companies in Palencia appear .

This query could also be performed in this way:

```
SELECT p.nombre nombre, p.apellido1, p.apellido2, e.nombre empresa
FROM people p
JOIN companies and USING (cif)
AND e.provincia= 'Palencia' ;
```

Internal partnerships or INNER JOIN

Using the above methods of relating tables (including those explained with SQL 92 syntax), only rows present in the related tables appear in the query result.

If we recall the tables with which this unit began, the Departments were:

cod_dep	Department
1	Sales
2	Production
3	Quality
4	Address

And the employees were:

cod_emp	Name	Last name	cod_dep	age
1	Marisa	Lion	1	54
2	Arthur	Curly	1	58
3	Ann	Ten	2	43
4	Pau	Cabanillas	2	29
5	Luisa	Rodriguez	3	34
6	Anxo	Olivenza	4	21
7	Peter	Anderez		40

This SELECT:

```
SELECT name, surname, department
FROM departments d
JOIN employees and USING (dep_code)
ORDER BY name, surname, department;
```

Returns:

Name	Last name	Department
Ann	Ten	Production
Anxo	Olivenza	Address
Arthur	Curly	Sales
Luisa	Rodriguez	Quality
Marisa	Lion	Sales
Pau	Cabanillas	Production

It is noted that *Peter Andérez* does not appear because he is not assigned any department.
This occurs because the **JOIN** clause uses the **INNER** value by default ; that is, it only uses values internal to the relationship.
In other words, only the related rows appear in both tables.

OUTER JOIN clause

However, it is possible to force the appearance of values that are outside the relationship (external, OUTER).

Its syntax is:

```
...
{ LEFT | RIGHT | FULL } OUTER JOIN table
{ ON( condition) | USING (expression) }
...
```

OUTER clause can only be used in **ON** or **USING** type **JOINs**.
LEFT indicates whether we want all the data from the table to the left of the word **JOIN** to appear.
Similarly, if we want the table on the right to display all the data, we indicate **RIGHT** .
Finally, if we want both to display all the data, we use **FULL** .
This query:

```
SELECT name, surname, department
FROM departments d
RIGHT OUTER JOIN employees and USING (cod_emp)
ORDER BY name, surname, department;
```

Sample:

Name	Last name	Department
Ann	Ten	Production
Anxo	Olivenza	Address
Arthur	Curly	Sales
Luisa	Rodriguez	Quality
Marisa	Lion	Sales
Pau	Cabanillas	Production
Peter	Anderez	

Now Pedro Andérez appears.

non-matching queries

A typical use case for external relationships is the use of so-called non-matching queries. These queries allow you to retrieve data from a table that is not related to any other tables.

Using the employees and departments tables above, suppose we need to know which employees are not assigned to a department. The query that achieves this is:

```
SELECT name, surname
FROM departments d
RIGHT OUTER JOIN employees and USING (cod_emp)
WHERE department IS NULL
ORDER BY name, surname, department;
```

Only *Peter Andérez*, the only employee without a department, would appear. The query works because we force all employees to appear and then specify (in the WHERE clause) that it should remove employees who have a department.

relationships without equality

The relationships described in all the previous sections, except those dedicated to cross-multiplication, are called equijoins , since the tables are related through fields that contain the same values in both tables. This is undoubtedly the usual situation.

However, tables don't always have that type of relationship. Let's suppose we have this Employees table :

Name	Salary
Antonio	28000
Marten	41000
Sonia	20000

And, on the other hand, a table of Categories .

Category	Minimum Wage	Maximum salary
Address	50000	72000
Coordination	35000	49999
Skilled position	25000	34999
Unskilled position	18000	24999

In the previous example, we could find out the category to which each employee belongs, in relation to where their salary fits in the category table.

relationships without equality

To determine an employee's category, we need to verify that their salary falls between the minimum and maximum wages.

```
SELECT name, salary, category
FROM employees and categories c
WHERE salary BETWEEN minimum_salary AND b.maximum_salary;
```

Also:

```
SELECT name, salary, category
FROM employees and
JOIN categories c
   ON (salary BETWEEN minimum_salary AND b.maximum_salary);
```

In any case, we obtain:

Name	Salary	Category
Antonio	28000	Skilled worker
Marten	41000	In charge
Sonia	20000	Unskilled position