

UT4. SQL: Lenguaje de Consulta de Datos (DML)

Sitio: [Aula Virtual I.E.S. Ramón Arcas Meca](#)
Curso: SDAW1 – Bases de Datos 2023–24 (Diego Heredia)
Libro: UT4. SQL: Lenguaje de Consulta de Datos (DML)
Imprimido por: ALBERTO CÁNOVAS LÓPEZ
Día: miércoles, 13 de marzo de 2024, 11:11

Tabla de contenidos

- [1. Consultas. Introducción.](#)
- [2. Consultas básicas sobre una tabla](#)
- [3. Consultas sobre varias tablas.](#)
- [4. Consultas Resumen](#)
- [5. Subconsultas](#)

1. Consultas. Introducción.

[Ir al INDICE](#)

Esta Unidad de Trabajo es la **más importante del módulo de Bases de Datos**, durante vuestra vida laboral puede que tengáis que diseñar alguna, muchas o ninguna Base de Datos y es importante saber hacerlo, pero lo que es seguro es que *vais a tener que realizar consultas para extraer datos de Bases de Datos diseñadas por vosotros o por otras personas*.

La vamos a dividir en los siguientes subapartados.

Consultas Básicas sobre una tabla.

Aquí vamos a repasar como se realiza una consulta básica sobre una tabla de nuestra Base de Datos, como ya vimos en la pasada Unidad de Trabajo, pero en esta ocasión, también profundizaremos un poco en algunos aspectos que se quedaron solamente introducidos en la unidad anterior.

Consultas sobre varias tablas. (**Combinaciones y operadores de conjunto**).

En un Sistema Gestor de Bases de Datos, donde cada entidad se representa por una tabla y las relaciones entre ellas son campos comunes entre distintas tablas, no tiene sentido que sólo sepamos realizar consultas sobre una tabla. Debemos conocer como acceder a los datos cuando se encuentra alojados entre distintas tablas de nuestra base de datos. Para ellos tenemos que conocer las **Combinaciones** tanto internas como externas, de eso trata este subapartado.

También veremos los **Operadores de conjunto**, necesarios si queremos acceder a conjuntos de datos que no se encuentran sobre la misma tabla, me explico mejor con un ejemplo:

- Supongamos que tuviéramos una tabla **CLIENTES** y otra **HISTORICO_CLIENTES** y que cuando un cliente deja de utilizarse en el ejercicio actual, se desplaza a la tablas **HISTORICO_CLIENTES** para que el rendimiento del sistema sea mejor.
- Y si ahora necesitamos un **listado de todos los clientes**, tanto los que están activos como los históricos, con una **combinación generariamos una consulta donde el nombre del cliente aparecería en dos columnas diferentes**, lo que necesitamos realmente es que sobre la columna **nombre** aparezcan los nombres de los clientes tanto de **CLIENTES** como de **HISTORICO_CLIENTES**
- Para eso utilizamos los operadores de conjunto: unión, intersección y diferencia

Consultas Resumen.

En ocasiones es necesario agrupar los datos para realizar operaciones estadísticas que nos devuelvan valores que necesitamos, ahora es el momento de profundizar en como se realizan estas tareas.

Subconsultas.

Una herramienta muy potente en SQL es poder sustituir una expresión por el resultado de otra consulta, a eso se conoce como **subconsulta** y las estudiaremos en otro subapartado.

Optimización de Consultas.

Siempre es bueno que nuestras consultas no solo sean correctas, sino que sean lo más rápidas posible, es decir, que tarden el menor tiempo en realizarse. Para eso es necesario conocer algunas técnicas de optimización de las mismas.

Voy a procurar que aprendáis a realizar todas estas consultas sobre al menos **dos Sistemas Gestores de Bases de Datos**, para que podáis apreciar las sutiles diferencias que existen entre ellos, aunque la mayoría de las sentencias son estándar.

Para eso, he seleccionado como SGBD libre y ampliamente utilizado **MySQL/MariaDB** y como sistema comercial **Microsoft SQL Server**, aunque en algún momento pueda hacer alusión a algún otro como es el caso de **Oracle**. Por tanto, **si no se indica lo contrario todos los ejemplos de esta Unidad funcionarán en ambos SGBD**.

Al los recursos, al final de cada subapartado, se incorporarán enlaces al **archivo sql** con el que poder crear la **base de datos** utilizada para los **ejemplos** de estos apuntes.

 **Nota:** espero que aprendáis mucho durante este confinamiento  y que lleguéis a entusiasmáros  por el apasionante mundo del acceso a la información a través de las Bases de Datos.

INDICE GENERAL

UT4.01 Consultas. Introducción.

UT4.02 Consultas básicas sobre una tabla

- [\[1\] Realización de consultas SQL](#)
 - [\[1.1\] El lenguaje DML de SQL](#)

- [2] Consultas básicas sobre una tabla
 - [2.1] Sintaxis de la instrucción SELECT
 - [2.1.1] Sintaxis simplificada SELECT
 - [2.2] Cláusula SELECT
 - [2.2.1] Cómo obtener los datos de todas las columnas de una tabla (SELECT *)
 - [2.2.2] Cómo obtener los datos de algunas columnas de una tabla
 - [2.2.3] Cómo realizar comentarios en sentencias SQL
 - [2.2.4] Cómo obtener columnas calculadas
 - [2.2.5] Cómo realizar alias de columnas con AS
 - [2.2.6] Cómo utilizar funciones en la cláusula SELECT
 - [2.3] Modificadores ALL y DISTINCT
 - [2.4] Cláusula ORDER BY
 - [2.4.1] Cómo ordenar de forma ascendente
 - [2.4.2] Cómo ordenar de forma descendente
 - [2.4.3] Cómo ordenar utilizando múltiples columnas
 - [2.5] Cláusula para limitar el número de filas
 - [2.5.1] SQL Server TOP
 - [2.5.2] MySQL o MariaDB LIMIT
 - Ejercicio LIMIT ?
 - [2.5.3] Oracle ROWNUM
 - [2.6] Cláusula WHERE
 - [2.6.1] Operadores disponibles
 - Ejercicios WHERE ?
 - [2.6.2] Operador BETWEEN
 - Ejercicio BETWEEN ?
 - [2.6.3] Operador IN
 - [2.6.4] Operador LIKE
 - [2.6.5] Operador REGEXP y expresiones regulares
 - [2.6.6] Operadores IS e IS NOT
 - [2.7] Funciones
 - [2.7.1] Funciones en MySQL
 - [2.7.1.1] Funciones con cadenas MySQL
 - [2.7.1.2] Funciones numéricas MySQL
 - [2.7.1.3] Funciones con fechas MySQL
 - [2.7.1.3] Otras funciones MySQL
 - [2.7.2] Funciones en SQL Server
 - [2.7.2.1] Funciones con cadenas SQL Server
 - [2.7.2.2] Funciones numéricas SQL Server
 - [2.7.2.3] Funciones con fechas SQL Server
 - [2.7.2.3] Otras funciones SQL Server
 - [2.7.3] Ejercicios con Funciones
 - Ejercicios FUNCIONES CON CADENAS ?
 - Ejercicios FUNCIONES FECHA Y HORA ?
- [3] Errores comunes
 - [3.1] Error al comprobar si una columna es NULL
 - [3.2] Error al comparar cadenas con patrones utilizando el operador =
 - [3.3] Error al comparar un rango de valores con AND
- [4] Créditos
- [5] Referencias
- [6] Licencia



UT4.03 Consultas sobre varias tablas

- [1] Consultas sobre varias tablas. Composiciones
 - [1.1] Consultas multitable SQL_1
 - [1.1.1] Composiciones cruzadas (Producto cartesiano)
 - [1.1.2] Composiciones internas (Intersección)
 - Ejercicios COMPOSICIONES SQL-1 ?
 - [1.2] Consultas multitable SQL_2
 - [1.2.1] Composiciones cruzadas
 - [1.2.2] Composiciones internas
 - [1.2.3] Composiciones externas

-  [Ejercicios COMPOSICIONES SQL-2?](#)
- [\[1.3\] Operadores de conjunto](#)
 - [\[1.3.1\] Operador UNION](#)
 - [\[1.3.2\] Operador INTERSECT](#)
 - [\[1.3.3\] Operador EXCEPT](#)
 - [\[1.3.4\] Uso de operadores de conjunto](#)
 -  [Ejercicios de Operadores de Conjunto](#)
- [\[1.4\] El orden en las tablas no afecta al resultado final](#)
- [\[1.5\] Podemos usar alias en las tablas](#)
- [\[1.6\] Unir tres o más tablas](#)
 - [\[1.6.1\] Combinaciones de más de dos tablas](#)
 - [\[1.6.2\] Operaciones de conjuntos de más de dos tablas](#)
- [\[1.7\] Utilizar la misma tabla varias veces](#)
- [\[1.8\] Unir una tabla consigo misma \(*self-equi-join*\)](#)
- [\[1.9\] Uniones equivalentes \(*equi-joins*\) y Uniones no equivalentes \(*non-equijoins*\)](#)
- [\[2\] Errores comunes](#)
- [\[3\] Referencias](#)
- [\[4\] Licencia](#)



UT4.04 Consultas Resumen

- [\[1\] Consultas Resumen](#)
 - [\[1.1\] Funciones de agregación](#)
 - [\[1.1.1\] Diferencia entre COUNT\(*\) y COUNT\(columna\)](#)
 - [\[1.1.2\] Contar valores distintos COUNT\(DISTINCT columna\)](#)
 - [\[1.2\] Agrupamiento de filas \(GROUP BY\)](#)
 - [\[1.2.1\] Modificador para crear subtotales ROLLUP](#)
 -  [Ejercicios GROUP BY?](#)
 - [\[1.3\] Condición de agrupamiento \(HAVING\)](#)
 - [\[1.4\] Ejemplo de agrupamiento de filas \(GROUP BY\) con condición de agrupamiento \(HAVING\)](#)
 -  [Ejercicios HAVING?](#)
- [\[2\] Errores comunes](#)
 - [\[2.1\] Error al contar el número de filas distintas](#)
 - [\[2.2\] Error al intentar utilizar una función de agregación en la cláusula WHERE](#)
 - [\[2.3\] Error al usar COUNT\(*\) y COUNT\(columna\) al hacer un LEFT JOIN](#)
- [\[3\] Créditos](#)
- [\[4\] Referencias](#)
- [\[5\] Licencia](#)



UT4.05 Subconsultas

- [\[1\] Subconsultas](#)
 - [\[1.1\] Tipos de subconsultas](#)
 - [\[1.2\] Subconsultas en la cláusula WHERE](#)
 - [\[1.3\] Subconsultas en la cláusula HAVING](#)
 - [\[1.4\] Subconsultas en la cláusula FROM](#)
 - [\[1.5\] Subconsultas en la cláusula SELECT](#)
- [\[2\] Operadores que podemos usar en las subconsultas](#)
 - [\[2.1\] Operadores básicos de comparación](#)
 - [\[2.2\] Subconsultas con ALL y ANY](#)
 - [\[2.3\] Subconsultas con IN y NOT IN](#)
 - [\[2.4\] Subconsultas con EXISTS y NOT EXISTS](#)
- [\[3\] Errores comunes](#)
 - [\[3.1\] Número incorrecto de columnas en la subconsulta](#)
 - [\[3.2\] Número incorrecto de filas en la subconsulta](#)
- [\[4\] Referencias](#)
- [\[5\] Licencia](#)



2. Consultas básicas sobre una tabla

Realizado por Diego Heredia Sánchez

Basado en una obra de [José Juan Sánchez Hernández](#)

- [\[1\] Realización de consultas SQL](#)
 - [\[1.1\] El lenguaje DML de SQL](#)
- [\[2\] Consultas básicas sobre una tabla](#)
 - [\[2.1\] Sintaxis de la instrucción SELECT](#)
 - [\[2.1.1\] Sintaxis simplificada SELECT](#)
 - [\[2.2\] Cláusula SELECT](#)
 - [\[2.2.1\] Cómo obtener los datos de todas las columnas de una tabla \(SELECT *\)](#)
 - [\[2.2.2\] Cómo obtener los datos de algunas columnas de una tabla](#)
 - [\[2.2.3\] Cómo realizar comentarios en sentencias SQL](#)
 - [\[2.2.4\] Cómo obtener columnas calculadas](#)
 - [\[2.2.5\] Cómo realizar alias de columnas con AS](#)
 - [\[2.2.6\] Cómo utilizar funciones en la cláusula SELECT](#)
 - [\[2.3\] Modificadores ALL y DISTINCT](#)
 - [\[2.4\] Cláusula ORDER BY](#)
 - [\[2.4.1\] Cómo ordenar de forma ascendente](#)
 - [\[2.4.2\] Cómo ordenar de forma descendente](#)
 - [\[2.4.3\] Cómo ordenar utilizando múltiples columnas](#)
 - [\[2.5\] Cláusula para limitar el número de filas](#)
 - [\[2.5.1\] SQL Server TOP](#)
 - [\[2.5.2\] MySQL o MariaDB LIMIT](#)
 - [Ejercicio LIMIT ?](#)
 - [\[2.5.3\] Oracle ROWNUM](#)
 - [\[2.6\] Cláusula WHERE](#)
 - [\[2.6.1\] Operadores disponibles](#)
 - [Ejercicios WHERE ?](#)
 - [\[2.6.2\] Operador BETWEEN](#)
 - [Ejercicio BETWEEN ?](#)
 - [\[2.6.3\] Operador IN](#)
 - [\[2.6.4\] Operador LIKE](#)
 - [\[2.6.5\] Operador REGEXP y expresiones regulares](#)
 - [\[2.6.6\] Operadores IS e IS NOT](#)
 - [\[2.7\] Funciones](#)
 - [\[2.7.1\] Funciones en MySQL](#)
 - [\[2.7.1.1\] Funciones con cadenas MySQL](#)
 - [\[2.7.1.2\] Funciones numéricas MySQL](#)
 - [\[2.7.1.3\] Funciones con fechas MySQL](#)
 - [\[2.7.1.3\] Otras funciones MySQL](#)
 - [\[2.7.2\] Funciones en SQL Server](#)
 - [\[2.7.2.1\] Funciones con cadenas SQL Server](#)
 - [\[2.7.2.2\] Funciones numéricas SQL Server](#)
 - [\[2.7.2.3\] Funciones con fechas SQL Server](#)
 - [\[2.7.2.3\] Otras funciones SQL Server](#)
 - [\[2.7.3\] Ejercicios con Funciones](#)
 - [Ejercicios FUNCIONES CON CADENAS ?](#)
 - [Ejercicios FUNCIONES FECHA Y HORA ?](#)
- [\[3\] Errores comunes](#)
 - [\[3.1\] Error al comprobar si una columna es NULL](#)
 - [\[3.2\] Error al comparar cadenas con patrones utilizando el operador =](#)
 - [\[3.3\] Error al comparar un rango de valores con AND](#)
- [\[4\] Créditos](#)
- [\[5\] Referencias](#)
- [\[6\] Licencia](#)

[1] Realización de consultas SQL

[1.1] El lenguaje DML de SQL

El **DML** (*Data Manipulation Language*) o **Lenguaje de Manipulación de Datos** es la parte de SQL dedicada a la manipulación de los datos. Las sentencias **DML** son las siguientes:

- **SELECT**: se utiliza para realizar consultas y extraer información de la base de datos.
- **INSERT**: se utiliza para insertar registros en las tablas de la base de datos.
- **UPDATE**: se utiliza para actualizar los registros de una tabla.
- **DELETE**: se utiliza para eliminar registros de una tabla.

En este apartado nos vamos a centrar en el uso de la sentencia **SELECT**.



[2] Consultas básicas sobre una tabla

[2.1] Sintaxis de la instrucción SELECT

SQL Server :

Según la [documentación oficial de Microsoft](#) la sintaxis para realizar una consulta con la sentencia **SELECT** tanto en **SQL Server** como en **Azure SQL Database** es la siguiente:

```
-- Syntax for SQL Server and Azure SQL Database
<SELECT statement> ::=

[ WITH { [ XMLNAMESPACES ,] [ <common_table_expression> [,...n] ] } ]
<query_expression>
[ ORDER BY { order_by_expression | column_position [ ASC | DESC ] }
[ ,...n ] ]
[ <FOR Clause>]
[ OPTION ( <query_hint> [ ,...n ] ) ]

<query_expression> ::=
{ <query_specification> | ( <query_expression> ) }
[ { UNION [ ALL ] | EXCEPT | INTERSECT }
<query_specification> | ( <query_expression> ) [...n] ]

<query_specification> ::=
SELECT [ ALL | DISTINCT ]
[ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
< select_list >
[ INTO new_table ]
[ FROM { <table_source> } [ ,...n ] ]
[ WHERE <search_condition> ]
[ <GROUP BY> ]
[ HAVING < search_condition > ]
```

MySQL / MariaDB :

Según la [documentación oficial de MySQL](#) esta sería la sintaxis para realizar una consulta con la sentencia **SELECT** en MySQL:

```

SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  [HIGH_PRIORITY]
  [STRAIGHT_JOIN]
  [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr ...]
  [FROM table_references]
  [PARTITION partition_list]
  [WHERE where_condition]
  [GROUP BY
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING having_condition]
  [ORDER BY
    [ASC | DESC], ...]
  [LIMIT]
  [PROCEDURE procedure_name(argument_list)]
  [INTO OUTFILE 'file_name'
    [CHARACTER SET charset_name]
    export_options
    | INTO DUMPFILE 'file_name'
    | INTO var_name [, var_name]]
  [FOR UPDATE | LOCK IN SHARE MODE]]

```



[2.1.1] Sintaxis simplificada SELECT

Para empezar con consultas sencillas podemos simplificar las definiciones anteriores y quedarnos con las siguientes:

```

SELECT [DISTINCT] [TOP] select_expr [, select_expr ...]
  [FROM table_references]
  [WHERE where_condition]
  [GROUP BY [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY [ASC | DESC], ...]
  [LIMIT]

```

Es muy importante conocer **en qué orden se ejecuta cada una de las cláusulas** que forman la sentencia **SELECT**. El orden de ejecución es el siguiente:

- Cláusula **FROM**.
- Cláusula **WHERE** (Es opcional, puede ser que no aparezca).
- Cláusula **GROUP BY** (Es opcional, puede ser que no aparezca).
- Cláusula **HAVING** (Es opcional, puede ser que no aparezca).
- Cláusula **SELECT**.
- Cláusula **ORDER BY** (Es opcional, puede ser que no aparezca).
- Cláusula **TOP** (Es opcional, puede ser que no aparezca, se usa en **SQL Server**).
- Cláusula **LIMIT** (Es opcional, puede ser que no aparezca, se usa en **MySQL/MariaDB**).

Hay que tener en cuenta que el resultado de una consulta siempre será una tabla de datos, que puede tener una o varias columnas y ninguna, una o varias filas.

El hecho de que el resultado de una consulta sea una tabla es muy interesante porque nos permite realizar cosas como almacenar los resultados como una nueva en la base de datos. También será posible combinar el resultado de dos o más consultas para crear una tabla mayor con la unión de los dos resultados. Además, los resultados de una consulta también pueden consultados por otras nuevas consultas.



[2.2] Cláusula SELECT

Nos permite indicar cuáles serán las columnas que tendrá la tabla de resultados de la consulta que estamos realizando. Las opciones que podemos indicar son las siguientes:

- El **nombre de una columna** de la tabla sobre la que estamos realizando la consulta. Será una columna de la tabla que aparece en la cláusula **FROM**.
- Una **constante** que aparecerá en todas las filas de la tabla resultado.
- Una **expresión** que nos permite calcular nuevos valores.



[2.2.1] Cómo obtener los datos de todas las columnas de una tabla (`SELECT *`)

Ejemplo:

Vamos a utilizar la siguiente base de datos de ejemplo para MySQL.

SQL Server :

```
USE master
GO
IF EXISTS (
    SELECT name
    FROM sys.databases
    WHERE name = 'bd_teoria_instituto'
) DROP DATABASE bd_teoria_instituto
GO
CREATE DATABASE bd_teoria_instituto;
GO
USE bd_teoria_instituto
GO
CREATE TABLE alumno (
    id INT IDENTITY(1,1) PRIMARY KEY,
    nombre NVARCHAR(100) NOT NULL,
    apellido1 NVARCHAR(100) NOT NULL,
    apellido2 NVARCHAR(100),
    fecha_nacimiento DATE NOT NULL,
    es_repetidor NVARCHAR(2) NOT NULL
        CHECK (es_repetidor IN ('sí', 'no')),
    teléfono NVARCHAR(9)
);

INSERT INTO alumno VALUES('María', 'Sánchez', 'Pérez', '1990/12/01', 'no', NULL);
INSERT INTO alumno VALUES('Juan', 'Sáez', 'Vega', '1998/04/02', 'no', 618253876);
INSERT INTO alumno VALUES('Pepe', 'Ramírez', 'Gea', '1988/01/03', 'no', NULL);
INSERT INTO alumno VALUES('Lucía', 'Sánchez', 'Ortega', '1993/06/13', 'sí', 678516294);
INSERT INTO alumno VALUES('Paco', 'Martínez', 'López', '1995/11/24', 'no', 692735409);
INSERT INTO alumno VALUES('Irene', 'Gutiérrez', 'Sánchez', '1991/03/28', 'sí', NULL);
INSERT INTO alumno VALUES('Cristina', 'Fernández', 'Ramírez', '1996/09/17', 'no', 628349590);
INSERT INTO alumno VALUES('Antonio', 'Carretero', 'Ortega', '1994/05/20', 'sí', 612345633);
INSERT INTO alumno VALUES('Manuel', 'Domínguez', 'Hernández', '1999/07/08', 'no', NULL);
INSERT INTO alumno VALUES('Daniel', 'Moreno', 'Ruiz', '1998/02/03', 'no', NULL);
```

MySQL / MariaDB :

```
DROP DATABASE IF EXISTS instituto;
CREATE DATABASE instituto CHARACTER SET utf8mb4;
USE instituto;

CREATE TABLE alumno (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    apellido1 VARCHAR(100) NOT NULL,
    apellido2 VARCHAR(100),
    fecha_nacimiento DATE NOT NULL,
    es_repetidor ENUM('sí', 'no') NOT NULL,
    teléfono VARCHAR(9)
);

INSERT INTO alumno VALUES(1, 'María', 'Sánchez', 'Pérez', '1990/12/01', 'no', NULL);
INSERT INTO alumno VALUES(2, 'Juan', 'Sáez', 'Vega', '1998/04/02', 'no', 618253876);
INSERT INTO alumno VALUES(3, 'Pepe', 'Ramírez', 'Gea', '1988/01/03', 'no', NULL);
INSERT INTO alumno VALUES(4, 'Lucía', 'Sánchez', 'Ortega', '1993/06/13', 'sí', 678516294);
INSERT INTO alumno VALUES(5, 'Paco', 'Martínez', 'López', '1995/11/24', 'no', 692735409);
INSERT INTO alumno VALUES(6, 'Irene', 'Gutiérrez', 'Sánchez', '1991/03/28', 'sí', NULL);
INSERT INTO alumno VALUES(7, 'Cristina', 'Fernández', 'Ramírez', '1996/09/17', 'no', 628349590);
INSERT INTO alumno VALUES(8, 'Antonio', 'Carretero', 'Ortega', '1994/05/20', 'sí', 612345633);
INSERT INTO alumno VALUES(9, 'Manuel', 'Domínguez', 'Hernández', '1999/07/08', 'no', NULL);
INSERT INTO alumno VALUES(10, 'Daniel', 'Moreno', 'Ruiz', '1998/02/03', 'no', NULL);
```

Supongamos que tenemos una tabla llamada **alumno** con la siguiente información de los alumnos matriculados en un determinado curso.

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
2	Juan	Sáez	Vega	1998-04-02	no	618253876
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
5	Paco	Martínez	López	1995-11-24	no	692735409
6	Irene	Gutiérrez	Sánchez	1991-03-28	sí	NULL
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
8	Antonio	Carretero	Ortega	1994-05-20	sí	612345633
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL

Vamos a ver qué consultas sería necesario realizar para obtener los siguientes datos.

1. Obtener todos los datos de todos los alumnos matriculados en el curso.

```
SELECT *
FROM alumno;
```

El carácter * es un comodín que utilizamos para indicar que queremos seleccionar todas las columnas de la tabla. La consulta anterior devolverá todos los datos de la tabla.

Tenga en cuenta que las palabras reservadas de SQL no son *case sensitive*, por lo tanto es posible escribir la sentencia anterior de la siguiente forma obteniendo el mismo resultado:

```
select *
from alumno;
```

Otra consideración que también debemos tener en cuenta es que una consulta SQL se puede escribir en una o varias líneas. Por ejemplo, la siguiente sentencia tendría el mismo resultado que la anterior:

```
SELECT * FROM alumno;
```

A lo largo del curso vamos a considerar como una buena práctica escribir las consultas SQL en varias líneas, empezando cada línea con la palabra reservada de la cláusula correspondiente que forma la consulta.



[2.2.2] Cómo obtener los datos de algunas columnas de una tabla

2. Obtener el nombre de todos los alumnos.

```
SELECT nombre
FROM alumno;
```

nombre
María
Juan
Pepe
Lucía
Paco
Irene
Cristina
Antonio
Manuel
Daniel

3. Obtener el nombre y los apellidos de todos los alumnos.

```
SELECT nombre, apellido1, apellido2
FROM alumno;
```

nombre	apellido1	apellido2
María	Sánchez	Pérez
Juan	Sáez	Vega
Pepe	Ramírez	Gea
Lucía	Sánchez	Ortega
Paco	Martínez	López
Irene	Gutiérrez	Sánchez
Cristina	Fernández	Ramírez
Antonio	Carretero	Ortega
Manuel	Domínguez	Hernández
Daniel	Moreno	Ruiz

Tenga en cuenta que el resultado de la consulta SQL mostrará las columnas que haya solicitado, siguiendo el orden en el que se hayan indicado. Por lo tanto la siguiente consulta:

```
SELECT apellido1, apellido2, nombre
FROM alumno;
```

Devolverá lo siguiente:

apellido1	apellido2	nombre
Sánchez	Pérez	María
Sáez	Vega	Juan
Ramírez	Gea	Pepe
Sánchez	Ortega	Lucía
Martínez	López	Paco
Gutiérrez	Sánchez	Irene
Fernández	Ramírez	Cristina
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Moreno	Ruiz	Daniel



[2.2.3] Cómo realizar comentarios en sentencias SQL

Para escribir comentarios en nuestras sentencias SQL podemos hacerlo de diferentes formas.

```
-- Esto es un comentario
SELECT nombre, apellido1, apellido2 -- Esto es otro comentario
FROM alumno;
```

```
/* Esto es un comentario
de varias líneas */
SELECT nombre, apellido1, apellido2 /* Esto es otro comentario */
FROM alumno;
```

Nota: En MySQL/MariaDB también se puede usar el carácter # como inicio de comentario de una sola línea, pero esto provocaría un error en SQL Server

```
# Esto es un comentario
SELECT nombre, apellido1, apellido2 # Esto es otro comentario
FROM alumno;
```



[2.2.4] Cómo obtener columnas calculadas

Ejemplo

Es posible realizar cálculos aritméticos entre columnas para calcular nuevos valores. Por ejemplo, supongamos que tenemos la siguiente tabla con información sobre ventas.

id	cantidad_comprada	precio_unitario
1	2	1,50
2	5	1,75
3	7	2,00

Y queremos calcular una nueva columna con el precio total de la venta, que sería equivalente a multiplicar el valor de la columna `cantidad_comprada` por `precio_unitario`.

Para obtener esta nueva columna podríamos realizar la siguiente consulta:

```
SELECT id, cantidad_comprada, precio_unitario, cantidad_comprada * precio_unitario
FROM ventas;
```

Y el resultado sería el siguiente:

id	cantidad_comprada	precio_unitario	(No column name)
1	2	1,50	3,00
2	5	1,75	8,75
3	7	2,00	14,00

Ejemplo

Supongamos que tenemos una tabla llamada `oficinas` que contiene información sobre las ventas reales que ha generado y el valor de ventas esperado, y nos gustaría conocer si las oficinas han conseguido el objetivo propuesto, y si están por debajo o por encima del valor de ventas esperado.

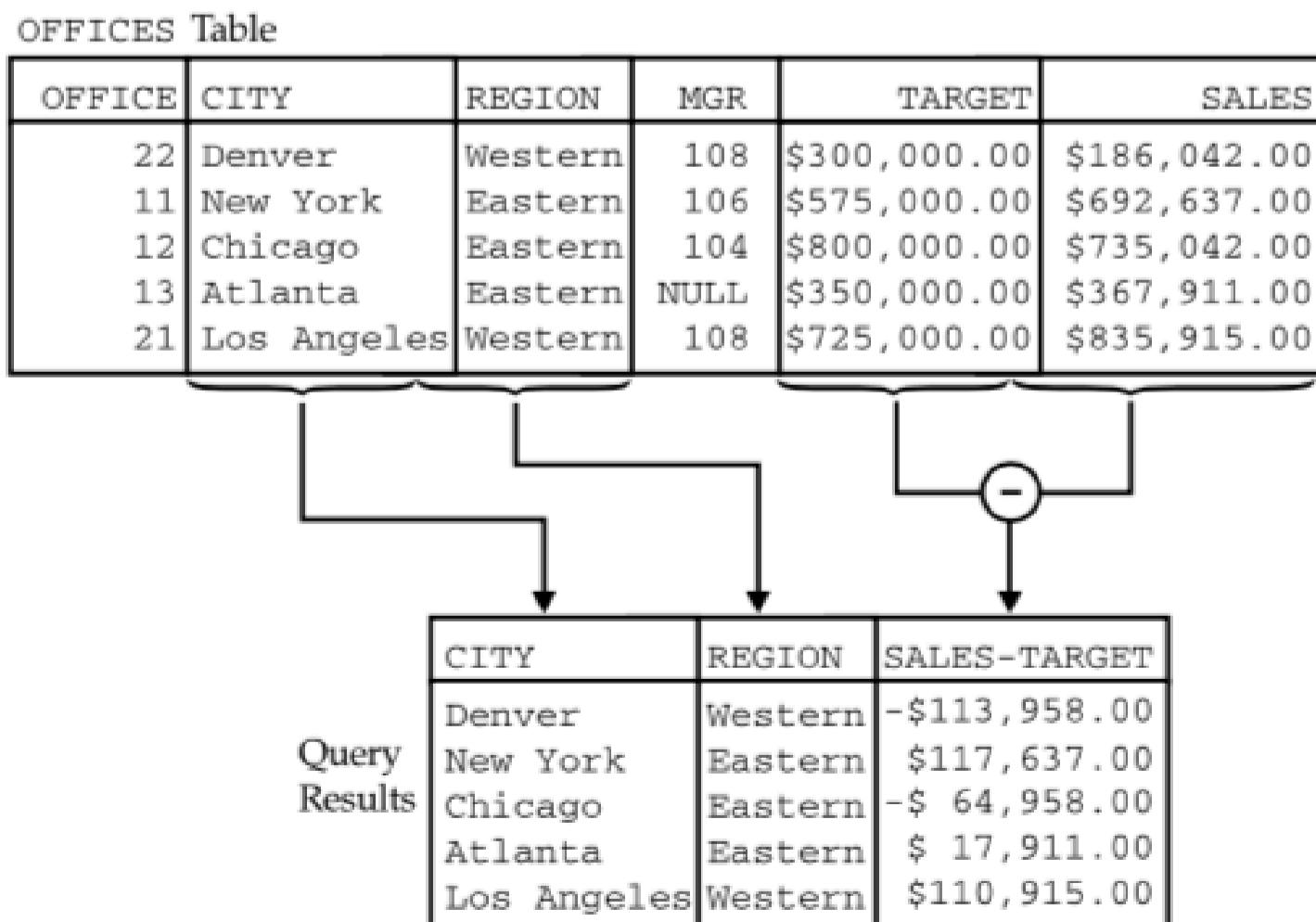


Imagen: Imagen extraída del libro SQL: The Complete Reference* de James R. Groff y otros.*

En este caso podríamos realizar la siguiente consulta:

```
SELECT ciudad, region, ventas - objetivo
FROM oficinas;
```



[2.2.5] Cómo realizar alias de columnas con AS

Con la palabra reservada **AS** podemos crear alias para las columnas. Esto puede ser útil cuando estamos calculando nuevas columnas a partir de valores de las columnas actuales. En el ejemplo anterior de la tabla que contiene información sobre las ventas, podríamos crear el siguiente alias:

```
SELECT id, cantidad_comprada, precio_unitario, cantidad_comprada * precio_unitario AS 'precio total'
FROM ventas;
```

Si el nuevo nombre que estamos creando para el alias contiene espacios en blanco es necesario usar comillas simples.

Al crear este alias obtendremos el siguiente resultado:

id	cantidad_comprada	precio_unitario	precio total
1	2	1,50	3,00
2	5	1,75	8,75
3	7	2,00	14,00

Nota: La palabra reservada **AS** es opcional, con separar el nombre de la columna del alias por un espacio en blanco, sería suficiente.



[2.2.6] Cómo utilizar funciones en la cláusula SELECT

Es posible hacer uso de funciones específicas en la cláusula **SELECT**. Cada Sistema Gestor de Bases de Datos (SGBD) nos ofrece funciones matemáticas, funciones para trabajar con cadenas y funciones para trabajar con fechas y horas. Algunos ejemplos de las funciones que utilizaremos a lo largo del curso son las siguientes.

Pondremos en minúsculas las de SQL Server y en mayúsculas las de MySQL/MariaDB, pero como sabéis, se trata de notación *case-insensitive*.

- **Funciones con cadenas**

MySQL/MariaDB	SQL Server	Descripción
CONCAT()	concat()	Concatena cadenas
CONCAT_WS()	concat_ws()	Concatena cadenas con un separador
LOWER() / LCASE()	lower()	Devuelve una cadena en minúscula
UPPER() / UCASE()	upper()	Devuelve una cadena en mayúscula
SUBSTRING() / SUBSTR()	substring()	Devuelve una subcadena

- **Funciones matemáticas**

MySQL/MariaDB	SQL Server	Descripción
ABS()	abs()	Devuelve el valor absoluto
POW(x,y)	power(x,y)	Devuelve el valor de x elevado a y
SQRT()	sqrt()	Devuelve la raíz cuadrada
PI()	pi()	Devuelve el valor del número PI
ROUND()	round(valor, pos, 0) / round(valor, pos)	Redondea un valor numérico el número de posiciones indicado
TRUNCATE()	round(valor, pos, 1)	Trunca un valor numérico desde la posición indicada

- **Funciones de fecha y hora**

MySQL/MariaDB	SQL Server	Descripción
NOW()	getdate()	Devuelve la fecha y la hora actual

MySQL/MariaDB

SQL Server

Descripción

CURTIME()

convert(time, getdate())

Devuelve la hora actual

 **Nota:** En la documentación oficial puede encontrar la **referencia completa de todas las funciones y operadores disponibles.** * [Funciones en MySQL/MariaDB](#). * [Funciones en SQL Server](#).

Ejemplo

Obtener el nombre y los apellidos de todos los alumnos en una única columna.

```
SELECT CONCAT(nombre, apellido1, apellido2) AS nombre_completo
FROM alumno;
```

nombre_completo
MaríaSánchezPérez
JuanSáezVega
PepeRamírezGea
LucíaSánchezOrtega
PacoMartínezLópez
IreneGutiérrezSánchez
CristinaFernándezRamírez
AntonioCarreteroOrtega
ManuelDomínguezHernández
DanielMorenoRuiz

En este caso estamos haciendo uso de la [función **CONCAT**] y la palabra reservada **AS** para crear un **alias de la columna** y renombrarla como *nombre_completo*.

La [función **CONCAT**] no añade ningún espacio entre las columnas, por eso los valores de las tres columnas aparecen como una sola cadena sin espacios entre ellas. Para resolver este problema podemos hacer uso de [la función **CONCAT_WS**](#) que nos permite definir un carácter separador entre cada columna. En el siguiente ejemplo haremos uso de la [función **CONCAT_WS**](#) y usaremos un espacio en blanco como separador.

```
SELECT CONCAT_WS(' ', nombre, apellido1, apellido2) AS nombre_completo
FROM alumno;
```

nombre_completo
María Sánchez Pérez
Juan Sáez Vega
Pepe Ramírez Gea
Lucía Sánchez Ortega
Paco Martínez López
Irene Gutiérrez Sánchez
Cristina Fernández Ramírez
Antonio Carretero Ortega
Manuel Domínguez Hernández
Daniel Moreno Ruiz

 **Atención!**: En **MySQL/MariaDB** La función **CONCAT()** devolverá **NULL** cuando alguna de las cadenas que está concatenando es igual **NULL**, mientras que la función **CONCAT_WS()** omitirá todas las cadenas que sean igual a **NULL** y realizará la concatenación con el resto de cadenas.

 **Nota:** En **SQL Server** el funcionamiento de ambas funciones **concat()** y **concat_ws()** es similar y lo que hace es realizar la concatenación omitiendo los valores **NULL**.



[2.3] Modificadores ALL y DISTINCT

Los modificadores **ALL** y **DISTINCT** indican si se deben incluir o no filas repetidas en el resultado de la consulta.

- **ALL** indica que se deben incluir todas las filas, incluidas las repetidas. Es la opción por defecto, por lo tanto no es necesario indicarla.
- **DISTINCT** elimina las filas repetidas en el resultado de la consulta.
- **DISTINCTROW** es un sinónimo de **DISTINCT**(en MySQL/MariaDB).



[2.4] Cláusula ORDER BY

ORDER BY permite ordenar las filas que se incluyen en el resultado de la consulta. La sintaxis de MySQL es la siguiente:

```
[ORDER BY [ASC | DESC], ...]
```

Esta cláusula nos permite ordenar el resultado de forma ascendente **ASC** o descendente **DESC**, además de permitirnos ordenar por varias columnas estableciendo diferentes niveles de ordenación.

Nota: Se pueden indicar en lugar del nombre de la columna, el **número de orden** que ocupa dicha columna. Esto es **útil cuando por ejemplo deseamos ordenar por un campo calculado**. Por ejemplo, la siguiente sentencia, ordenará por el **nombre** del alumno

```
SELECT *
FROM alumno
ORDER BY 2
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
8	Antonio	Carretero	Ortega	1994-05-20	sí	612345633
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL
6	Irene	Gutiérrez	Sánchez	1991-03-28	sí	NULL
2	Juan	Sáez	Vega	1998-04-02	no	618253876
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
1	María	Sánchez	Pérez	1990-12-01	no	NULL
5	Paco	Martínez	López	1995-11-24	no	692735409
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL



[2.4.1] Cómo ordenar de forma ascendente

Ejemplo

1. Obtener el nombre y los apellidos de todos los alumnos, ordenados por su primer apellido de forma ascendente.

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1;
```

Si no indicamos nada en la cláusula **ORDER BY** se ordenará por defecto de forma ascendente.

La consulta anterior es equivalente a esta otra.

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1 ASC;
```

El resultado de ambas consultas será:

apellido1	apellido2	nombre
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Fernández	Ramírez	Cristina
Gutiérrez	Sánchez	Irene
Martínez	López	Paco
Moreno	Ruiz	Daniel
Ramírez	Gea	Pepe
Sáez	Vega	Juan
Sánchez	Pérez	María
Sánchez	Ortega	Lucía

Las filas están ordenadas correctamente por el primer apellido, pero todavía hay que resolver cómo ordenar el listado cuando existen varias filas donde coincide el valor del primer apellido. En este caso tenemos dos filas donde el primer apellido es **Sánchez**:

apellido1	apellido2	nombre
Sánchez	Pérez	María
Sánchez	Ortega	Lucía

Más adelante veremos cómo podemos ordenar el listado teniendo en cuenta más de una columna.



[2.4.2] Cómo ordenar de forma descendente

Ejemplo

1. Obtener el nombre y los apellidos de todos los alumnos, ordenados por su primer apellido de forma ascendente.

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1 DESC;
```

apellido1	apellido2	nombre
Sánchez	Ortega	Lucía
Sánchez	Pérez	María
Sáez	Vega	Juan
Ramírez	Gea	Pepe
Moreno	Ruiz	Daniel
Martínez	López	Paco
Gutiérrez	Sánchez	Irene
Fernández	Ramírez	Cristina
Domínguez	Hernández	Manuel
Carretero	Ortega	Antonio



[2.4.3] Cómo ordenar utilizando múltiples columnas

Ejemplo

En este ejemplo vamos a obtener un listado de todos los alumnos ordenados por el primer apellido, segundo apellido y nombre, de forma ascendente.

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1, apellido2, nombre;
```

Como adelantamos anteriormente, en lugar de indicar el nombre de las columnas en la cláusula **ORDER BY** podemos indicar sobre la posición donde aparecen en la cláusula **SELECT**, de modo que la consulta anterior sería equivalente a la siguiente:

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY 1, 2, 3;
```

El resultado de ambas consultas será:

apellido1	apellido2	nombre
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Fernández	Ramírez	Cristina
Gutiérrez	Sánchez	Irene
Martínez	López	Paco
Moreno	Ruiz	Daniel
Ramírez	Gea	Pepe
Sáez	Vega	Juan
Sánchez	Ortega	Lucía
Sánchez	Pérez	María



[2.5] Cláusula para limitar el número de filas

Cuando el número de filas devueltas en el resultado de nuestra consulta, es mayor de lo que deseamos, podemos limitar la cantidad resultante. Cada Sistema Gestor de Bases de Datos (SGBD) lo puede hacer de una manera diferente.

[2.5.1] SQL Server TOP

Utiliza la palabra reservada **TOP** justo después de la palabra **SELECT** o **DISTINCT** (*si existe*), que permite limitar el número de filas que se incluyen en el resultado de la consulta. La sintaxis de *SQL Server* es la siguiente

```
[TOP ( expression ) [PERCENT] [ WITH TIES ] ]
```

Donde:

- **expression**: debe ser un número que indica la cantidad de filas. **TOP 1** indica que deseamos obtener sólo una fila.
- **PERCENT**: se utiliza para indicar que **expression** no es un número de filas, sino un **porcentaje** de las filas que forman el conjunto de resultados.
- **WITH TIES**: nos indica que si hay elementos repetidos, se incluyan aunque esto suponga exceder el número de filas indicado por **expression**.

Ejemplo

Vamos a obtener el 20% de los alumnos (*en nuestras tablas serían 2*) ordenados descendentes por el id de alumno **id**

```
SELECT TOP 20 PERCENT *
FROM alumno
ORDER BY id DESC
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL



[2.5.2] MySQL o MariaDB LIMIT

Utiliza la palabra reservada **LIMIT** al final de la instrucción, que permite limitar el número de filas que se incluyen en el resultado de la consulta. La sintaxis de *MySQL* es la siguiente

```
[LIMIT row_COUNT]
| [LIMIT offset , row_COUNT]
```

Donde `row_COUNT` es el número de filas que queremos obtener y `offset` el número de filas que nos saltamos (*desplazamiento*) antes de empezar a contar. Es decir, la primera fila que se obtiene como resultado es la que está situada en la posición `offset + 1`.

 **Nota:** `LIMIT 1` es lo mismo que `LIMIT 0 , 1`

Ejemplo

Vamos a obtener el primer alumno cuyo primer apellido empiece por M, ordenando ascendentemente por `apellido1`, `apellido2` y `nombre`

```
SELECT *
FROM alumno
WHERE apellido1 like 'M%'
ORDER BY apellido1, apellido2, nombre
LIMIT 1;
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
5	Paco	Martínez	López	1995-11-24	no	692735409

 **Nota:** Introducimos aquí la cláusula `WHERE` que estudiaremos a continuación.



Ejercicio LIMIT ?:

Construye la siguiente base de datos en **MySQL** o **MariaDB** y responde a las preguntas formuladas a continuación:

```
DROP DATABASE IF EXISTS google;
CREATE DATABASE google CHARACTER SET utf8mb4;
USE google;

CREATE TABLE resultado (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    descripcion VARCHAR(200) NOT NULL,
    url VARCHAR(512) NOT NULL
);

INSERT INTO resultado VALUES (1, 'Resultado 1', 'Descripción 1', 'http://....');
INSERT INTO resultado VALUES (2, 'Resultado 2', 'Descripción 2', 'http://....');
INSERT INTO resultado VALUES (3, 'Resultado 3', 'Descripción 3', 'http://....');
INSERT INTO resultado VALUES (4, 'Resultado 4', 'Descripción 4', 'http://....');
INSERT INTO resultado VALUES (5, 'Resultado 5', 'Descripción 5', 'http://....');
INSERT INTO resultado VALUES (6, 'Resultado 6', 'Descripción 6', 'http://....');
INSERT INTO resultado VALUES (7, 'Resultado 7', 'Descripción 7', 'http://....');
INSERT INTO resultado VALUES (8, 'Resultado 8', 'Descripción 8', 'http://....');
INSERT INTO resultado VALUES (9, 'Resultado 9', 'Descripción 9', 'http://....');
INSERT INTO resultado VALUES (10, 'Resultado 10', 'Descripción 10', 'http://....');
INSERT INTO resultado VALUES (11, 'Resultado 11', 'Descripción 11', 'http://....');
INSERT INTO resultado VALUES (12, 'Resultado 12', 'Descripción 12', 'http://....');
INSERT INTO resultado VALUES (13, 'Resultado 13', 'Descripción 13', 'http://....');
INSERT INTO resultado VALUES (14, 'Resultado 14', 'Descripción 14', 'http://....');
INSERT INTO resultado VALUES (15, 'Resultado 15', 'Descripción 15', 'http://....');
```

Responde ahora a estas preguntas:

1. Suponga que queremos mostrar en una página web las filas que están almacenadas en la tabla `resultados`, y queremos mostrar la información en diferentes páginas, donde cada una de las páginas muestra solamente 5 resultados. ¿Qué consulta SQL necesitamos realizar para incluir los primeros 5 resultados de la primera página?
2. ¿Qué consulta necesitaríamos para mostrar resultados de la segunda página?
3. ¿Y los resultados de la tercera página?

4. ¿Hay alguna forma que hacer algo similar a lo conseguido con la cláusula **LIMIT** en **SQL Server**? Adapta la construcción de la base de datos **google** para su ejecución en **SQL Server** y si fuera posible, realiza los ejercicios anteriores para este Sistema Gestor de Bases de Datos (SGBD).



[2.5.3] Oracle ROWNUM

En Oracle se utiliza una sintaxis diferente mediante la palabra reservada **ROWNUM** que indica el número de la fila en el conjunto de resultados, por tanto, se podría indicar que deseamos limitar la consulta a las **number** primeras filas, de la siguiente forma:

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



[2.6] Cláusula WHERE

La cláusula **WHERE** nos permite añadir filtros a nuestras consultas para seleccionar sólo aquellas filas que cumplen una determinada condición. Estas condiciones se denominan **predicados** y el resultado de estas condiciones puede ser **verdadero**, **falso** o **desconocido**.

Una condición tendrá un resultado **desconocido** cuando alguno de los valores utilizados tiene el valor **NULL**.

Podemos diferenciar cinco tipos de condiciones que pueden aparecer en la cláusula **WHERE**:

- Condiciones para comparar valores o expresiones.
- Condiciones para comprobar si un valor está dentro de un rango de valores.
- Condiciones para comprobar si un valor está dentro de un conjunto de valores.
- Condiciones para comparar cadenas con patrones.
- Condiciones para comprobar si una columna tiene valores a **NULL**.

Los operandos usados en las condiciones pueden ser nombres de columnas, constantes o expresiones. Los operadores que podemos usar en las condiciones pueden ser aritméticos, de comparación, lógicos, etc.



[2.6.1] Operadores disponibles

A continuación se muestran los operadores más utilizados en *MySQL* para realizar las consultas y su equivalente en *SQL Server*, donde apenas hay diferencias.

Operadores aritméticos

MySQL/MariaDB	SQL Server	Descripción
+	+	Suma
-	-	Resta
*	*	Multiplicación
/	/	División
%	%	Módulo

Operadores de comparación

MySQL/MariaDB	SQL Server	Descripción
<	<	Menor que
<=	<=	Menor o igual
>	>	Mayor que
>=	>=	Mayor o igual
<>	<>	Distinto

MySQL/MariaDB	SQL Server	Descripción
!=	!=	Distinto
=	=	Igual que

Operadores lógicos

MySQL/MariaDB	SQL Server	Descripción
AND	AND	Y lógica
&&		Y lógica
OR	OR	O lógica
		O lógica
NOT	NOT	Negación lógica
!		Negación lógica

Ejemplos

Vamos a continuar con el ejemplo de la tabla **alumno** que almacena la información de los alumnos matriculados en un determinado curso.

¿Qué consultas serían necesarias para obtener los siguientes datos?

1. Obtener el nombre de todos los alumnos que su primer apellido sea Martínez.

```
SELECT nombre
FROM alumno
WHERE apellido1 = 'Martínez';
```

2. Obtener todos los datos del alumno que tiene un **id** igual a 9.

```
SELECT *
FROM alumno
WHERE id = 9;
```

3. Obtener el nombre y la fecha de nacimiento de todos los alumnos nacieron después del 1 de enero de 1997.

```
SELECT nombre, fecha_nacimiento
FROM alumno
WHERE fecha_nacimiento >= '1997-01-01';
```

nombre	fecha_nacimiento
Juan	1998-04-02
Manuel	1999-07-08
Daniel	1998-02-03

 **Nota:** Como verás, las fechas se escriben como cadenas de caracteres encerradas entre comillas simples, donde en primer lugar se introduce el año, después el mes y por último el día. También se habría podido indicar con '**1997/01/01**'



Ejercicios WHERE :

Realice las siguientes consultas teniendo en cuenta la **base de datos instituto**.

1. Devuelve los datos del alumno cuyo **id** es igual a 1.

2. Devuelve los datos del alumno cuyo **teléfono** es igual a 692735409.

3. Devuelve un listado de todos los alumnos que son repetidores.

4. Devuelve un listado de todos los alumnos que no son repetidores.

5. Devuelve el listado de los alumnos que han nacido antes del 1 de enero de 1993.

6. Devuelve el listado de los alumnos que han nacido después del 1 de enero de 1994.

7. Devuelve el listado de los alumnos que han nacido después del 1 de enero de 1994 y no son repetidores.

8. Devuelve el listado de todos los alumnos que nacieron en 1998.

9. Devuelve el listado de todos los alumnos que **no** nacieron en 1998.



[2.6.2] Operador BETWEEN

Sintaxis:

```
expresión [NOT] BETWEEN cota_inferior AND cota_superior
```

Se utiliza para comprobar si un valor está dentro de un rango de valores. Por ejemplo, si queremos obtener los pedidos que se han realizado durante el mes de enero de 2018 podemos realizar la siguiente consulta:

```
SELECT *
FROM pedido
WHERE fecha_pedido BETWEEN '2018-01-01' AND '2018-01-31';
```



Ejercicio BETWEEN ?:

Realice las siguientes consultas teniendo en cuenta la **base de datos instituto**.

1. Devuelve los datos de los alumnos que hayan nacido entre el 1 de enero de 1998 y el 31 de mayo de 1998.



[2.6.3] Operador IN

Este operador nos permite comprobar si el valor de una determinada columna, (*o el resultado de una expresión*) está incluido en una lista de valores.

Ejemplo:

Obtener todos los datos de los alumnos que tengan como primer apellido **Sánchez, Martínez o Domínguez**.

```
SELECT *
FROM alumno
WHERE apellido1 IN ('Sánchez', 'Martínez', 'Domínguez');
```

Ejemplo:

Obtener todos los datos de los alumnos que **no tengan** como primer apellido **Sánchez, Martínez o Domínguez**.

```
SELECT *
FROM alumno
WHERE apellido1 NOT IN ('Sánchez', 'Martínez', 'Domínguez');
```



[2.6.4] Operador LIKE

Sintaxis:

```
columna [NOT] LIKE patrón
```

Se utiliza para comparar si una cadena de caracteres coincide con un patrón. En el patrón podemos utilizar cualquier carácter alfanumérico, pero hay dos caracteres que tienen un significado especial, el símbolo del porcentaje (%) y el carácter de subrayado (_).

- %: Este carácter equivale a cualquier conjunto de caracteres.
- _: Este carácter equivale a cualquier carácter.

Ejemplos:

Devuelva un listado de todos los alumnos que su primer apellido empiece por la letra S.

```
SELECT *
FROM alumno
WHERE apellido1 LIKE 'S%';
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
2	Juan	Sáez	Vega	1998-04-02	no	618253876
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294

Devuelva un listado de todos los alumnos que su primer apellido termine por la letra z.

```
SELECT *
FROM alumno
WHERE apellido1 LIKE '%z';
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
2	Juan	Sáez	Vega	1998-04-02	no	618253876
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
5	Paco	Martínez	López	1995-11-24	no	692735409
6	Irene	Gutiérrez	Sánchez	1991-03-28	sí	NULL
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL

Devuelva un listado de todos los alumnos que su nombre tenga el carácter a.

```
SELECT *
FROM alumno
WHERE nombre LIKE '%a%';
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
2	Juan	Sáez	Vega	1998-04-02	no	618253876
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
5	Paco	Martínez	López	1995-11-24	no	692735409
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
8	Antonio	Carretero	Ortega	1994-05-20	sí	612345633
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL

Devuelva un listado de todos los alumnos que tengan un nombre de cuatro caracteres.

```
SELECT *
FROM alumno
WHERE nombre LIKE '____';
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
2	Juan	Sáez	Vega	1998-04-02	no	618253876
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
5	Paco	Martínez	López	1995-11-24	no	692735409

Devuelve un listado de todos los productos cuyo nombre empieza con estas cuatro letras ‘A%BC’.

En este caso, el patrón que queremos buscar contiene el carácter %, por lo tanto, tenemos que usar un carácter de escape.

```
SELECT *
FROM producto
WHERE nombre LIKE 'A$%BC%' ESCAPE '$';
```

En MySQL/MariaDB por defecto, se utiliza el carácter ‘\’ como carácter de escape. De modo que podríamos escribir la consulta de la siguiente manera.

```
SELECT *
FROM producto
WHERE nombre LIKE 'A\%BC%';
```

⚠️ Atención!: usar el carácter de escape por defecto sin indicarlo explicitamente, no funcionará en SQL Server, por lo que deberemos usar **ESCAPE '\'** si queremos emplearlo.

Nota: SQL Server adicionalmente nos permite especificar un conjunto de caracteres encerrados entre corchetes [], o la NO coincidencia con cuaquiera de los caracteres encerrados entre corchetes con [^caracteres], ([más información sobre los caracteres comodín en SQL Server, en la documentación oficial](#)). Veamoslo en el siguiente ejemplo:

Selecciona aquellos alumnos cuya segunda letra en su nombre sea una a o una e

```
SELECT *
FROM alumno
WHERE nombre LIKE '_[ae]%' ;
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
5	Paco	Martínez	López	1995-11-24	no	692735409
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL

En MySQL/MariaDB se podría hacer así:

```
SELECT *
FROM alumno
WHERE nombre LIKE '_a%' OR nombre LIKE '_e%' ;
```



[2.6.5] Operador REGEXP y expresiones regulares

El operador **REGEXP** de MySQL/MariaDB nos permite realizar búsquedas mucho más potentes haciendo uso de expresiones regulares. Puede consultar la [documentación oficial](#) para conocer más detalles sobre cómo usar este operador.

Nota: en SQL Server también se podrían usar expresiones regulares haciendo uso de procedimientos almacenados.



[2.6.6] Operadores IS e IS NOT

Estos operadores nos permiten comprobar si el valor de una determinada columna es **NULL** o no lo es.

Ejemplo:

Obtener la lista de alumnos que tienen un valor **NULL** en la columna **teléfono**.

```
SELECT *
FROM alumno
WHERE teléfono IS NULL;
```

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
6	Irene	Gutiérrez	Sánchez	1991-03-28	sí	NULL
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL

Ejemplo:

Obtener la lista de alumnos que no tienen un valor **NULL** en la columna **teléfono**.

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
2	Juan	Sáez	Vega	1998-04-02	no	618253876
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
5	Paco	Martínez	López	1995-11-24	no	692735409
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
8	Antonio	Carretero	Ortega	1994-05-20	sí	612345633

```
SELECT *
FROM alumno
WHERE teléfono IS NOT NULL;
```



[2.7] Funciones

A continuación se muestran una referencia de las funciones disponibles en cada SGBD que pueden ser utilizadas para realizar consultas.

Las funciones se pueden utilizar en las cláusulas **SELECT**, **WHERE** y **ORDER BY**.



[2.7.1] Funciones en MySQL

En la documentación oficial puede encontrar la [referencia completa de todas las funciones disponibles en MySQL](#). En este apartado voy a incorporar unas tablas resumen de la mayoría de ellas.

Créditos® : Para la realización de este apartado se han utilizado tablas resumen de funciones, publicadas en [w3schools.com](#)

[2.7.1.1] Funciones con cadenas MySQL

Función	Descripción
ASCII	Returns the ASCII value for the specific character
CHAR_LENGTH	Returns the length of a string (in characters)
CHARACTER_LENGTH	Returns the length of a string (in characters)
CONCAT	Adds two or more expressions together
CONCAT_WS	Adds two or more expressions together with a separator
FIELD	Returns the index position of a value in a list of values
FIND_IN_SET	Returns the position of a string within a list of strings
FORMAT	Formats a number to a format like “#,###,###.##”, rounded to a specified number of decimal places
INSERT	Inserts a string within a string at the specified position and for a certain number of characters

Función	Descripción
INSTR	Returns the position of the first occurrence of a string in another string
LCASE	Converts a string to lower-case
LEFT	Extracts a number of characters from a string (starting from left)
LENGTH	Returns the length of a string (in bytes)
LOCATE	Returns the position of the first occurrence of a substring in a string
LOWER	Converts a string to lower-case
LPAD	Left-pads a string with another string, to a certain length
LTRIM	Removes leading spaces from a string
MID	Extracts a substring from a string (starting at any position)
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string as many times as specified
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RPAD	Right-pads a string with another string, to a certain length
RTRIM	Removes trailing spaces from a string
SPACE	Returns a string of the specified number of space characters
STRCMP	Compares two strings
SUBSTR	Extracts a substring from a string (starting at any position)
SUBSTRING	Extracts a substring from a string (starting at any position)
SUBSTRING_INDEX	Returns a substring of a string before a specified number of delimiter occurs
TRIM	Removes leading and trailing spaces from a string
UCASE	Converts a string to upper-case
UPPER	Converts a string to upper-case



[2.7.1.2] Funciones numéricas MySQL

Ejemplos:

ABS devuelve el valor absoluto de un número.

```
SELECT ABS(-25);
```

25

POW(x, y) devuelve el valor de x elevado a y.

```
SELECT POW(2, 10);
```

1024

SQRT devuelve la raíz cuadrada de un número.

```
SELECT SQRT(1024);
```

32

PI() devuelve el valor del número PI.

```
SELECT PI();
```

3.141593

ROUND redondea un valor numérico

```
SELECT ROUND(37.6234);
```

38

TRUNCATE Trunca un valor numérico.

```
SELECT TRUNCATE(37.6234, 0);
```

37

CEIL devuelve el **entero inmediatamente superior o igual** al parámetro de entrada.

```
SELECT CEIL(4.3);
```

5

FLOOR devuelve el **entero inmediatamente inferior o igual** al parámetro de entrada.

```
SELECT FLOOR(4.3);
```

4

Función	Descripción
ABS	Returns the absolute value of a number
ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of one or two numbers
ATAN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEIL	Returns the smallest integer value that is \geq to a number
CEILING	Returns the smallest integer value that is \geq to a number
COS	Returns the cosine of a number
COT	Returns the cotangent of a number
COUNT	Returns the number of records returned by a select query
DEGREES	Converts a value in radians to degrees
DIV	Used for integer division
EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is \leq to a number
GREATEST	Returns the greatest value of the list of arguments
LEAST	Returns the smallest value of the list of arguments
LN	Returns the natural logarithm of a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
LOG2	Returns the natural logarithm of a number to base 2
MAX	Returns the maximum value in a set of values
MIN	Returns the minimum value in a set of values

Función	Descripción
MOD	Returns the remainder of a number divided by another number
PI	Returns the value of PI
POW	Returns the value of a number raised to the power of another number
POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number
SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number
TRUNCATE	Truncates a number to the specified number of decimal places



[2.7.1.3] Funciones con fechas MySQL

⚠️ Atención!: Tenga en cuenta que para que los nombres de los meses y las abreviaciones aparezcan en español deberá configurar la variable global `lc_time_names`. Esta variable afecta al resultado de las funciones `DATE_FORMAT`, `DAYNAME` y `MONTHNAME`.

Para configurar la variable global a nuestro idioma tendrá que hacer lo siguiente:

```
SET lc_time_names = 'es_ES';
```

Una vez hecho esto podrá consultar su valor haciendo:

```
SELECT @@lc_time_names;
```

⚠️ Atención!: Tenga en cuenta que también será necesario configurar nuestra zona horaria para que las funciones relacionadas con la hora devuelvan los valores de nuestra zona horaria. Las variables de MySQL que hay que configurar son las siguientes:

```
SET GLOBAL time_zone = 'Europe/Madrid';
SET time_zone = 'Europe/Madrid';
```

Podemos comprobar el estado de las variables haciendo:

```
SELECT @@GLOBAL.time_zone, @@SESSION.time_zone;
```

Ejemplos:

`NOW()` devuelve la fecha y la hora actual.

```
SELECT NOW();
```

`CURTIME()` devuelve la hora actual.

```
SELECT CURTIME();
```

Función	Descripción
ADDDATE	Adds a time/date interval to a date and then returns the date
ADDTIME	Adds a time interval to a time/datetime and then returns the time/datetime
CURDATE	Returns the current date
CURRENT_DATE	Returns the current date

Función	Descripción
<u>CURRENT_TIME</u>	Returns the current time
<u>CURRENT_TIMESTAMP</u>	Returns the current date and time
<u>CURTIME</u>	Returns the current time
<u>DATE</u>	Extracts the date part from a datetime expression
<u>DATEDIFF</u>	Returns the number of days between two date values
<u>DATE_ADD</u>	Adds a time/date interval to a date and then returns the date
<u>DATE_FORMAT</u>	Formats a date
<u>DATE_SUB</u>	Subtracts a time/date interval from a date and then returns the date
<u>DAY</u>	Returns the day of the month for a given date
<u>DAYNAME</u>	Returns the weekday name for a given date
<u>DAYOFMONTH</u>	Returns the day of the month for a given date
<u>DAYOFWEEK</u>	Returns the weekday index for a given date
<u>DAYOFYEAR</u>	Returns the day of the year for a given date
<u>EXTRACT</u>	Extracts a part from a given date
<u>FROM_DAYS</u>	Returns a date from a numeric datevalue
<u>HOUR</u>	Returns the hour part for a given date
<u>LAST_DAY</u>	Extracts the last day of the month for a given date
<u>LOCALTIME</u>	Returns the current date and time
<u>LOCALTIMESTAMP</u>	Returns the current date and time
<u>MAKEDATE</u>	Creates and returns a date based on a year and a number of days value
<u>MAKETIME</u>	Creates and returns a time based on an hour, minute, and second value
<u>MICROSECOND</u>	Returns the microsecond part of a time/datetime
<u>MINUTE</u>	Returns the minute part of a time/datetime
<u>MONTH</u>	Returns the month part for a given date
<u>MONTHNAME</u>	Returns the name of the month for a given date
<u>NOW</u>	Returns the current date and time
<u>PERIOD_ADD</u>	Adds a specified number of months to a period
<u>PERIOD_DIFF</u>	Returns the difference between two periods
<u>QUARTER</u>	Returns the quarter of the year for a given date value
<u>SECOND</u>	Returns the seconds part of a time/datetime
<u>SEC_TO_TIME</u>	Returns a time value based on the specified seconds
<u>STR_TO_DATE</u>	Returns a date based on a string and a format
<u>SUBDATE</u>	Subtracts a time/date interval from a date and then returns the date
<u>SUBTIME</u>	Subtracts a time interval from a datetime and then returns the time/datetime
<u>SYSDATE</u>	Returns the current date and time
<u>TIME</u>	Extracts the time part from a given time/datetime
<u>TIME_FORMAT</u>	Formats a time by a specified format
<u>TIME_TO_SEC</u>	Converts a time value into seconds
<u>TIMEDIFF</u>	Returns the difference between two time/datetime expressions
<u>TIMESTAMP</u>	Returns a datetime value based on a date or datetime value
<u>TO_DAYS</u>	Returns the number of days between a date and date "0000-00-00"
<u>WEEK</u>	Returns the week number for a given date

Función	Descripción
WEEKDAY	Returns the weekday number for a given date
WEEKOFYEAR	Returns the week number for a given date
YEAR	Returns the year part for a given date
YEARWEEK	Returns the year and week number for a given date



[2.7.1.3] Otras funciones MySQL

Function	Description
BIN	Returns a binary representation of a number
BINARY	Converts a value to a binary string
CASE	Goes through conditions and return a value when the first condition is met
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list
CONNECTION_ID	Returns the unique connection ID for the current connection
CONV	Converts a number from one numeric base system to another
CONVERT	Converts a value into the specified datatype or character set
CURRENT_USER	Returns the user name and host name for the MySQL account that the server used to authenticate the current client
DATABASE	Returns the name of the current database
IF	Returns a value if a condition is TRUE, or another value if a condition is FALSE
IFNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNULL	Returns 1 or 0 depending on whether an expression is NULL
LAST_INSERT_ID	Returns the AUTO_INCREMENT id of the last row that has been inserted or updated in a table
NULLIF	Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned
SESSION_USER	Returns the current MySQL user name and host name
SYSTEM_USER	Returns the current MySQL user name and host name
USER	Returns the current MySQL user name and host name
VERSION	Returns the current version of the MySQL database



[2.7.2] Funciones en SQL Server

En la documentación oficial puede encontrar la [referencia completa de todas las funciones disponibles en SQL Server](#). En este apartado voy a incorporar unas tablas resumen de la mayoría de ellas.

Créditos® : para la realización de este apartado se han utilizado tablas resumen de funciones, publicadas en [w3schools.com](#).

[2.7.2.1] Funciones con cadenas SQL Server

Función	Descripción
ASCII	Returns the ASCII value for the specific character
CHAR	Returns the character based on the ASCII code
CHARINDEX	Returns the position of a substring in a string
CONCAT	Adds two or more strings together
Concatenar con +	Adds two or more strings together

Función	Descripción
CONCAT_WS	Adds two or more strings together with a separator
DATALENGTH	Returns the number of bytes used to represent an expression
DIFFERENCE	Compares two SOUNDEX values, and returns an integer value
FORMAT	Formats a value with the specified format
LEFT	Extracts a number of characters from a string (starting from left)
LEN	Returns the length of a string
LOWER	Converts a string to lower-case
LTRIM	Removes leading spaces from a string
NCHAR	Returns the Unicode character based on the number code
PATINDEX	Returns the position of a pattern in a string
QUOTENAME	Returns a Unicode string with delimiters added to make the string a valid SQL Server delimited identifier
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REPLICATE	Repeats a string a specified number of times
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RTRIM	Removes trailing spaces from a string
SOUNDEX	Returns a four-character code to evaluate the similarity of two strings
SPACE	Returns a string of the specified number of space characters
STR	Returns a number as string
STUFF	Deletes a part of a string and then inserts another part into the string, starting at a specified position
SUBSTRING	Extracts some characters from a string
TRANSLATE	Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified in the third argument.
TRIM	Removes leading and trailing spaces (or other specified characters) from a string
UNICODE	Returns the Unicode value for the first character of the input expression
UPPER	Converts a string to upper-case



[2.7.2.2] Funciones numéricas SQL Server

Función	Descripción
ABS	Returns the absolute value of a number
ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of a number
ATN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEILING	Returns the smallest integer value that is \geq a number
COUNT	Returns the number of records returned by a select query
COS	Returns the cosine of a number
COT	Returns the cotangent of a number
DEGREES	Converts a value in radians to degrees

Función	Descripción
EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is <= to a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
MAX	Returns the maximum value in a set of values
MIN	Returns the minimum value in a set of values
PI	Returns the value of PI
POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number
SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SQUARE	Returns the square of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number



[2.7.2.3] Funciones con fechas SQL Server

Función	Descripción
CURRENT_TIMESTAMP	Returns the current date and time
DATEADD	Adds a time/date interval to a date and then returns the date
DATEDIFF	Returns the difference between two dates
DATEFROMPARTS	Returns a date from the specified parts (year, month, and day values)
DATENAME	Returns a specified part of a date (as string)
DATEPART	Returns a specified part of a date (as integer)
DAY	Returns the day of the month for a specified date
GETDATE	Returns the current database system date and time
GETUTCDATE	Returns the current database system UTC date and time
ISDATE	Checks an expression and returns 1 if it is a valid date, otherwise 0
MONTH	Returns the month part for a specified date (a number from 1 to 12)
SYSDATETIME	Returns the date and time of the SQL Server
YEAR	Returns the year part for a specified date



[2.7.2.3] Otras funciones SQL Server

Función	Descripción
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list
CONVERT	Converts a value (of any type) into a specified datatype
CURRENT_USER	Returns the name of the current user in the SQL Server database

Función	Descripción
IIF	Returns a value if a condition is TRUE, or another value if a condition is FALSE
ISNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNUMERIC	Tests whether an expression is numeric
NULLIF	Returns NULL if two expressions are equal
SESSION_USER	Returns the name of the current user in the SQL Server database
SESSIONPROPERTY	Returns the session settings for a specified option
SYSTEM_USER	Returns the login name for the current user
USER_NAME	Returns the database user name based on the specified id



[2.7.3] Ejercicios con Funciones

Ejercicios FUNCIONES CON CADENAS ?:

Realice las siguientes consultas teniendo en cuenta la **base de datos instituto**, instalada en un SGBD MySQL o MariaDB

- Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre de los alumnos y en la segunda, el nombre con todos los caracteres invertidos.
- Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos y en la segunda, el nombre y los apellidos con todos los caracteres invertidos.
- Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos en mayúscula y en la segunda, el nombre y los apellidos con todos los caracteres invertidos en minúscula.
- Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos y en la segunda, el número de caracteres que tiene en total el nombre y los apellidos.
- Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los dos apellidos de los alumnos. En la segunda columna se mostrará una dirección de correo electrónico que vamos a calcular para cada alumno. La dirección de correo estará formada por el nombre y el primer apellido, separados por el carácter . y seguidos por el dominio **@iesramonarcas.es**. Tenga en cuenta que la dirección de correo electrónico debe estar en minúscula. Utilice un alias apropiado para cada columna.
- Devuelve un listado con tres columnas, donde aparezca en la primera columna el nombre y los dos apellidos de los alumnos. En la segunda columna se mostrará una dirección de correo electrónico que vamos a calcular para cada alumno. La dirección de correo estará formada por el nombre y el primer apellido, separados por el carácter . y seguidos por el dominio **@iesramonarcas.es**. Tenga en cuenta que la dirección de correo electrónico debe estar en minúscula. La tercera columna será una contraseña que vamos a generar formada por los caracteres invertidos del segundo apellido, seguidos de los cuatro caracteres del año de la fecha de nacimiento. Utilice un alias apropiado para cada columna.
- Realiza los ejercicios anteriores utilizando funciones de **SQL Server**



Ejercicios FUNCIONES FECHA Y HORA ?:

Realice las siguientes consultas teniendo en cuenta la **base de datos instituto**, instalada en un SGBD MySQL o MariaDB

- Devuelva un listado con cuatro columnas, donde aparezca en la primera columna la fecha de nacimiento completa de los alumnos, en la segunda columna el día, en la tercera el mes y en la cuarta el año. Utilice las funciones **DAY**, **MONTH** y **YEAR**.

2. Devuelva un listado con tres columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos, en la segunda el nombre del día de la semana de la fecha de nacimiento y en la tercera el nombre del mes de la fecha de nacimiento.

- Resuelva la consulta utilizando las funciones **DAYNAME** y **MONTHNAME**.
- Resuelva la consulta utilizando la función **DATE_FORMAT**.

3. Devuelva un listado con dos columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos y en la segunda columna el número de días que han pasado desde la fecha actual hasta la fecha de nacimiento. Utilice las funciones **DATEDIFF** y **NOW**. [Consulte la documentación oficial de MySQL para DATEDIFF](#).

4. Devuelva un listado con dos columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos y en la segunda columna la edad de cada alumno/a. Intenta calcular la edad exacta, en caso contrario, la edad *aproximada* la podemos calcular realizando las siguientes operaciones:

- Calcula el número de días que han pasado desde la fecha actual hasta la fecha de nacimiento. Utilice las funciones **DATEDIFF** y **NOW**.
- Divide entre 365.25 el resultado que ha obtenido en el paso anterior. (El 0.25 es para compensar los años bisiestos que han podido existir entre la fecha de nacimiento y la fecha actual).
- Trunca las cifras decimales del número obtenido.

5. Realiza los ejercicios anteriores utilizando funciones de **SQL Server**



[3] Errores comunes

[3.1] Error al comprobar si una columna es NULL

Ejemplo: Obtener la lista de alumnos que tienen un valor **NULL** en la columna **teléfono**.

👎 Consulta incorrecta.

```
SELECT *
FROM alumno
WHERE teléfono = NULL;
```

👍 Consulta correcta.

```
SELECT *
FROM alumno
WHERE teléfono IS NULL;
```



[3.2] Error al comparar cadenas con patrones utilizando el operador =

Ejemplo: Devuelve un listado de los alumnos cuyo primer apellido empieza por **S**.

👎 Consulta incorrecta.

```
SELECT *
FROM alumno
WHERE apellido1 = 'S%';
```

👍 Consulta correcta.

```
SELECT *
FROM alumno
WHERE apellido1 LIKE 'S%';
```



[3.3] Error al comparar un rango de valores con AND

Cuando queremos comparar si un valor está dentro de un rango tenemos que utilizar dos condiciones unidas con la operación lógica **AND**. En el siguiente ejemplo se muestra una consulta incorrecta donde la segunda condición siempre será verdadera porque no estamos comparando con ningún valor, estamos poniendo un valor constante que al ser distinto de 0 siempre nos dará un valor verdadero como resultado de la comparación.

Consulta incorrecta

```
SELECT *
FROM alumno
WHERE fecha_nacimiento >= '1999/01/01' AND '1999/12/31'
```

Consulta correcta

```
SELECT *
FROM alumno
WHERE fecha_nacimiento >= '1999/01/01' AND fecha_nacimiento <= '1999/12/31'
```



[4] Créditos

Algunas de las imágenes utilizadas en este documento han sido extraídas de las siguientes fuentes:

- **SQL: The Complete Reference** de James R. Groff y otros.



[5] Referencias

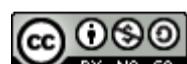
- [Wikibook SQL Exercises](#).
- [Tutorial SQL de w3resource](#).
- [MySQL Join Types by Steve Stedman](#).
- **Bases de Datos.** 2ª Edición. Grupo editorial Garceta. Iván López Montalbán, Manuel de Castro Vázquez y John Ospino Rivas.
- Referencia a [funciones MySQL](#) y [SQL Server](#) de [w3schools.com](#)
- [Script para generar la BD instituto en MySQL/MariaDB](#)
- [Script para generar la BD instituto en SQL Server](#)
- [Script para generar la BD empresa en MySQL/MariaDB](#)
- [Script para generar la BD empresa en SQL Server](#)



[6] Licencia

Elaborado por **Diego Heredia Sánchez**.

Basado en una obra de [José Juan Sánchez Hernández](#)



El contenido de esta web está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).

3. Consultas sobre varias tablas.

Realizado por Diego Heredia Sánchez

Basado en una obra de [José Juan Sánchez Hernández](#)

- [1] Consultas sobre varias tablas. Composiciones
 - [1.1] Consultas multitable SQL_1
 - [1.1.1] Composiciones cruzadas (Producto cartesiano)
 - [1.1.2] Composiciones internas (Intersección)
 - Ejercicios COMPOSICIONES SQL-1?
 - [1.2] Consultas multitable SQL_2
 - [1.2.1] Composiciones cruzadas
 - [1.2.2] Composiciones internas
 - [1.2.3] Composiciones externas
 - Ejercicios COMPOSICIONES SQL-2?
 - [1.3] Operadores de conjunto
 - [1.3.1] Operador UNION
 - [1.3.2] Operador INTERSECT
 - [1.3.3] Operador EXCEPT
 - [1.3.4] Uso de operadores de conjunto
 - Ejercicios de Operadores de Conjunto
 - [1.4] El orden en las tablas no afecta al resultado final
 - [1.5] Podemos usar alias en las tablas
 - [1.6] Unir tres o más tablas
 - [1.6.1] Combinaciones de más de dos tablas
 - [1.6.2] Operaciones de conjuntos de más de dos tablas
 - [1.7] Utilizar la misma tabla varias veces
 - [1.8] Unir una tabla consigo misma (*self-equi-join*)
 - [1.9] Uniones equivalentes (*equi-joins*) y Uniones no equivalentes (*non-equijoins*)
- [2] Errores comunes
- [3] Referencias
- [4] Licencia

[1] Consultas sobre varias tablas. Composiciones

Las consultas multitable nos permiten consultar información en más de una tabla. La única diferencia respecto a las consultas sencillas es que vamos a tener que especificar en la cláusula **FROM** cuáles son las tablas que vamos a usar y cómo las vamos a relacionar entre sí.

Para realizar este tipo de consultas podemos usar dos alternativas, la sintaxis de **SQL 1** (SQL-86), que consiste en realizar el producto cartesiano de las tablas y añadir un filtro para relacionar los datos que tienen en común, y la sintaxis de **SQL 2** (SQL-92 y SQL-2003) que incluye todas las cláusulas de tipo **JOIN**.

Atención!: En estos apuntes nos vamos a centrar en la Sintaxis utilizada por **MySQL/MariaDB** y **SQL Server**, por lo que sólo indicaremos explícitamente aquellos casos en los que haya diferencias.



[1.1] Consultas multitable SQL 1

[1.1.1] Composiciones cruzadas (Producto cartesiano)

El **producto cartesiano** de dos conjuntos, es una operación que consiste en obtener otro conjunto cuyos elementos son **todas las parejas que pueden formarse entre los dos conjuntos**. Por ejemplo, tendríamos que coger el primer elemento del primer conjunto y formar una pareja con cada uno de los elementos del segundo conjunto. Una vez hecho esto, repetimos el mismo proceso para cada uno de los elementos del primer conjunto.

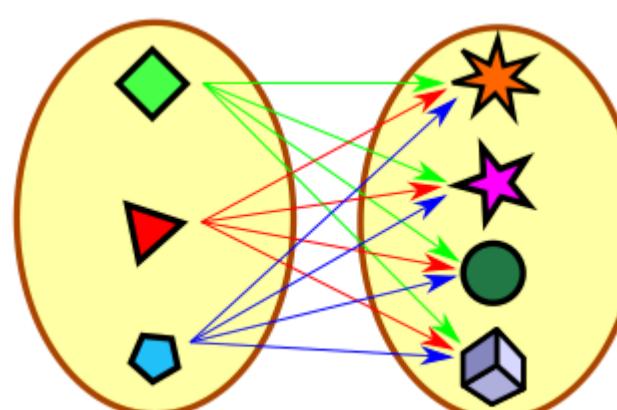


Imagen: Imagen extraída de [Wikipedia](#). Autor: [GermanX](#)

Ejemplo

Suponemos que tenemos una base de datos con dos tablas: **empleado** y **departamento**.

```
SELECT *
FROM empleado;
```

codigo	nombre	id_departamento
1	Pepe	1
2	Maria	2
3	Juan	NULL

```
SELECT *
FROM departamento;
```

id	nombre	presupuesto
1	Desarrollo	120000,00
2	Sistemas	150000,00
3	Recursos Humanos	280000,00

El **producto cartesiano** de las dos tablas se realiza con la siguiente consulta:

```
SELECT *
FROM empleado, departamento;
```

El resultado sería el siguiente:

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	1	Desarrollo	120000,00
3	Juan	NULL	1	Desarrollo	120000,00
1	Pepe	1	2	Sistemas	150000,00
2	Maria	2	2	Sistemas	150000,00
3	Juan	NULL	2	Sistemas	150000,00
1	Pepe	1	3	Recursos Humanos	280000,00
2	Maria	2	3	Recursos Humanos	280000,00
3	Juan	NULL	3	Recursos Humanos	280000,00

 **Nota:** Podéis observar como aquí se relacionan todos los elementos de la primera tabla con todos los de la segunda. Puede que esto no nos interese y que lo que realmente queramos sea filtrar solamente los elementos relacionados y descartar el resto. Esto sería una *Composición interna o intersección* en la que utilizaremos la cláusula **where**.



[1.1.2] Composiciones internas (Intersección)

La **intersección de dos conjuntos** es una operación que resulta en otro conjunto que contiene **sólo los elementos comunes** que existen en ambos conjuntos.

$$A = \{\text{orange pentagon}, \text{blue diamond}, \text{green square}, \text{yellow rectangle}\}$$

$$B = \{\text{red star}, \text{green square}, \text{green triangle}, \text{orange pentagon}\}$$

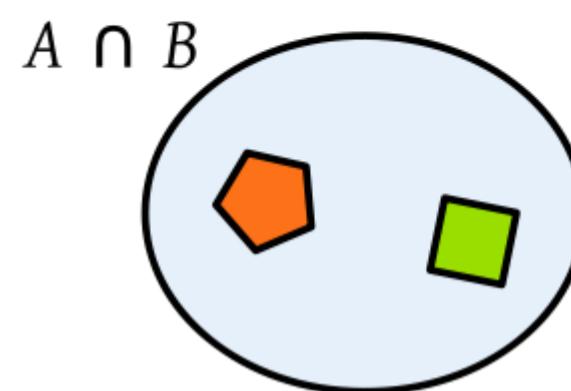


Imagen: Imagen extraída de [Wikipedia](#). Autor: Kismalac.

Ejemplo

Para poder realizar una **operación de intersección** entre las dos tablas debemos utilizar la cláusula **WHERE** para indicar la columna con la que queremos relacionar las dos tablas. Por ejemplo, para obtener un listado de los empleados y el departamento donde trabaja cada uno podemos realizar la siguiente consulta:

```
SELECT *
FROM empleado, departamento
WHERE empleado.id_departamento = departamento.id
```

El resultado sería el siguiente:

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00

Nota: Tenga en cuenta que con la **operación de intersección** sólo obtendremos los elementos de existan en ambos conjuntos. Por lo tanto, en el ejemplo anterior puede ser que existan filas en la tabla **empleado** que no aparecen en el resultado porque no tienen ningún departamento asociado, al igual que pueden existir filas en la tabla **departamento** que no aparecen en el resultado porque no tienen ningún empleado asociado.

INNER JOIN

1

```
/* SQL 1 */
SELECT *
FROM empleado, departamento
WHERE empleado.id_departamento = departamento.id
```

```
/* SQL 2 */
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id
```

2

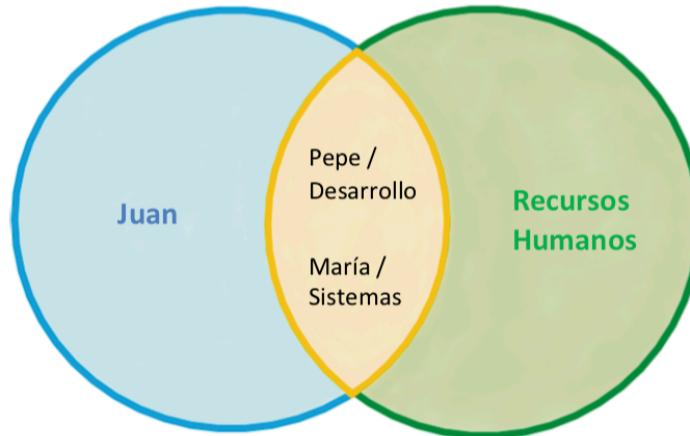
Tabla: empleado

id	nombre	id_departamento
1	Pepe	1
2	Maria	2
3	Juan	NULL

Tabla: departamento

id	nombre
1	Desarrollo
2	Sistemas
3	Recursos Humanos

Estas filas quedan fueras de la intersección



3

El resultado de la operación INNER JOIN es:

empleado.id	empleado.nombre	empleado.id_departamento	departamento.id	departamento.nombre
1	Pepe	1	1	Desarrollo
2	Maria	2	2	Sistemas



<http://josejuansanchez.org/bd>



Ejercicios COMPOSICIONES SQL-1 :

Dada la base de datos `bd_teoria_productos` que te puedes descargar desde el apartado [Referencias](#) se pide, utilizando sintaxis `SQL-1`:

- Realiza la **composición cruzada** (*producto cartesiano*) entre la tabla `fabricante` y la tabla `producto` descartando aquellas uniones en las que el **código** del fabricante sea **par**
- Realiza la **composición interna** (*intersección*) entre la tabla `fabricante` y la tabla `producto`



[1.2] Consultas multitabla SQL 2

[1.2.1] Composiciones cruzadas

- Producto cartesiano
 - **CROSS JOIN**

Ejemplo de **CROSS JOIN**:

```
SELECT *
FROM empleado CROSS JOIN departamento
```

Esta consulta nos devolvería el producto cartesiano de las dos tablas.



[1.2.2] Composiciones internas

- Join interna
 - **INNER JOIN** o **JOIN**
 - **NATURAL JOIN** (No implementada en *SQL Server*)

Ejemplo de **INNER JOIN**:

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id
```

Esta consulta nos devolvería la intersección entre las dos tablas.

La palabra reservada **INNER** es opcional, de modo que la consulta anterior también se puede escribir así:

```
SELECT *
FROM empleado JOIN departamento
ON empleado.id_departamento = departamento.id
```

⚠️ Atención!: Tenga en cuenta que **si olvidamos incluir la cláusula ON obtendremos el producto cartesiano de las dos tablas**.

Por ejemplo, la siguiente consulta nos devolverá el producto cartesiano de las tablas **empleado** y **departamento**.

```
SELECT *
FROM empleado INNER JOIN departamento
```

Ejemplo de **NATURAL JOIN**:

⚠️ Atención!: **SQL Server** no dispone de este tipo de sintaxis, a diferencia de **MySQL, MariaDB u Oracle**

```
SELECT *
FROM empleado NATURAL JOIN departamento
```

Esta consulta nos devolvería la intersección de las dos tablas, pero utilizaría las columnas que tengan el mismo nombre para relacionarlas. En este caso usaría la columna **nombre**. Sólo deberíamos utilizar una composición de tipo **NATURAL JOIN** cuando estemos seguros que los nombres de las columnas sobre las que quiero relacionar las dos tablas se llaman igual en las dos tablas. Lo normal es que no suelan tener el mismo nombre y que debamos usar una composición de tipo **INNER JOIN**.

Vamos a forzar un ejemplo:

```
INSERT INTO departamento(id,nombre,presupuesto)
VALUES(4, 'Pepe', 100);
```

Supongamos que en la tabla **departamento** insertamos un nuevo departamento llamado **Pepe**. Entonces la tabla quedaría como:

id	nombre	presupuesto
1	Desarrollo	120000,00
2	Sistemas	150000,00
3	Recursos Humanos	280000,00

id	nombre	presupuesto
4	Pepe	100,00

Pues bien, tras ejecutar la sentencia siguiente en MySQL/MariaDB u Oracle:

```
SELECT *
FROM empleado NATURAL JOIN departamento
```

Obtendríamos el siguiente resultado:

nombre	codigo	id_departamento	id	presupuesto
Pepe	1	1	4	100,00

Que realmente no tiene ningun sentido 🤦, puesto que las columnas **nombre** de ambas tablas no tienen ninguna relación.



[1.2.3] Composiciones externas

- Join externa
 - LEFT OUTER JOIN o LEFT JOIN
 - RIGHT OUTER JOIN o RIGHT JOIN
 - FULL OUTER JOIN o FULL JOIN (No implementada en MySQL/MariaDB)
 - NATURAL LEFT OUTER JOIN (No implementada en SQL Server)
 - NATURAL RIGHT OUTER JOIN (No implementada en SQL Server)

Ejemplo de **LEFT OUTER JOIN**:

```
SELECT *
FROM empleado LEFT JOIN departamento
ON empleado.id_departamento = departamento.id
```

O

```
SELECT *
FROM empleado LEFT OUTER JOIN departamento
ON empleado.id_departamento = departamento.id
```

Estas consultas devolverán todas las filas de la tabla que hemos colocado a la izquierda de la composición, en este caso la tabla **empleado**. Y relacionará las filas de la tabla de la izquierda (**empleado**) con las filas de la tabla de la derecha (**departamento**) con las que encuentre una coincidencia. Si no encuentra ninguna coincidencia, se mostrarán los valores de la fila de la tabla izquierda (**empleado**) y en los valores de la tabla derecha (**departamento**) donde no ha encontrado una coincidencia mostrará el valor **NULL**.

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00
3	Juan	NULL	NULL	NULL	NULL

LEFT JOIN

1

```
/* SQL 2 */
SELECT *
FROM empleado LEFT JOIN departamento
ON empleado.id_departamento = departamento.id
```

2

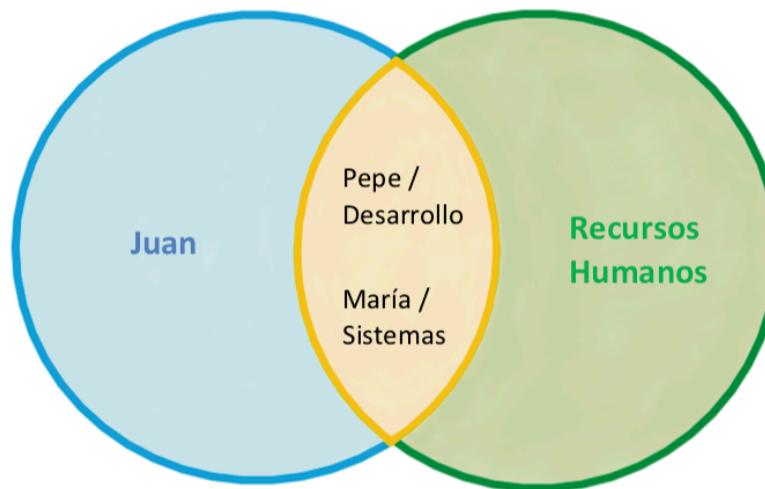
Tabla: empleado

id	nombre	id_departamento
1	Pepe	1
2	María	2
3	Juan	NULL

Tabla: departamento

id	nombre
1	Desarrollo
2	Sistemas
3	Recursos Humanos

Estas filas quedan fueras de la intersección



3

El resultado de la operación LEFT JOIN es:

empleado. id	empleado. nombre	empleado. id_departamento	departamento. id	departamento. nombre
1	Pepe	1	1	Desarrollo
2	María	2	2	Sistemas
3	Juan	NULL	NULL	NULL



<http://josejuansanchez.org/bd>

Ejemplo de **RIGHT OUTER JOIN**:

```
SELECT *
FROM empleado RIGHT JOIN departamento
ON empleado.id_departamento = departamento.id
```

O

```
SELECT *
FROM empleado RIGHT OUTER JOIN departamento
ON empleado.id_departamento = departamento.id
```

Estas consultas devolverán todas las filas de la tabla que hemos colocado a la derecha de la composición, en este caso la tabla `departamento`. Y relacionará las filas de la tabla de la derecha (`departamento`) con las filas de la tabla de la izquierda (`empleado`) con las que encuentre una coincidencia. Si no encuentra ninguna coincidencia, se mostrarán los valores de la fila de la tabla derecha (`departamento`) y en los valores de la tabla izquierda (`empleado`) donde no ha encontrado una coincidencia mostrará el valor `NULL`.

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00
NULL	NULL	NULL	3	Recursos Humanos	280000,00

RIGHT JOIN

1

```
/* SQL 2 */
SELECT *
FROM empleado RIGHT JOIN departamento
ON empleado.id_departamento = departamento.id
```

2

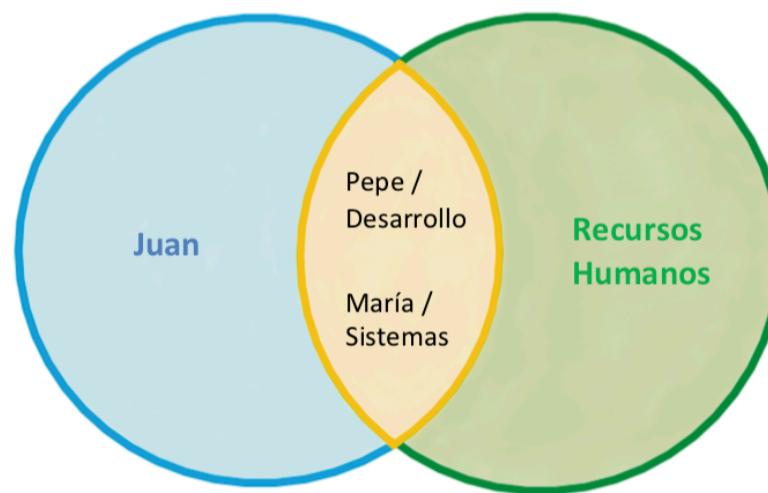
Tabla: empleado

id	nombre	id_departamento
1	Pepe	1
2	María	2
3	Juan	NULL

Tabla: departamento

id	nombre
1	Desarrollo
2	Sistemas
3	Recursos Humanos

Estas filas quedan fueras de la intersección



3

El resultado de la operación RIGHT JOIN es:

empleado.id	empleado.nombre	empleado.id_departamento	departamento.id	departamento.nombre
1	Pepe	1	1	Desarrollo
2	María	2	2	Sistemas
NULL	NULL	NULL	3	Recursos Humanos



<http://josejuansanchez.org/bd>

Ejemplo de FULL OUTER JOIN:

El resultado esperado de una composición de tipo **FULL OUTER JOIN** es obtener la intersección de las dos tablas, junto las filas de ambas tablas que no se puedan combinar. Dicho con otras palabras, el resultado sería el equivalente a realizar la unión de una consulta de tipo **LEFT JOIN** y una consultas de tipo **RIGHT JOIN** sobre las mismas tablas.

SQL Server:

```
SELECT *
FROM empleado FULL OUTER JOIN departamento
ON empleado.id_departamento = departamento.id
```

O

```
SELECT *
FROM empleado FULL JOIN departamento
ON empleado.id_departamento = departamento.id
```

Estas consultas obtendrían como resultado:

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00
3	Juan	NULL	NULL	NULL	NULL
NULL	NULL	NULL	3	Recursos Humanos	280000,00

MySQL / MariaDB :

La composición **FULL OUTER JOIN** no está implementada en MySQL, por lo tanto para poder simular esta operación será necesario hacer uso del operador **UNION**, que nos realiza la unión del resultado de dos consultas.

```
SELECT *
FROM empleado LEFT JOIN departamento
ON empleado.id_departamento = departamento.id

UNION

SELECT *
FROM empleado RIGHT JOIN departamento
ON empleado.id_departamento = departamento.id
```

Con lo cual conseguiremos un resultado idéntico al presentado anteriormente.

Ejemplo de NATURAL LEFT JOIN:

```
SELECT *
FROM empleado NATURAL LEFT JOIN departamento
```

Esta consulta realiza un **LEFT JOIN** entre las dos tablas, la única diferencia es que en este caso no es necesario utilizar la cláusula **ON** para indicar sobre qué columna vamos a relacionar las dos tablas. **En este caso las tablas se van a relacionar sobre aquellas columnas que tengan el mismo nombre**. Por lo tanto, sólo deberíamos utilizar una composición de tipo **NATURAL LEFT JOIN** cuando estemos seguros de que los nombres de las columnas sobre las que quiero relacionar las dos tablas se llaman igual en las dos tablas.

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	NULL	NULL	NULL
2	Maria	2	NULL	NULL	NULL
3	Juan	NULL	NULL	NULL	NULL

Atención!: Recordad que no están implementadas en *SQL Server*, pero siempre podríamos usar el equivalente:

```
SELECT *
FROM empleado LEFT JOIN departamento
ON empleado.nombre = departamento.nombre
```



Ejercicios COMPOSICIONES SQL-2 :

Dada la base de datos **bd_teoria_productos** que te puedes descargar desde el apartado [Referencias](#) se pide, utilizando sintaxis **SQL-2**:

1. Realiza la **composición cruzada** entre la tabla **fabricante** y la tabla **producto** descartando aquellas uniones en las que el **código** del fabricante sea **impar**

2. Realiza la **composición interna (intersección)** entre la tabla **fabricante** y la tabla **producto**, en la que se muestren solo aquellos productos cuyo precio sea mayor o igual que 120€ ordenados por **precio** ascendenteamente.

3. Deseamos obtener de todos los **producto** un listado de todos sus campos, junto al nombre del **fabricante** de los mismos. En la consulta deben aparecer los datos de los productos incluso si no tienen un fabricante asignado. Realiza la consulta usando la sintaxis **SQL -2** de más de una forma.

4. Deseamos obtener de todos los **producto** un listado de todos sus campos, junto a los nombres de los **fabricante** de los mismos. En la consulta deben aparecer también aquellos **fabricantes** que no tienen productos asociados.

5. Deseamos obtener de todos los **producto** un listado de todos sus campos, junto a los nombres de los **fabricante** de los mismos. En la consulta deben aparecer también tanto los **producto** que no tienen un fabricante asignado, como aquellos **fabricantes** que no tienen productos asociados. *Indica las instrucciones para realizarlo tanto en SQL Server como en MySQL/MariaDB.*



[1.3] Operadores de conjunto

Los operadores de conjunto **UNION**, **INTERSECT** y **EXCEPT** se utilizan para comparar y fusionar los resultados de dos expresiones de consulta diferentes.

Por ejemplo:

- Si deseamos saber qué usuarios de un sitio web son compradores y vendedores pero los nombres de usuario están almacenados en diferentes columnas o tablas, podemos buscar la intersección de estos dos tipos de usuarios.
- Si deseamos saber qué usuarios de un sitio web son compradores pero no vendedores, puede usar el operador **EXCEPT** para buscar la diferencia entre las dos listas de usuarios.
- Si deseamos crear una lista de todos los usuarios, independientemente de la función, puede usar el operador **UNION**.

Vamos a definir **query1** como una expresión de consulta que corresponde, en la forma de su lista de selección, a una segunda expresión de consulta **query2**. Entre ambas **query** debe existir el operador de conjunto deseado UNION, INTERSECT o EXCEPT.

⚠️ Atención: las dos expresiones (**query1** y **query2**) deben contener la misma cantidad de columnas de salida con tipos de datos compatibles; de lo contrario, no se podrán comparar ni fusionar los dos conjuntos de resultados. Las operaciones de conjunto no permiten conversiones implícitas entre diferentes categorías de tipos de datos.

💡 Nota: podemos crear consultas que contengan una cantidad ilimitada de expresiones de consulta y vincularlas con operadores **UNION**, **INTERSECT** y **EXCEPT** en cualquier combinación. Por ejemplo, la siguiente estructura de consulta es válida, suponiendo que las tablas **T1**, **T2** y **T3** contienen conjuntos de columnas compatibles:

```
select * from T1
UNION
select * from T2
EXCEPT
select * from T3
ORDER BY c1;
```



[1.3.1] Operador UNION

Operación de conjunto que devuelve un nuevo conjunto de resultados formado por todas las filas de la primera consulta junto a todas las de la segunda consulta. El comportamiento por defecto es que si en la unión existen elementos **duplicados**, se **descartan en el resultado final**.

Sintaxis:

```
query1
UNION [ ALL ]
query2
```

La palabra clave **ALL** indica que deseamos conservar cualquier fila duplicada que la **UNION** produce. Como hemos dicho, el comportamiento predeterminado cuando no se usa la palabra clave **ALL** es descartar todos estos duplicados.

Ejemplos

Vamos a realizar una primera consulta de los alumnos no repetidores y añadiremos un campo **estado** con el texto ‘**NO REPETIDOR**’

```
SELECT nombre, apellido1, apellido2, 'NO REPETIDOR' AS estado
FROM alumno
WHERE es_repetidor='no';
```

nombre	apellido1	apellido2	estado
María	Sánchez	Pérez	NO REPETIDOR
Juan	Sáez	Vega	NO REPETIDOR
Pepe	Ramírez	Gea	NO REPETIDOR
Paco	Martínez	López	NO REPETIDOR
Cristina	Fernández	Ramírez	NO REPETIDOR
Manuel	Domínguez	Hernández	NO REPETIDOR
Daniel	Moreno	Ruiz	NO REPETIDOR

Vamos a realizar una segunda consulta de los alumnos que tienen telefono y añadiremos un campo **estado** con el texto ‘**TIENE TELEFONO**’

```
SELECT nombre, apellido1, apellido2, 'TIENE TELEFONO' AS estado
FROM alumno
WHERE teléfono IS NOT NULL;
```

nombre	apellido1	apellido2	estado
Juan	Sáez	Vega	TIENE TELEFONO
Lucía	Sánchez	Ortega	TIENE TELEFONO
Paco	Martínez	López	TIENE TELEFONO
Cristina	Fernández	Ramírez	TIENE TELEFONO
Antonio	Carretero	Ortega	TIENE TELEFONO

Si realizamos ahora la **UNION** de ambas consultas:

```
SELECT nombre, apellido1, apellido2, 'NO REPETIDOR' AS estado
FROM alumno
WHERE es_repetidor='no'
UNION
SELECT nombre, apellido1, apellido2, 'TIENE TELEFONO' AS estado
FROM alumno
WHERE teléfono IS NOT NULL;
```

Observamos como habrá alumnos con dos filas en los resultados puesto que son *no repetidores con teléfono*.

nombre	apellido1	apellido2	estado
Antonio	Carretero	Ortega	TIENE TELEFONO
Cristina	Fernández	Ramírez	NO REPETIDOR
Cristina	Fernández	Ramírez	TIENE TELEFONO
Daniel	Moreno	Ruiz	NO REPETIDOR
Juan	Sáez	Vega	NO REPETIDOR
Juan	Sáez	Vega	TIENE TELEFONO
Lucía	Sánchez	Ortega	TIENE TELEFONO
Manuel	Domínguez	Hernández	NO REPETIDOR
María	Sánchez	Pérez	NO REPETIDOR

nombre	apellido1	apellido2	estado
Paco	Martínez	López	NO REPETIDOR
Paco	Martínez	López	TIENE TELEFONO
Pepe	Ramírez	Gea	NO REPETIDOR

¿Qué habría ocurrido si no incorporamos una columna **estado** en ambas consultas?

```
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
UNION
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL);
```

 **Nota:** Observad como las consultas se pueden encerrar entre paréntesis para establecer explícitamente el orden de ejecución, si no se hace así, habrá que tener en cuenta la precedencia de operadores, donde **INTERSECT** tiene mayor preferencia que **UNION** como [veremos más adelante](#).

nombre	apellido1	apellido2
Antonio	Carretero	Ortega
Cristina	Fernández	Ramírez
Daniel	Moreno	Ruiz
Juan	Sáez	Vega
Lucía	Sánchez	Ortega
Manuel	Domínguez	Hernández
Maria	Sánchez	Pérez
Paco	Martínez	López
Pepe	Ramírez	Gea

Fijaos como ahora, en la **UNION**, los **elementos duplicados se observa claramente como se descartan** y no aparecen nada más que una vez. Si deseamos que **NO SE DESCARTEN LOS DUPLICADOS** tenemos que utilizar el modificador **ALL**

 **Nota:** No se admiten las expresiones ~~INTERSECT ALL, EXCEPT ALL ni MINUS ALL~~

```
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
UNION ALL
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL);
```

nombre	apellido1	apellido2
Maria	Sánchez	Pérez
Juan	Sáez	Vega
Pepe	Ramírez	Gea
Paco	Martínez	López
Cristina	Fernández	Ramírez
Manuel	Domínguez	Hernández
Daniel	Moreno	Ruiz
Juan	Sáez	Vega
Lucía	Sánchez	Ortega
Paco	Martínez	López

nombre	apellido1	apellido2
Cristina	Fernández	Ramírez
Antonio	Carretero	Ortega



[1.3.2] Operador INTERSECT

Operación de conjunto que devuelve filas que provienen de dos expresiones de consulta. Las filas que no se devuelven en las dos expresiones se descartan.

Sintaxis:

```
query1
INTERSECT
query2
```

Ejemplos

Si necesitamos saber el **nombre**, **apellido1** y **apellido2** de los alumnos que **no son repetidores** y que además **tengan teléfono** podemos utilizar un operador de *intersección*.

```
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
INTERSECT
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL);
```

nombre	apellido1	apellido2
Cristina	Fernández	Ramírez
Juan	Sáez	Vega
Paco	Martínez	López

Nota: Se deja como ejercicio, realizar esta misma consulta sin utilizar operadores de conjuntos, la cual será más rápida de ejecutar.



[1.3.3] Operador EXCEPT

La operación de conjunto **EXCEPT** devuelve filas que provienen de la primera de las expresiones de consulta. Las filas deben existir en la primera tabla de resultados, pero no en la segunda.

Nota: en algunos Sistemas Gestor de Bases de Datos (SGBD) como sinónimo de **EXCEPT** se puede utilizar **MINUS**.

Sintaxis:

```
query1
EXCEPT
query2
```

Ejemplos

Si necesitamos saber el **nombre**, **apellido1** y **apellido2** de los alumnos que **no son repetidores** pero que **NO tengan teléfono** podemos utilizar un operador de *diferencia*, que elimine del primer conjunto de resultados, los contenidos en el segundo.

```
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
EXCEPT
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL);
```

nombre	apellido1	apellido2
Daniel	Moreno	Ruiz
Manuel	Domínguez	Hernández
María	Sánchez	Pérez
Pepe	Ramírez	Gea



[1.3.4] Uso de operadores de conjunto

Cuando utilizamos operadores de conjunto, debemos conocer que:

- Los **nombres de la columnas** que se devuelven en el resultado de una consulta de operación de conjunto **son los nombres (o alias) de la columnas de las tablas de la primera expresión de consulta**. Debido a que estos nombres de columnas pueden ser confusos, porque los valores de la columna provienen de tablas de cualquier lado del operador de conjunto, se recomienda proporcionar alias significativos para el conjunto de resultados.
- Una expresión de consulta que precede a un operador de conjunto **NO PUEDE** contener una cláusula **ORDER BY**. Una cláusula **ORDER BY** produce resultados significativos ordenados solo cuando se utiliza al final de una consulta que contiene operadores de conjunto. En este caso, la cláusula **ORDER BY** se aplica a los resultados finales de todas las operaciones de conjunto.

```
(SELECT nombre as 'nombre de pila', apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
EXCEPT
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL)
ORDER BY nombre;
```

nombre de pila	apellido1	apellido2
Daniel	Moreno	Ruiz
Manuel	Domínguez	Hernández
María	Sánchez	Pérez
Pepe	Ramírez	Gea

- La consulta extrema también puede contener cláusula **LIMIT** estándar cuando estamos usando **MySQL/MariaDB**. Pero no puede usarse **LIMIT** para restringir la cantidad de filas que devuelve un resultado intermedio de una operación de conjuntos.

⚠️ Atención!: tampoco se puede utilizar la cláusula **TOP** de **SQL Server**, ni en la última de las consultas.

```
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE es_repetidor='no')
EXCEPT
(SELECT nombre, apellido1, apellido2
FROM alumno
WHERE teléfono IS NOT NULL)
ORDER BY nombre
LIMIT 1;
```

nombre	apellido1	apellido2
Daniel	Moreno	Ruiz

- Cuando las consultas del operador de conjunto devuelven *resultados decimales*, las columnas de resultado correspondientes se promueven a devolver la misma precisión y escala.
- En el caso de las operaciones de conjunto, *las dos filas se tratan como idénticas si, para cada par de columnas correspondiente, los dos valores de datos son iguales o NULL*. Por ejemplo, si las tablas T1 y T2 contienen una columna y una fila, y esa fila es NULL en ambas tablas, una operación **INTERSECT** sobre esas tablas devuelve esa fila.



Ejercicios de Operadores de Conjunto:

1. Realiza la **UNION** del **nombre**, **apellido1** y **apellido2** de los alumnos **repetidores** junto a los alumnos cuyo nombre termine en **o**.

2. Sobre el resultado de la consulta anterior elimina aquellos alumnos cuyos dos apellidos acaben en **z**.

3. Selecciona los apellidos que aparecen en el nombre de un alumno, tanto en **apellido1** como en **apellido2** y ordenalos por ese campo de manera descendente. El nuevo campo se debe llamar **apellido1y2**.



[1.4] El orden en las tablas no afecta al resultado final

Estas dos consultas **devuelven el mismo conjunto de resultados**, salvo por el hecho de que las columnas de ambas tablas aparecen en distinto orden, pero es solo a efectos de presentación, si hubieramos especificado un orden para los campos o atributos, aparecerían en la misma posición:

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00

```
SELECT *
FROM departamento INNER JOIN empleado
ON empleado.id_departamento = departamento.id
```

id	nombre	presupuesto	codigo	nombre	id_departamento
1	Desarrollo	120000,00	1	Pepe	1
2	Sistemas	150000,00	2	Maria	2



[1.5] Podemos usar alias en las tablas

Utilizando la palabra reservada **AS** o simplemente separando el nombre de la tabla de su alias por un espacio.  Esto nos va a facilitar la escritura de los nombres de los atributos cuando existan coincidencias.

```
SELECT *
FROM empleado AS e INNER JOIN departamento AS d
ON e.id_departamento = d.id
```

```
SELECT *
FROM empleado e INNER JOIN departamento d
ON e.id_departamento = d.id
```

Ambas consultas nos devolverían el mismo resultado:

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00



[1.6] Unir tres o más tablas

[1.6.1] Combinaciones de más de dos tablas

En notación **SQL-2** Los conjuntos de resultados en las combinaciones de van obteniendo de izquierda a derecha, por tanto, podemos combinar más de dos tablas **especificando tras el nombre de la tabla y la palabra reservada ON** para cada combinación el/los campo/campos que se utilizan para relacionar el conjunto de resultados anterior, con la nueva tabla.

⚠️ Atención!: En un producto cartesiano **SQL-1** tenemos que no olvidar poner todas las condiciones necesarias en la cláusula **WHERE** u obtendremos un resultado erróneo.

Ejemplo:

```
SELECT *
FROM cliente INNER JOIN empleado
ON cliente.codigo_empleado_rep_ventas = empleado.codigo_empleado
INNER JOIN pago
ON cliente.codigo_cliente = pago.codigo_cliente;
```



[1.6.2] Operaciones de conjuntos de más de dos tablas

Podemos utilizar operadores de conjuntos como **UNION** para unir tres o más tablas, en estos casos es importante conocer el **Orden de evaluación para los operadores de conjunto**.

Los operadores de conjunto **UNION** y **EXCEPT** se asocian por la izquierda. Si no se especifican paréntesis para establecer el orden de prioridad, los operadores se evalúan de izquierda a derecha. Por ejemplo, en la siguiente consulta, **UNION** de **T1** y **T2** se evalúa primero, luego se realiza la operación **EXCEPT** en el resultado de UNION:

```
select * from T1
UNION
select * from T2
EXCEPT
select * from T3
order by c1;
```

El operador **INTERSECT** prevalece sobre los operadores **UNION** y **EXCEPT** cuando se utiliza una combinación de operadores en la misma consulta. Por ejemplo, la siguiente consulta evalúa la intersección de **T2** y **T3**, y luego unirá el resultado con **T1**:

```
select * from T1
UNION
select * from T2
INTERSECT
select * from T3
order by c1;
```

Al añadir paréntesis, puede aplicar un orden diferente de evaluación. En el siguiente caso, el resultado de la unión de **T1** y **T2** está intersectado con **T3**, y la consulta probablemente produzca un resultado diferente.

```
(select * from T1
UNION
select * from T2)
INTERSECT
(select * from T3)
order by c1;
```

💡 Idea!: para no equivocarnos, es recomendable usar siempre paréntesis en operaciones de conjuntos de más de una tabla



[1.7] Utilizar la misma tabla varias veces

Si necesitamos utilizar la misma tabla varias veces, podemos usar alias para darle nombres distintos a la aparición de cada una de ellas en la consulta.

Vamos a forzar un ejemplo:

Supongamos que existiera una tabla llamada **observaciones**

num	texto
10	Devuelve las llaves
20	Trabaja este fin de semana
30	Está de baja
100	Obs.Dpto Desarrollo

num	texto
200	Nota para Sisemas
300	Falta personal

Y que tanto la tabla `empleado` como la tabla `departamento` incorporaran un campo adicional `num_observacion`.

¿Como obtendrías un listado de todos los empleados, junto a sus observaciones (si las hay), y a las observaciones del departamento al que pertenece (si lo tiene y ese departamento tiene observaciones)?

⚠️Atención!: Vas a tener que usar dos veces la tabla `observaciones` en la consulta **@@**.

```
SELECT e.nombre, o1.texto 'Obs Empleado', o2.texto 'Obs Dpto'
FROM empleado e LEFT JOIN observaciones o1
ON e.num_observacion = o1.num
LEFT JOIN departamento d
ON e.id_departamento = d.id
LEFT JOIN observaciones o2
ON d.num_observacion = o2.num
```

nombre	Obs Empleado	Obs Dpto
Pepe	Devuelve las llaves	Obs.Dpto Desarrollo
Maria	Trabaja este fin de semana	Nota para Sisemas
Juan	Está de baja	NULL



[1.8] Unir una tabla consigo misma (*self-equi-join*)

Para poder hacer una operación de `INNER JOIN` sobre la misma tabla es necesario utilizar un alias para la tabla. A continuación se muestra un ejemplo de las dos formas posibles de hacer una operación de `INNER JOIN` sobre la misma tablas haciendo uso de alias.

Ejemplo:

Para este ejemplo se supone que la tabla `empleado` incorpora un nuevo campo llamado `codigo_jefe` que va a contener el `codigo` del empleado que actúa como su jefe.

Deseamos obtener un listado de los empleados con sus jefes.

```
SELECT empleado.nombre, empleado.apellido1, empleado.apellido2, jefe.nombre, jefe.apellido1, jefe.apellido2
FROM empleado INNER JOIN empleado AS jefe
ON empleado.codigo_jefe = jefe.codigo_empleado
```

El uso de `AS` como siempre, es opcional:

```
SELECT empleado.nombre, empleado.apellido1, empleado.apellido2, jefe.nombre, jefe.apellido1, jefe.apellido2
FROM empleado INNER JOIN empleado jefe
ON empleado.codigo_jefe = jefe.codigo_empleado
```



[1.9] Uniones equivalentes (*equi-joins*) y Uniones no equivalentes (*non-equijoins*)

- Las uniones equivalentes *EQUI JOIN*:
 - Es un **JOIN simple**.
 - Utilizan el operador de comparación `=` en la condición.

Ejemplo equi-join:

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00

- Los uniones no equivalentes *NON EQUI JOIN*:
 - Son **JOIN** que **NO ULIZAN EL OPERADOR =** en la condición.
 - Se utilizan operadores como **>**, **<**, **>=**, **<=** o **like** con la condición.

Ejemplos non-equi-join:

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento >= departamento.id
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
2	Maria	2	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00

Este otro ejemplo solo funcionará hace uso del operador **like**

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento like '_';
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
1	Pepe	1	2	Sistemas	150000,00
1	Pepe	1	3	Recursos Humanos	280000,00
2	Maria	2	1	Desarrollo	120000,00
2	Maria	2	2	Sistemas	150000,00
2	Maria	2	3	Recursos Humanos	280000,00

⚠️ Atención! Fijáos como solo se combinan los empleados con **id_departamento** y lo harán con todos los departamentos cuyo código este formado por un solo dígito, por ejemplo, si existiera un departamento **10** no sería incluido en la combinación.



[2] Errores comunes

- En lugar de una combinación, utilizando el formato **SQL-1** nos olvidamos de incluir en el **WHERE** la condición que nos relaciona las dos tablas, y lo que realmente hacemos es el *producto cartesiano*.

👎 Consulta incorrecta

```
SELECT *
FROM producto, fabricante
WHERE fabricante.nombre = 'Lenovo';
```

👍 Consulta correcta

```
SELECT *
FROM producto, fabricante
WHERE producto.codigo_fabricante = fabricante.codigo AND fabricante.nombre = 'Lenovo';
```

💡 Nota: **💡** si utilizamos la sintaxis **SQL-2** y usamos **SQL-Server** no se nos puede olvidar, puesto que el **ON** es obligatorio. Sin embargo en **MySQL/MariaDB** no se percataría del error.

- Nos olvidamos de incluir **ON** en las consultas de tipo **INNER JOIN** en **MySQL/MariaDB**.

 Consulta incorrecta

```
SELECT *
FROM empleado INNER JOIN departamento
WHERE empleado.nombre = 'Pepe';
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00
1	Pepe	1	2	Sistemas	150000,00
1	Pepe	1	3	Recursos Humanos	280000,00

 Consulta correcta

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id
WHERE empleado.nombre = 'Pepe';
```

codigo	nombre	id_departamento	id	nombre	presupuesto
1	Pepe	1	1	Desarrollo	120000,00

3. Relacionamos las tablas utilizando nombres de columnas incorrectos.

 Consulta incorrecta

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.codigo = departamento.id;
```

Es más fácil cometer este error cuando los nombres de las columnas son iguales, pero tienen distinto significado:

```
SELECT *
FROM producto INNER JOIN fabricante
ON producto.codigo = fabricante.codigo;
```

 Consultas correctas

```
SELECT *
FROM empleado INNER JOIN departamento
ON empleado.id_departamento = departamento.id;
```

```
SELECT *
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo;
```

4. Cuando hacemos la intersección de tres tablas con **INNER JOIN** nos olvidamos de incluir **ON** en alguna de las intersecciones.

 Consulta incorrecta

```
SELECT DISTINCT nombre_cliente, nombre, apellido1
FROM cliente INNER JOIN empleado
INNER JOIN pago
ON cliente.codigo_cliente = pago.codigo_cliente;
```

 Consulta correcta

```
SELECT DISTINCT nombre_cliente, nombre, apellido1
FROM cliente INNER JOIN empleado
ON cliente.codigo_empleado_rep_ventas = empleado.codigo_empleado
INNER JOIN pago
ON cliente.codigo_cliente = pago.codigo_cliente;
```



[3] Referencias

- [Wikibook SQL Exercises.](#)
- [Tutorial SQL de w3resource.](#)
- [MySQL Join Types by Steve Stedman.](#)
- [Guía visual de SQL Joins.](#)
- **Bases de Datos.** 2^a Edición. Grupo editorial Garceta. Iván López Montalbán, Manuel de Castro Vázquez y John Ospino Rivas.
- [INNER JOIN.](#)
- [LEFT JOIN.](#)
- [RIGHT JOIN.](#)
- [Script para generar la BD instituto en MySQL/MariaDB](#)
- [Script para generar la BD instituto en SQL Server](#)
- [Script para generar la BD empresa en MySQL/MariaDB](#)
- [Script para generar la BD empresa en SQL Server](#)
- [Script para generar la BD productos en MySQL/MariaDB](#)
- [Script para generar la BD productos en SQL Server](#)



[4] Licencia

Elaborado por **Diego Heredia Sánchez**.

Basado en una obra de [José Juan Sánchez Hernández](#)



El contenido de esta web está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).



4. Consultas Resumen

Realizado por Diego Heredia Sánchez

Basado en una obra de [José Juan Sánchez Hernández](#)

- [1] Consultas Resumen
 - [1.1] Funciones de agregación
 - [1.1.1] Diferencia entre COUNT(*) y COUNT(columna).
 - [1.1.2] Contar valores distintos COUNT(DISTINCT columna).
 - [1.2] Agrupamiento de filas (GROUP BY)
 - [1.2.1] Modificador para crear subtotales ROLLUP
 - Ejercicios GROUP BY ?
 - [1.3] Condición de agrupamiento (HAVING)
 - [1.4] Ejemplo de agrupamiento de filas (GROUP BY) con condición de agrupamiento (HAVING)
 - Ejercicios HAVING ?
- [2] Errores comunes
 - [2.1] Error al contar el número de filas distintas
 - [2.2] Error al intentar utilizar una función de agregación en la cláusula WHERE
 - [2.3] Error al usar COUNT(*) y COUNT(columna) al hacer un LEFT JOIN
- [3] Créditos
- [4] Referencias
- [5] Licencia

[1] Consultas Resumen

Vamos a recordar la sintaxis simplificada para realizar una consulta con la sentencia **SELECT** en MySQL/MariaDB (sin **TOP**) y SQL Server (sin **LIMIT**):

```
SELECT [DISTINCT] [TOP] select_expr [, select_expr ...]
[FROM table_references]
[WHERE where_condition]
[GROUP BY [ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_condition]
[ORDER BY [ASC | DESC], ...]
[LIMIT]
```

Es muy importante conocer **en qué orden se ejecuta cada una de las cláusulas** que forman la sentencia **SELECT**. El orden de ejecución, como vimos, es el siguiente:

- Cláusula **FROM**.
- Cláusula **WHERE** (Es opcional, puede ser que no aparezca).
- Cláusula **GROUP BY** (Es opcional, puede ser que no aparezca).
- Cláusula **HAVING** (Es opcional, puede ser que no aparezca).
- Cláusula **SELECT**.
- Cláusula **ORDER BY** (Es opcional, puede ser que no aparezca).
- Cláusula **TOP** (Es opcional, puede ser que no aparezca, se usa en **SQL Server**).
- Cláusula **LIMIT** (Es opcional, puede ser que no aparezca, se usa en **MySQL/MariaDB**, para **SQL Server** existe una cláusula equivalente llamada **OFFSET**).

En esta unidad vamos a trabajar con dos nuevas cláusulas **GROUP BY** y **HAVING**.



[1.1] Funciones de agregación

Estas funciones realizan una operación específica sobre todas las filas de un grupo.

Las funciones de agregación más comunes, y válidas en la mayoría de SGBD (**MySQL/MariaDB** y **SQL Server** incluidos) son:

Función	Descripción
MAX(expr)	Valor máximo del grupo
MIN(expr)	Valor mínimo del grupo
AVG(expr)	Valor medio (promedio) del grupo
SUM(expr)	Suma de todos los valores del grupo

Función	Descripción
COUNT(*)	Número de filas que tiene el resultado de la consulta
COUNT(columna)	Número de valores no nulos que hay en esa columna

En la [documentación oficial de MySQL](#) puedes encontrar una lista completa de todas las funciones de agregación que se pueden usar, por su parte también puedes consultar las [funciones de agregación para SQL Server en la documentación de Microsoft](#).

⚠️ Atención!: las funciones de agregación sólo se pueden usar en las cláusulas **SELECT Y HAVING**.



[1.1.1] Diferencia entre COUNT(*) y COUNT(columna)

- **COUNT(*)**: Calcula el número de filas que tiene el resultado de la consulta.
- **COUNT(columna)**: Cuenta el número de valores no nulos que hay en esa columna.

⚠️ Atención!: tenga en cuenta la diferencia que existe entre las funciones **COUNT(*)** y **COUNT(columna)**, ya que devolverán resultados diferentes cuando haya valores nulos en la columna que estamos usando en la función.

Ejemplos:

Supongamos que tenemos los siguientes valores en la tabla **alumno**, dentro de nuestra base de datos **bd_teoria_instituto** disponible para su descarga desde la sección [referencias](#)

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	teléfono
1	María	Sánchez	Pérez	1990-12-01	no	NULL
2	Juan	Sáez	Vega	1998-04-02	no	618253876
3	Pepe	Ramírez	Gea	1988-01-03	no	NULL
4	Lucía	Sánchez	Ortega	1993-06-13	sí	678516294
5	Paco	Martínez	López	1995-11-24	no	692735409
6	Irene	Gutiérrez	Sánchez	1991-03-28	sí	NULL
7	Cristina	Fernández	Ramírez	1996-09-17	no	628349590
8	Antonio	Carretero	Ortega	1994-05-20	sí	612345633
9	Manuel	Domínguez	Hernández	1999-07-08	no	NULL
10	Daniel	Moreno	Ruiz	1998-02-03	no	NULL

La consulta:

```
SELECT COUNT(teléfono)
FROM alumno;
```

Devolverá:

(No column name)

5

En MySQL, las columnas calculadas aparecen con la expresión que produce el resultado de la consulta, por tanto, en lugar de **(No column name)** veremos **count(teléfono)** como nombre de dicha columna.

mientras que la consulta:

```
SELECT COUNT(*)
FROM alumno;
```

Devolverá:

(No column name)

5



[1.1.2] Contar valores distintos COUNT(DISTINCT columna)

Supongamos que tenemos los siguientes valores en la tabla `producto`:

id	nombre	tipo	precio	código_fabricante
1	Disco duro SATA3 1TB	almacenamiento	86,00	5
2	Memoria RAM DDR4 8GB	almacenamiento	120,00	4
3	Disco SSD 1 TB	almacenamiento	150,00	5
4	GeForce GTX 1050Ti	gráficos	185,00	3
5	Ati 7000Ti	gráficos	170,00	5
6	Guantes Covid-19	higiene	1,00	NULL

Y otra tabla `fabricante` que contiene los siguientes valores:

código	nombre
1	El hijo de Ep
2	Tecnología Lógica
3	Pecado Capital
4	Rey Casi Piedra
5	Puerta Al Mar

Y nos piden calcular el número de valores distintos de código de fabricante que aparecen en la tabla `producto`.

```
SELECT COUNT(DISTINCT código_fabricante)
FROM producto;
```

Esta consulta devolverá:

(No column name)

3



[1.2] Agrupamiento de filas (GROUP BY)

La cláusula `GROUP BY` nos permite crear **grupos de filas** que tienen los mismos valores en las columnas por las que se desea agrupar.

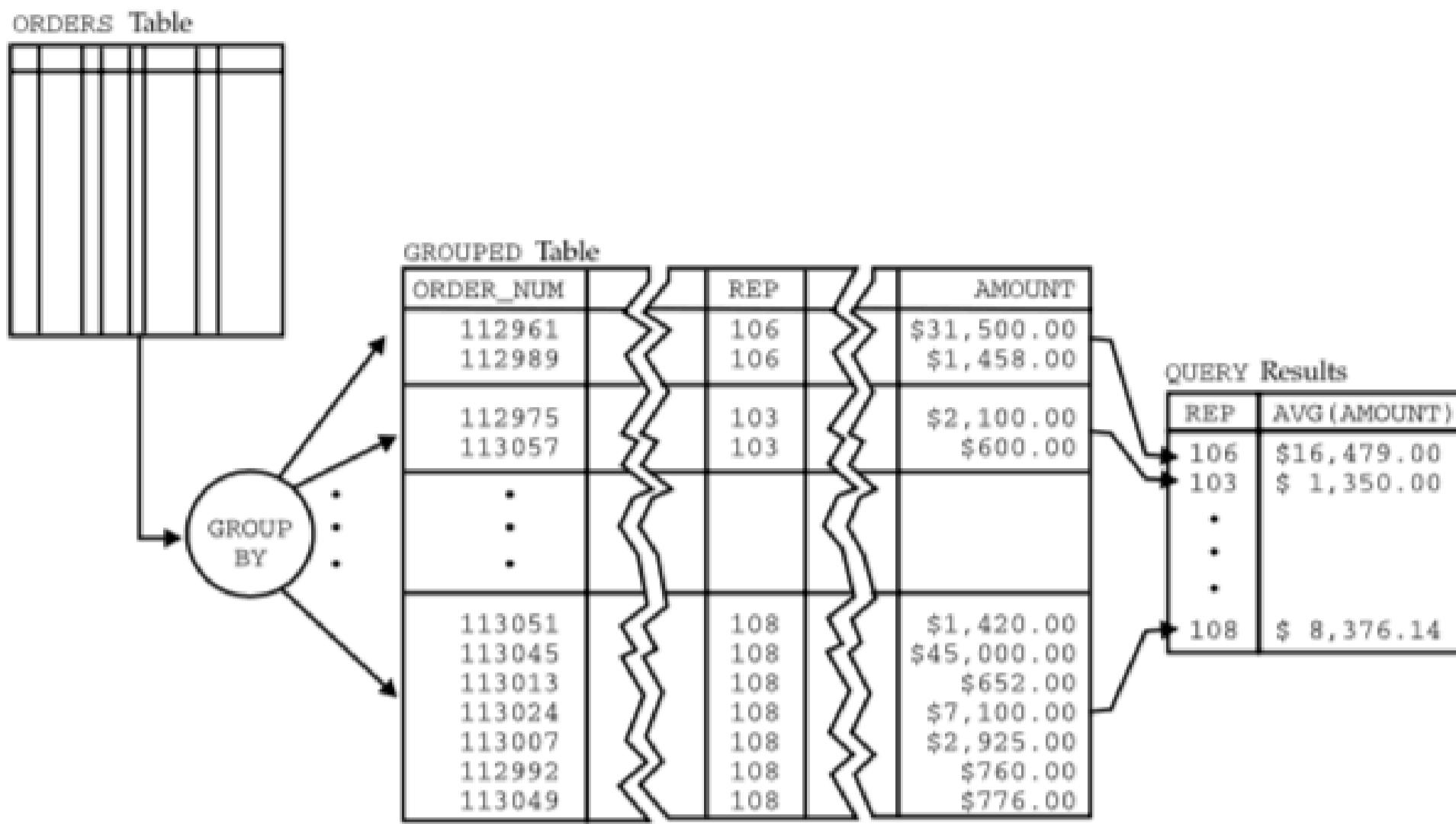


Imagen: Imagen extraída del libro SQL: The Complete Reference* de James R. Groff y otros.*

En la [documentación oficial de MySQL/MariaDB](#) puede consultar la lista de modificadores que puede utilizar con **GROUP BY**. De manera análoga, en la [documentación de Microsoft](#), podemos encontrar los modificadores de **GROUP BY** para **SQL Server**.

Ejemplo

Vamos a agrupar los productos por **código_fabricante** para contar cuantos hay de cada fabricante:

```
SELECT código_fabricante, count(*)
FROM producto
GROUP BY código_fabricante;
```

código_fabricante	(No column name)
NULL	1
3	1
4	1
5	3

Atención!: Cuando existe algún valor **null** en una columna por la que realizamos el agrupamiento, ese contará como un nuevo grupo de valores y aparecerá en los resultados.



[1.2.1] Modificador para crear subtotales **ROLLUP**

Existe un modificador que se puede emplear con **GROUP BY** llamado **ROLLUP** el cual me permite crear subtotales y un total general cuando la agrupación se realiza con más de una columna. Para ello, mueve de derecha a izquierda reduciendo el número de expresiones de columna para las que crea grupos y agregaciones.

El orden de las columnas afecta a la salida de **ROLLUP** y puede afectar al número de filas del conjunto de resultados.

Por ejemplo, **GROUP BY ROLLUP (col1, col2, col3, col4)** crea grupos para cada combinación de expresiones de columna en las listas siguientes.

- col1, col2, col3, col4
- col1, col2, col3, NULL
- col1, col2, NULL, NULL
- col1, NULL, NULL, NULL
- NULL, NULL, NULL, NULL (este es el total general)

Ejemplo

Vamos contar el número de productos agrupados por fabricante y tipo de producto y obteniendo todos los subtotales disponibles con el modificador **ROLLUP**, pero sólo de aquellos productos con fabricante

SQL Server :

Atención: Si utilizamos el modificador **ROLLUP** es importante que los campos por los que agrupemos, no contengan valores nulos, puesto que en caso de contenerlos, van a interferir en los resultados de subtotales de la consulta.

Es decir, si ejecutamos esta consulta:

```
SELECT código_fabricante, tipo, count(*) as 'cantidad'
FROM producto
GROUP BY ROLLUP (código_fabricante, tipo);
```

Vamos a obtener el siguiente resultado:

código_fabricante	tipo	cantidad
NULL	higiene	1
NULL	NULL	1
3	gráficos	1
3	NULL	1
4	almacenamiento	1
4	NULL	1
5	almacenamiento	2
5	gráficos	1
5	NULL	3
NULL	NULL	6

Dónde las dos primeras líneas se refieren al fabricante **NULL** y por tanto el total general es el que aparece en la última de las líneas y no en la segunda, que es el subtotal de productos del fabricante **NULL**

Por eso sólo vamos a obtener los grupos de aquellos productos que tengan fabricante:

```
SELECT código_fabricante, tipo, count(*) as 'cantidad'
FROM producto
WHERE código_fabricante IS NOT NULL
GROUP BY ROLLUP (código_fabricante, tipo);
```

código_fabricante	tipo	cantidad
3	gráficos	1
3	NULL	1
4	almacenamiento	1
4	NULL	1
5	almacenamiento	2
5	gráficos	1
5	NULL	3
NULL	NULL	5

Ahora se aprecian claramente las líneas con *subtotales por código de fabricante y tipo*, las líneas con **subtotales por fabricante** (*las que tienen tipo con valor NULL*) y línea con el **total general** (**NULL NULL**).



MySQL / MariaDB :

 **Nota:** La sintaxis del modificador **ROLLUP** en MySQL/MariaDB es un poco diferente, no se ponen los campos entre paréntesis y el modificador se añade al final del **GROUP BY** precedido de la palabra reservada **WITH**.

Veamos el ejemplo anterior.

```
SELECT código_fabricante, tipo, count(*) as 'cantidad'
FROM producto
GROUP BY código_fabricante, tipo WITH ROLLUP;
```

código_fabricante	tipo	cantidad
NULL	higiene	1
NULL	NULL	1
3	gráficos	1
3	NULL	1
4	almacenamiento	1
4	NULL	1
5	almacenamiento	2
5	gráficos	1
5	NULL	3
NULL	NULL	6

Vamos a eliminar los productos con código de fabricante **NULL** para que no interfieran en los resultados, igual que antes:

```
SELECT código_fabricante, tipo, count(*) as 'cantidad'
FROM producto
WHERE código_fabricante IS NOT NULL
GROUP BY código_fabricante, tipo WITH ROLLUP;
```

código_fabricante	tipo	cantidad
3	gráficos	1
3	NULL	1
4	almacenamiento	1
4	NULL	1
5	almacenamiento	2
5	gráficos	1
5	NULL	3
NULL	NULL	5

Ahora se aprecian claramente las líneas con **subtotales por código de fabricante y tipo**, las líneas con **subtotales por fabricante** (*las que tienen tipo con valor NULL*) y línea con el **total general** (**NULL NULL**), exactamente igual que en SQL Server.



Ejercicios GROUP BY?:

1. ¿Cómo podrías **mostrar el nombre del fabricante** en lugar de su código, en la consulta que hemos realizado en el apartado anterior?: *Contar el número de productos agrupados por fabricante y tipo de producto y obteniendo todos los subtotales disponibles con el modificador ROLLUP, pero sólo de aquellos productos con fabricante.*
2. Muestra el número de productos que existen ('num productos') de cada **tipo** y fabricante, incluyendo el '**nombre fabricante**' en el resultado de la consulta.
3. Deseamos obtener una consulta que nos sume el precio de todos los productos agrupados por fabricante, en los resultados de la consulta debe aparecer una primera columna llamada **fabricante** con el nombre del fabricante y la segunda debe ser la suma de los precios de sus productos, que se llamará '**total productos**'.

4. Realiza el ejercicio anterior (3), pero además de aparecer todos los fabricantes, debe aparecer una línea con el contenido **SIN FABRICANTE** que sumará el precio de todos aquellos productos que no tuvieran fabricante.



[1.3] Condición de agrupamiento (HAVING)

La cláusula **HAVING** nos permite crear **filtros** sobre los grupos de filas que tienen los mismos valores en las columnas por las que se desea agrupar.

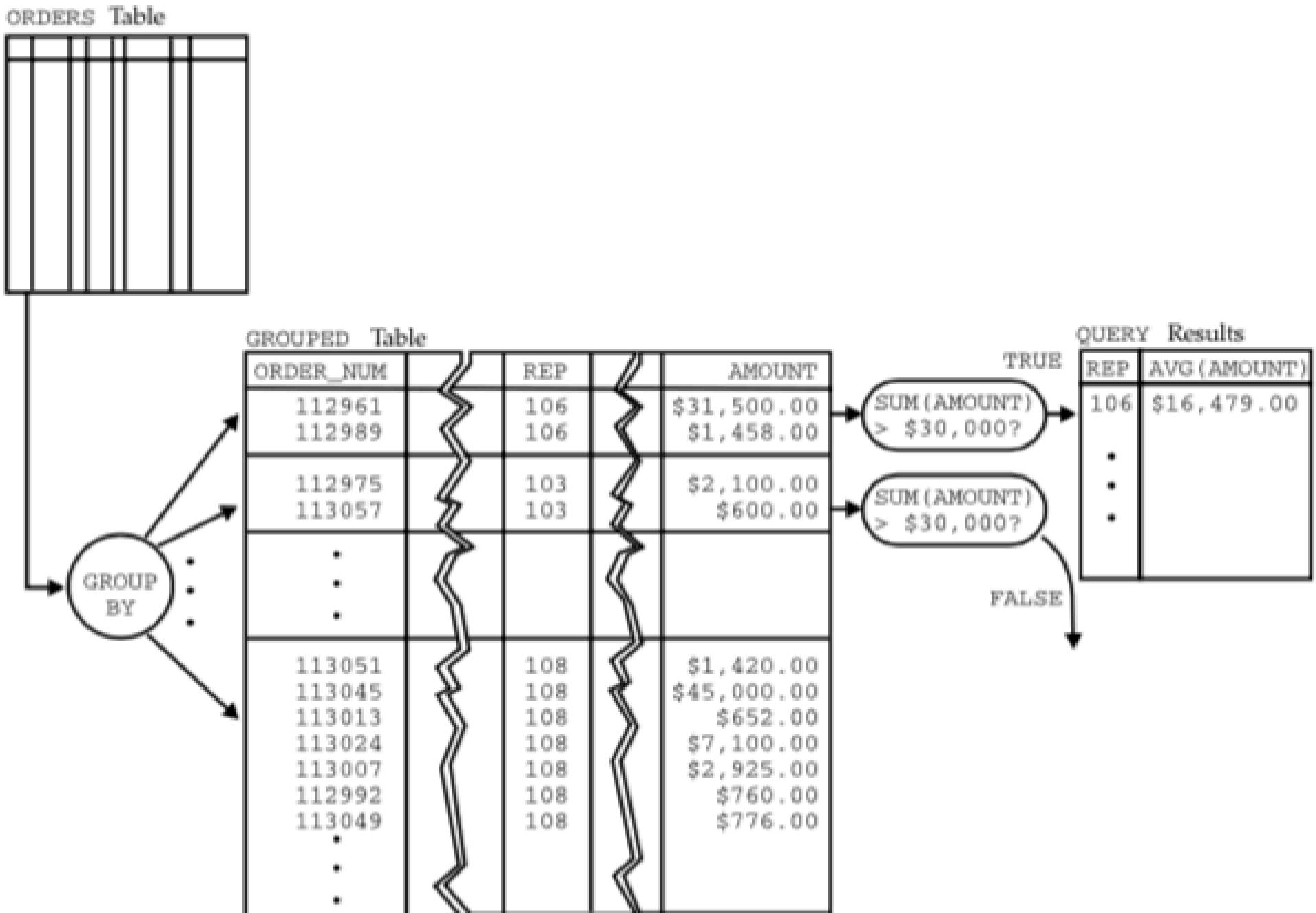


Imagen: Imagen extraída del libro *SQL: The Complete Reference* de James R. Groff y otros.



[1.4] Ejemplo de agrupamiento de filas (**GROUP BY**) con condición de agrupamiento (**HAVING**)

Consultas resumen

1

```
SELECT fabricante.nombre, AVG(producto.precio)
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo
WHERE fabricante.nombre != 'Seagate'
GROUP BY fabricante.codigo
HAVING AVG(producto.precio) >= 150
```

Tabla: producto

codigo	nombre	precio	codigo_fabricante
1	Portátil A		1
2	Monitor 24		2
3	Disco SSD 1 TB		3
4	Impresora		4
5	Monitor 27		2
6	Portátil B		1

Tabla: fabricante

codigo	nombre
1	Lenovo
2	Asus
3	Seagate
4	HP

El resultado de la operación INNER JOIN es:

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo
2	Monitor 24	203	2	2	Asus
3	Disco SSD 1 TB	115	3	3	Seagate
4	Impresora	49	4	4	HP
5	Monitor 27	242	2	2	Asus
6	Portátil B	320	1	1	Lenovo

2

```
SELECT fabricante.nombre, AVG(producto.precio)
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo
WHERE fabricante.nombre != 'Seagate'
GROUP BY fabricante.codigo
HAVING AVG(producto.precio) >= 150
```

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo
2	Monitor 24	203	2	2	Asus
3	Disco SSD 1 TB	115	3	3	Seagate
4	Impresora	49	4	4	HP
5	Monitor 27	242	2	2	Asus
6	Portátil B	320	1	1	Lenovo



El resultado después de aplicar el filtro del WHERE es:

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo
2	Monitor 24	203	2	2	Asus

	producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
4	Impresora	49	4	4	HP	
5	Monitor 27	242	2	2	Asus	
6	Portátil B	320	1	1	Lenovo	

3

```
SELECT fabricante.nombre, AVG(producto.precio)
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo
WHERE fabricante.nombre != 'Seagate'
GROUP BY fabricante.codigo
HAVING AVG(producto.precio) >= 150
```

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo
6	Portátil B	320	1	1	Lenovo
2	Monitor 24	203	2	2	Asus
5	Monitor 27	242	2	2	Asus
4	Impresora	49	4	4	HP

4

```
SELECT fabricante.nombre, AVG(producto.precio)
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo
WHERE fabricante.nombre != 'Seagate'
GROUP BY fabricante.codigo
HAVING AVG(producto.precio) >= 150
```

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo
6	Portátil B	320	1	1	Lenovo

459.5

AVG(producto.precio) >= 150 ✓

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
2	Monitor 24	203	2	2	Asus
5	Monitor 27	242	2	2	Asus

222.5

AVG(producto.precio) >= 150 ✓

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
4	Impresora	49	4	4	HP

49

AVG(producto.precio) >= 150 X

El resultado después de aplicar el filtro del HAVING es:

producto.codigo	producto.nombre	producto.precio	producto.codigo_fabricante	fabricante.codigo	fabricante.nombre
1	Portátil A	599	1	1	Lenovo

6	Portátil B	320	1	1	Lenovo
2	Monitor 24	203	2	2	Asus
5	Monitor 27	242	2	2	Asus

5

```
SELECT fabricante.nombre, AVG(producto.precio)
FROM producto INNER JOIN fabricante
ON producto.codigo_fabricante = fabricante.codigo
WHERE fabricante.nombre != 'Seagate'
GROUP BY fabricante.codigo
HAVING AVG(producto.precio) >= 150
```

fabricante.nombre	AVG(producto.precio)
Lenovo	459.5
Asus	222.5



<http://josejuansanchez.org/bd>



Ejercicios HAVING :

1. Sobre la tabla `alumno` de la base de datos `instituto` necesitamos contar los alumnos nacidos en cada año, de aquellos años donde haya nacido más de uno.
2. Y si quisieramos contar los alumnos nacidos en cada año, de aquellos años donde haya nacido más de uno, pero sin tener en cuenta los que no tengan `teléfono`.
3. Queremos obtener el número promedio del `id` de los alumnos que agrupados por la inicial de su primer apellido, tengan una **suma de sus `id` mayor que 5**, el resultado debe mostrar la inicial del campo `apellido1`, junto al número promedio de `id` que llamaremos '`media id`'.
4. Si deseáramos mostrar en el ejercicio anterior el campo `apellido1` completo, ¿cómo lo podríamos hacer?.



[2] Errores comunes

[2.1] Error al contar el número de filas distintas

Ejemplo: Cuenta el número fabricantes distintos que aparecen en la tabla `producto`.

👎 Consulta incorrecta.

```
SELECT DISTINCT COUNT(codigo_fabricante)
FROM producto;
```

👍 Consulta correcta.

```
SELECT COUNT(DISTINCT codigo_fabricante)
FROM producto;
```

[2.2] Error al intentar utilizar una función de agregación en la cláusula WHERE

Las funciones de agregación sólo se pueden usar en las cláusulas `SELECT` Y `HAVING`, por lo tanto si intenta hacer uso de una función de agregación en la cláusula `WHERE` obtendrá un error.

Ejemplo: Devuelve un listado con los productos que tienen un precio superior al precio medio de todos los artículos que existen en la tabla `productos`.

👎 Consulta incorrecta.

```
SELECT *
FROM producto
WHERE precio > AVG(precio);
```

👍 Consulta correcta.

```
SELECT *
FROM producto
WHERE precio > (SELECT AVG(precio) FROM producto);
```



[2.3] Error al usar COUNT(*) y COUNT(columna) al hacer un LEFT JOIN

Ejemplo:

Devuelve un listado con el número de productos que tiene cada fabricante. El listado debe incluir aquellos fabricantes que no tienen productos asociados indicando que tienen 0 productos.

En este caso es necesario realizar un **LEFT JOIN** con las tablas **fabricante** y **producto**. Por ejemplo, al ejecutar la siguiente consulta podemos ver que los fabricantes **El hijo de Ep (Epson)** y **Tecnología Lógica (Logitech)** no tienen productos asociados.

```
SELECT fabricante.código, fabricante.nombre, producto.id
FROM fabricante LEFT JOIN producto
ON producto.código_fabricante = fabricante.código
ORDER BY fabricante.código;
```

código	nombre	id
1	El hijo de Ep	NULL
2	Tecnología Lógica	NULL
3	Pecado Capital	4
4	Rey Casi Piedra	2
5	Puerta Al Mar	1
5	Puerta Al Mar	3
5	Puerta Al Mar	5

Para contar el número de productos que tiene cada fabricante es necesario realizar un **GROUP BY** por el código del fabricante y contar el número de filas que tiene cada uno de los grupos que hemos creado. Pero debemos tener cuidado porque obtendremos resultados diferentes al contar el número de filas de cada grupo con **COUNT(*)** y **COUNT(producto.id)**. La opción correcta es hacer uso de **COUNT(producto.id)** porque contará aquellas filas que tienen un valor distinto de **NULL** en la columna **producto.codigo**.

👎 Consulta incorrecta.

```
SELECT fabricante.código, COUNT(*) as 'num productos'
FROM fabricante LEFT JOIN producto
ON producto.código_fabricante = fabricante.código
GROUP BY fabricante.código
ORDER BY 2 DESC;
```

código	num productos
5	3
1	1
2	1
3	1
4	1

Antes de continuar, como podríamos el nombre junto al código del fabricante:

```
SELECT fabricante.código, fabricante.nombre,
       COUNT(*) as 'num productos'
  FROM fabricante LEFT JOIN producto
    ON producto.código_fabricante = fabricante.código
   GROUP BY fabricante.código, fabricante.nombre
  ORDER BY 3 DESC;
```

⚠️ *Atención!*: observa como **para poder poner el nombre del fabricante, no solo hay que ponerlo en la cláusula `SELECT` sino que debemos añadirlo al `GROUP BY` o nos dará un error.

Bien, arreglados los errores sintácticos, vemos como `El hijo de Ep` y `Tecnología Lógica` aparecen como si tuvieran `1` producto en la tabla, y sin embargo esto **no es correcto**. *Esta contando las filas con valor `NULL` como si fuesen productos reales.*

👍 Consulta correcta.

```
SELECT fabricante.código, fabricante.nombre,
       COUNT(producto.id) as 'num productos'
  FROM fabricante LEFT JOIN producto
    ON producto.código_fabricante = fabricante.código
   GROUP BY fabricante.código, fabricante.nombre
  ORDER BY 3 DESC;
```

código	nombre	num productos
5	Puerta Al Mar	3
3	Pecado Capital	1
4	Rey Casi Piedra	1
1	El hijo de Ep	0
2	Tecnología Lógica	0

Ahora si estamos contando los productos reales, ignorando aquellas filas cuyo `producto.id` era `NULL`



[3] Créditos

Algunas de las imágenes utilizadas en este documento han sido extraídas de las siguientes fuentes:

- **SQL: The Complete Reference** de James R. Groff y otros.

[4] Referencias

- [Wikibook SQL Exercises](#).
- [Tutorial SQL de w3resource](#).
- [MySQL Join Types by Steve Stedman](#).
- **Bases de Datos.** 2ª Edición. Grupo editorial Garceta. Iván López Montalbán, Manuel de Castro Vázquez y John Ospino Rivas.
- [Consultas resumen](#).
- [Script para generar la BD instituto en MySQL/MariaDB](#)
- [Script para generar la BD instituto en SQL Server](#)
- [Script para generar la BD empresa en MySQL/MariaDB](#)
- [Script para generar la BD empresa en SQL Server](#)
- [Script para generar la BD productos en MySQL/MariaDB](#)
- [Script para generar la BD productos en SQL Server](#)



[5] Licencia

Elaborado por **Diego Heredia Sánchez**.

Basado en una obra de [José Juan Sánchez Hernández](#)



El contenido de esta web está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).



5. Subconsultas

Realizado por Diego Heredia Sánchez

Basado en una obra de [José Juan Sánchez Hernández](#)

- [\[1\] Subconsultas](#)
 - [\[1.1\] Tipos de subconsultas](#)
 - [\[1.2\] Subconsultas en la cláusula WHERE](#)
 - [\[1.3\] Subconsultas en la cláusula HAVING](#)
 - [\[1.4\] Subconsultas en la cláusula FROM](#)
 - [\[1.5\] Subconsultas en la cláusula SELECT](#)
- [\[2\] Operadores que podemos usar en las subconsultas](#)
 - [\[2.1\] Operadores básicos de comparación](#)
 - [\[2.2\] Subconsultas con ALL y ANY](#)
 - [\[2.3\] Subconsultas con IN y NOT IN](#)
 - [\[2.4\] Subconsultas con EXISTS y NOT EXISTS](#)
- [\[3\] Errores comunes](#)
 - [\[3.1\] Número incorrecto de columnas en la subconsulta](#)
 - [\[3.2\] Número incorrecto de filas en la subconsulta](#)
- [\[4\] Referencias](#)
- [\[5\] Licencia](#)

[1] Subconsultas

Una subconsulta es una consulta anidada dentro de otra consulta.

Debe tener en cuenta que no existe una única solución para resolver una consulta en SQL. En esta unidad vamos a estudiar cómo podemos resolver haciendo uso de subconsultas, algunas de las consultas que hemos resuelto en las unidades anteriores.



[1.1] Tipos de subconsultas

El estándar SQL define tres tipos de subconsultas:

- **Subconsultas de fila.** Son aquellas que devuelven más de una columna pero una única fila.
- **Subconsultas de tabla.** Son aquellas que devuelven una o varias columnas y cero o varias filas.
- **Subconsultas escalares.** Son aquellas que devuelven una columna y una fila.

[1.2] Subconsultas en la cláusula WHERE

Por ejemplo, suponga que queremos conocer el nombre del producto que tiene el mayor precio. En este caso podríamos realizar una primera consulta para buscar cuál es el valor del precio máximo y otra segunda consulta para buscar el nombre del producto cuyo precio coincide con el valor del precio máximo. La consulta sería la siguiente:

```
SELECT nombre
FROM producto
WHERE precio = (SELECT MAX(precio) FROM producto)
```

En este caso sólo hay un nivel de anidamiento entre consultas pero pueden existir varios niveles de anidamiento.



[1.3] Subconsultas en la cláusula HAVING

Ejemplo: Devuelve un listado con todos los nombres de los fabricantes que tienen el mismo número de productos que el fabricante **Asus**.

```
SELECT fabricante.nombre, COUNT(*)
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
GROUP BY fabricante.nombre
HAVING COUNT(*) >= (
    SELECT COUNT(producto.codigo)
    FROM fabricante INNER JOIN producto
    ON fabricante.codigo = producto.codigo_fabricante
    WHERE fabricante.nombre = 'Asus');
```



[1.4] Subconsultas en la cláusula FROM

Ejemplo: Devuelve una lista de todos los productos que tienen un precio mayor o igual al precio medio de todos los productos de su mismo fabricante.

```
SELECT *
FROM producto INNER JOIN (
    SELECT codigo_fabricante, AVG(precio) AS media
    FROM producto
    GROUP BY codigo_fabricante) AS t
ON producto.codigo_fabricante = t.codigo_fabricante
WHERE producto.precio >= t.media;
```



[1.5] Subconsultas en la cláusula SELECT

Las subconsultas que pueden aparecer en la cláusula **SELECT** tienen que ser subconsultas de tipo **escalar**, que devuelven una única fila y columna.

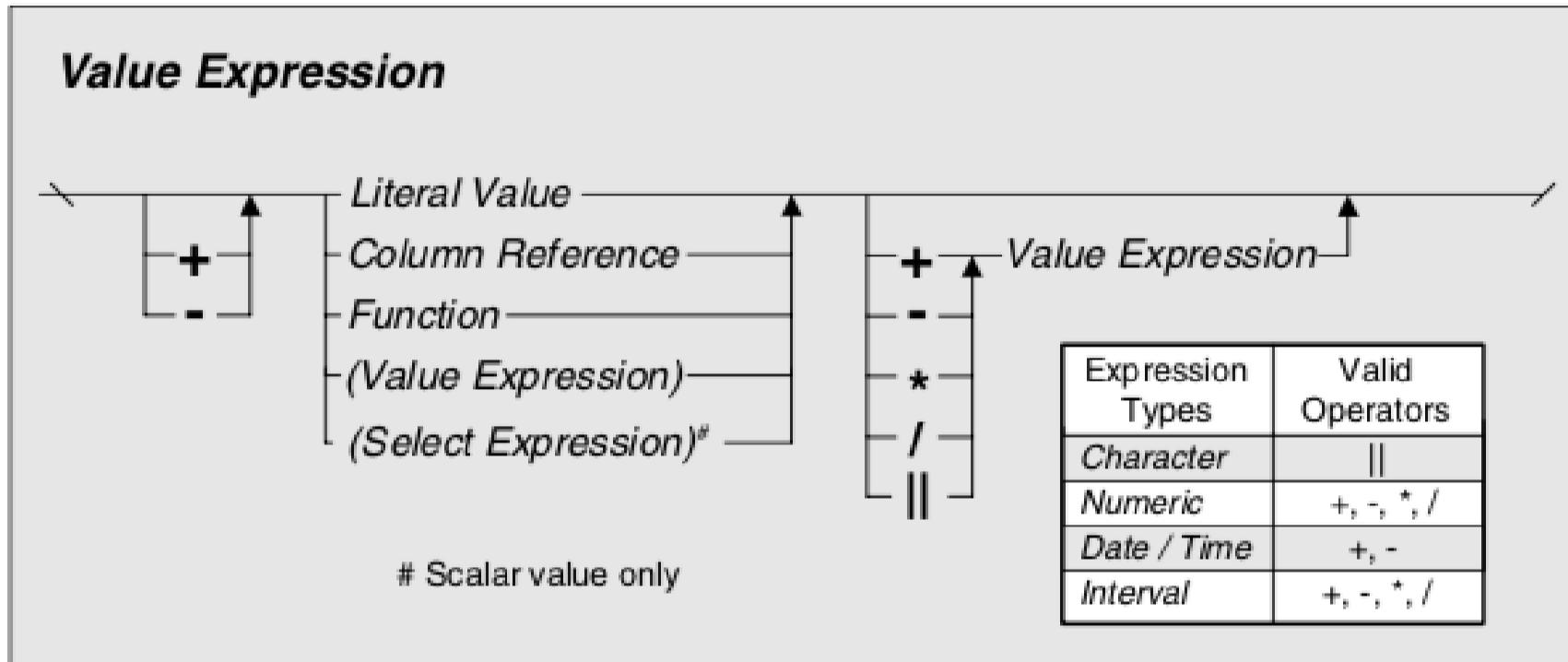


Imagen: Imagen extraída del libro «*SQL Queries for Mere Mortals*» de John L. Viescas.



[2] Operadores que podemos usar en las subconsultas

Los operadores que podemos usar en las subconsultas son los siguientes:

- Operadores básicos de comparación (>, >=, <, <=, !=, <>, =).
- Predicados **ALL** y **ANY**.
- Predicado **IN** y **NOT IN**.
- Predicado **EXISTS** y **NOT EXISTS**.



[2.1] Operadores básicos de comparación

Los operadores básicos de comparación (>, >=, <, <=, !=, <>, =) se pueden usar cuando queremos comparar una expresión con el valor que devuelve una subconsulta.

Los operadores básicos de comparación los vamos a utilizar para realizar comparaciones con **subconsultas que devuelven un único valor, es decir, una columna y una fila**.

Ejemplo: Devuelve todos los productos de la base de datos que tienen un precio mayor o igual al producto más caro del fabricante **Asus**.

```
SELECT *
FROM producto
WHERE precio >= (SELECT MAX(precio)
                  FROM fabricante INNER JOIN producto
                  ON fabricante.codigo = producto.codigo_fabricante
                  WHERE fabricante.nombre = 'Asus');
```

La consulta anterior también se puede escribir con subconsultas sin hacer uso de **INNER JOIN**.

```
SELECT *
FROM producto
WHERE precio = (
    SELECT MAX(precio)
    FROM producto
    WHERE codigo_fabricante = (
        SELECT codigo
        FROM fabricante
        WHERE nombre = 'Asus'));
```



[2.2] Subconsultas con ALL y ANY

ALL y ANY se utilizan con los operadores de comparación ($>$, \geq , $<$, \leq , \neq , \neq , $=$) y nos permiten comparar una expresión con el conjunto de valores que devuelve una subconsulta.

Sintaxis:

expresión-buscada Operador-de-comparación [ALL|ANY] (subconsulta-que-devuelve-una-lista-de-valores)

- ALL comprobará que todos los valores de la subconsulta cumplen el *operador de comparación* indicado, y en ese caso devolverá true.
- ANY o SOME comprobará si alguno de los valores de la subconsulta cumple el *operador de comparación* indicado, y en ese caso devolverá true

ALL y ANY los vamos a utilizar para realizar comparaciones con **subconsultas que pueden devolver varios valores, es decir, una columna y varias filas**.

Ejemplo: Podemos escribir la consulta que devuelve todos los productos de la base de datos que tienen un precio mayor o igual al producto más caro del fabricante Asus, haciendo uso de ALL. Por lo tanto estas dos consultas darían el mismo resultado.

```
SELECT *
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
WHERE precio >= (SELECT MAX(precio)
    FROM fabricante INNER JOIN producto
    ON fabricante.codigo = producto.codigo_fabricante
    WHERE fabricante.nombre = 'Asus');
```

```
SELECT *
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
WHERE precio >= ALL (SELECT precio
    FROM fabricante INNER JOIN producto
    ON fabricante.codigo = producto.codigo_fabricante
    WHERE fabricante.nombre = 'Asus');
```

La palabra reservada SOME es un alias de ANY. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 <> ANY (SELECT s1 FROM t2);
```

```
SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
```

Nota: Si bien utilizar estos operadores es una forma muy cómoda y clara de escribir una consulta, se tiene que considerar que **el cálculo y obtención de la subconsulta se hace cada vez que se evalúa la consulta principal**, es decir, una vez por cada registro. Este hecho, dependiendo del tamaño de las tablas involucradas, **puede penalizar el rendimiento de la consulta**.



[2.3] Subconsultas con IN y NOT IN

IN y NOT IN nos permiten comprobar si un valor está o no incluido en un conjunto de valores, que puede ser el conjunto de valores que devuelve una subconsulta.

IN y NOT IN los vamos a utilizar para realizar comparaciones con **subconsultas que pueden devolver varios valores, es decir, una columna y varias filas**.

Ejemplo: Devuelve un listado de los clientes que no han realizado ningún pedido.

```
SELECT *
FROM cliente
WHERE id NOT IN (SELECT id_cliente FROM pedido);
```

Cuando estamos trabajando con subconsultas, **IN** y **= ANY** realizan la misma función. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);

SELECT s1 FROM t1 WHERE s1 IN      (SELECT s1 FROM t2);
```

Ocurre lo mismo con **NOT IN** y **<> ALL**. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2);

SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2);
```

⚠️ Atención! Importante: Tenga en cuenta que cuando hay un valor **NULL** en el resultado de la consulta interna, la consulta externa no devuelve ningún valor.

Ejemplo: Devuelve un listado con el nombre de los departamentos que no tienen empleados asociados.

```
SELECT nombre
FROM departamento
WHERE codigo NOT IN (
    SELECT id_departamento
    FROM empleado);
```

La consulta interna **SELECT id_departamento FROM empleado**, devuelve algunas filas con valores **NULL** y por lo tanto la consulta externa no devuelve ningún valor.

La forma de solucionarlo sería quitando los valores **NULL** de la consulta interna:

```
SELECT nombre
FROM departamento
WHERE codigo NOT IN (
    SELECT id_departamento
    FROM empleado
    WHERE id_departamento IS NOT NULL);
```



[2.4] Subconsultas con **EXISTS** y **NOT EXISTS**

EXISTS especifica una subconsulta para probar la existencia de filas. Si la subconsulta devuelve registros, será **true** y **false** en caso contrario.

Ejemplo: Devuelve un listado de los clientes que no han realizado ningún pedido.

```
SELECT *
FROM cliente
WHERE NOT EXISTS (SELECT id_cliente FROM pedido WHERE cliente.id = pedido.id_cliente);
```



[3] Errores comunes

[3.1] Número incorrecto de columnas en la subconsulta

```
ERROR 1241 (ER_OPERAND_COL)
SQLSTATE = 21000
Message = "Operand should contain 1 column(s)"
```

```
SELECT (SELECT column1, column2 FROM t2) FROM t1;
```



[3.2] Número incorrecto de filas en la subconsulta

```
ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
SQLSTATE = 21000
Message = "Subquery returns more than 1 row"
```

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

La consulta anterior sólo se podrá ejecutar cuando la consulta interna `SELECT column1 FROM t2` devuelva una única fila.

 **Nota:** esto se podría solucionar añadiendo una cláusula `LIMIT 1` o `TOP 1` de forma que nos estaríamos asegurando que nunca se devuelve más de una fila en la subconsulta.



[4] Referencias

- **SQL: The Complete Reference** de James R. Groff y otros.
- **SQL Queries for Mere Mortals** de John L. Viescas.
- [Subquery Syntax](#). Documentación oficial de MySQL.
- [Optimizing Subqueries](#). Documentación oficial de MySQL.



[5] Licencia

Elaborado por **Diego Heredia Sánchez**.

Basado en una obra de [José Juan Sánchez Hernández](#)



El contenido de esta web está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).