

■ Índice

1. RESUMEN DE CONSULTAS.....	3
2. INTRODUCCION.....	8
2.1 BREVE HISTORIA.....	8
2.2 COMPONENTES DEL SQL	8
2.3 COMANDOS.....	8
2.4 CLÁUSULAS	9
2.5 OPERADORES LÓGICOS.....	9
2.6 OPERADORES DE COMPARACIÓN	10
2.7 FUNCIONES DE AGREGADO.....	10
2.8 PRECEDENCIA DE LOS OPERADORES	10
2.9 ORDEN DE EJECUCIÓN DE LOS COMANDOS.....	11
3. CONSULTAS DE SELECCIÓN (SELECT FROM)	12
3.1 CONSULTAS BÁSICAS.....	12
3.2 REUNIR VARIAS TABLAS: CONSULTAS DE UNIÓN INTERNAS (INNER JOIN, NATURAL JOIN)	13
3.2.1 INNER JOIN	13
3.2.2 COMO REUNIR 3 O MÁS TABLAS	14
3.2.3 OTRAS MANERAS DE REUNIR 2 TABLAS:	15
3.2.3.1 NATURAL JOIN.....	15
3.2.3.2 Producto cartesiano con WHERE	15
3.3 ORDER BY: ORDENAR LOS REGISTROS	16
3.3.1 LIMIT	17
3.3.2 DISTINCT /DISTINCTROW.....	17
3.4 ALIAS: AS	17
3.5 INDICAR EN QUE TABLA O BASE DE DATOS SE ENCUENTRA LA INFORMACIÓN.....	18
4. CRITERIOS DE SELECCIÓN (WHERE)	19
4.1 LA CLÁUSULA WHERE	20
4.2 OPERADORES ARITMETICOS EN MySQL.....	21
4.3 OPERADORES DE COMPARACIÓN	21
4.4 OPERADORES LÓGICOS.....	22
4.5 INTERVALOS DE VALORES.....	23
4.6 IS NULL / IS NOT NULL	24
4.7 EL OPERADOR LIKE.....	25
#Es preferible esta búsqueda con =, frente a la que emplea LIKE más abajo.....	25
4.8 EL OPERADOR IN.....	26
4.9 CUÁNDO USAR = / IN, <> / NOT IN, > / > ALL() / > ANY()	26
4.10 IFNULL, COALESCE.....	27
4.11 FECHAS, COMO TRABAJAR CON FECHAS.....	28
4.11.1 EJEMPLOS DE CONSULTAS POR FECHAS:	28
4.11.2 CAMBIAR EL FORMATO DE SALIDA DE UNA FECHA (POCO IMPORTANTE).....	29
4.11.3 FUNCIONES PARA TRABAJAR CON FECHAS	29
4.11.4 DATE_FORMAT().....	31
4.12 REALIZAR CÁLCULOS CON FECHAS	33

4.13	EXPRESIONES REGULARES.....	33
4.13.1	33
5.	<u>FUNCIONES DE AGREGADO: SUM, AVG, MAX, MIN, COUNT.....</u>	37
5.1	AVG.....	37
5.2	COUNT	38
5.3	MAX, MIN.....	39
5.4	SUM	39
5.5	STDEV, STDEVP	40
5.6	VARIANCE	40
5.7	EFFECTO DE LOS VALORES NULL SOBRE LAS FUNCIONES DE AGREGADO:.....	40
6.	<u>AGRUPAMIENTO DE REGISTROS (GROUP BY)</u>	41
6.1	GROUP BY	41
6.1.1	HAVING.....	41
6.2	SELECT INTO OUTFILE	42
7.	<u>CONSULTAS DE ACCIÓN (INSERT, UPDATE, DELETE)</u>	42
7.1	INSERT INTO	42
7.1.1	INSERTAR UNA FILA EN UNA TABLA:	43
7.1.1.1	Opción A: Indicar valores para todos los atributos de la fila:.....	43
7.1.1.2	Opción B: Indicar sólo algunos atributos de la fila:.....	43
7.1.2	INSERTAR VARIAS FILAS CON UN ÚNICO COMANDO INSERT:	44
7.1.3	INSERTAR UNA FILA EN UNA TABLA QUE INCLUYE UN ATRIBUTO QUE TIENE LA PROPIEDAD AUTO_INCREMENT	44
7.1.4	EXPLICACIÓN MEDIANTE EJEMPLOS DE VARIAS ALTERNATIVAS DE INSERT	45
7.1.5	INSERTAR FILAS OBTENIENDO LOS DATOS MEDIANTE UNA CONSULTA A OTRAS TABLAS	46
7.1.6	LOAD DATA INFILE: INSERTAR FILAS PROCEDENTES DE UN FICHERO	46
7.1.7	REPLACE: UTILIZA LOS DATOS DE LA TABLA B PARA ACTUALIZAR O INSERTAR DATOS EN LA TABLA A	46
7.1.8	DESHABILITAR EL CHEQUEO DE LAS FOREIGN KEYS PARA PODER INSERTAR CIERTAS FILAS QUE INCUMPLEN	
	RESTRICCIONES DE FOREIGN KEY:.....	47
7.1.8.1	SET FOREIGN_KEY_CHECKS = 0 consideraciones a tener en cuenta.....	48
7.2	UPDATE.....	49
7.2.1	RESTRICCIÓN DEL WORKBENCH QUE IMPIDE UPDATE O DELETE DEBIDO AL SAFE UPDATES CUANDO NO SE	
	INCLUYE UNA KEY EN EL WHERE O NO SE USA LIMIT.....	50
7.3	DELETE	51
7.3.1	TRUNCATE	52
8.	<u>SUBCONSULTAS.....</u>	54
9.	<u>CONSULTAS DE UNIÓN EXTERNAS.....</u>	59
10.	<u>INNER JOIN, NATURAL JOIN, LEF JOIN, RIGHT JOIN, FULL JOIN</u>	61
10.1	INNER JOIN: JUNTAR 2 TABLAS PARA TAREAS NORMALES.....	61
10.2	NATURAL JOIN	62
10.2.1	OTRAS OPCIONES (MENOS INTERESANTES) PARA REALIZAR INNER JOIN:.....	62
10.3	OTROS TIPOS DE JOIN: CROSS JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN	63

10.4	TIPOS DE JOIN	66
10.5	EJEMPLOS DE DISTINTOS TIPOS DE JOIN:	67
11.	CREATE TABLE	69
11.1	ÍNDICES: PRIMARY KEY, UNIQUE, INDEX.....	69
11.1.1	ORDEN DE COLOCACIÓN EN CREATE TABLE DE LOS DISTINTOS ÍNDICES:	69
11.1.2	PRIMARY KEY	70
11.1.2.1	3 maneras distintas de definir la PRIMARY KEY:	70
11.1.3	UNIQUE [INDEX KEY].....	71
11.1.4	INDEX KEY	72
11.2	FOREIGN KEY	76
11.2.1	ON DELETE/UPDATE [RESTRICT CASCADE SET NULL NO ACTION]	76
11.2.2	ERRORES TÍPICOS AL CREAR UNA FOREIGN KEY	76
11.2.3	EJEMPLOS CORRECTOS E INCORRECTOS DE FOREIGN KEY QUE PROVIENEN DE PRIMARY KEY FORMADA POR MÁS DE UN ATRIBUTO	78
11.3	NULL / NOT NULL	79
11.4	DEFAULT.....	79
11.4.1	VALOR IMPLÍCITO POR DEFECTO DE CADA TIPO DE DATO.	81
11.4.2	DATETIME, DATE, TIME, TIMESTAMP: ASIGNAR VALORES DEFAULT.....	81
11.5	AUTO_INCREMENT = 77	83
11.6	CHECK	83
11.6.1	COMANDOS PARA AÑADIR, MODIFICAR, BORRAR CHECKS.....	85
11.7	CREAR UNA TABLA NUEVA A PARTIR DE OTRA TABLA EXISTENTE, PERO SIN LAS KEYS DE LA ORIGINAL	85
11.8	CREAR UNA TABLA TEMPORAL (EN MEMORIA RAM) QUE INCLUYA CIERTAS FILAS DE OTRA TABLA.	85
12.	ALTER TABLE.....	85
12.1.1	MODIFICAR LOS ATRIBUTOS DE UNA TABLA:.....	86
12.1.2	AÑADIR UNA FOREIGN KEY A UNA TABLA	87
13.	ANEXOS.....	87
13.1	RESOLUCIÓN DE PROBLEMAS	87
13.1.1	BUSCAR INFORMACIÓN DUPLICADA EN UN CAMPO DE UNA TABLA.	87
13.1.2	RECUPERAR REGISTROS DE UNA TABLA QUE NO CONTENGAN REGISTROS RELACIONADOS EN OTRA.	87

Fuente. Este documento es una copia modificada del siguiente documento:

Tutorial de SQL por Claudio Casares

<http://www.asptutor.com/zip/sql.pdf>

<http://www.aprendeaprogramar.com/course/view.php?id=8>

al que se han añadido algunos contenidos elaborados por mí.

1. Resumen de consultas

```
SELECT nombre, telefono
FROM clientes;
# Esta consulta devuelve los campos nombre y teléfono de la tabla clientes.
```

```
SELECT *
FROM empleados;
# muestra todos los atributos de la tabla

SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY CodigoPostal, Nombre;
```

```
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY CodigoPostal DESC , Nombre ASC;
```

```
SELECT *
FROM productos
ORDER BY precio_actual DESC
LIMIT 5
# si existiesen 6 productos de precio más alto porque hay varios que empatan, solo se muestran 5,
el sistema descarta una fila.
```

```
SELECT *
FROM productos
ORDER BY precio_actual DESC
LIMIT 4, 2
```

```
SELECT AVG(salario), MIN(salario)
FROM empleados # está BIEN: todas son funciones de agregado
```

```
SELECT COUNT(*), COUNT(comision), COUNT(oficio), COUNT(DISTINCT oficina) FROM empleados;

# Resultado:
5, 3, 4, 3
```

```
SELECT SUM(salario)
FROM empleados;
```

```
SELECT DISTINCT (oficio), dep_no FROM empleados; #DISTINCT no se aplica sólo a oficina.
```

```
SELECT dnombre AS 'nombre del departamento' , localidad
```

```
FROM departamentos;
```

```
SELECT Apellidos, Salario
FROM empleados
WHERE Salario > 21000;
```

```
SELECT *
FROM pedidos
WHERE Fecha_Envio = '1994/10/25';
```

```
SELECT Apellidos, Nombre
FROM empleados
WHERE Apellidos = 'King';
```

```
SELECT Apellidos, Salario
FROM empleados
WHERE Salario BETWEEN 200 AND 300;
```

```
SELECT Apellidos, Nombre, Ciudad
FROM empleados
WHERE Ciudad IN ('Sevilla', 'Los Angeles', 'Barcelona');
```

```
SELECT apellidos, salario + comisión
FROM empleados;
```

```
SELECT apellidos, salario * 1.05
FROM empleados;
```

```
SELECT *
FROM empleados
WHERE (edad >= 25 AND edad <= 50) OR sueldo = 100;
WHERE (edad BETWEEN 25 AND 50) OR sueldo = 100;

WHERE (edad > 25 AND edad < 50) OR sueldo = 100;
WHERE (edad BETWEEN 26 AND 49) OR sueldo = 100;
```

```
SELECT *
FROM empleados
WHERE comision IS NULL;
```

```
WHERE comison = NULL;  
# ERROR
```

```
SELECT * FROM clientes WHERE codigoPostal LIKE '15%'  
# Busca los clientes cuyo código Postal comienza por 15
```

```
SELECT * FROM productos WHERE codigo LIKE '__AM_'  
# Busca los productos cuyo código tiene 5 caracteres, de los cuales el 3º y 4º son AM.
```

```
SELECT * FROM pedidos  
WHERE Provincia IN ('Madrid', 'Barcelona', 'Sevilla');  
# Busca los pedidos cuya provincia es una de las siguientes: 'Madrid', 'Barcelona', 'Sevilla'
```

La expresión con IN anterior equivale a esta otra:

```
SELECT * FROM pedidos  
WHERE Provincia = 'Madrid' OR Provincia = 'Barcelona' OR Provincia = 'Sevilla'
```

```
#error:  
SELECT IFNULL(salario + complemento, 0)  
#debería ser así:  
SELECT IFNULL(salario, 0) + IFNULL(complemento, 0)
```

```
#buscar una fecha concreta:  
SELECT * FROM pedidos WHERE fecha_pedido = '1999-10-6';  
  
#buscar todos los días del mes de octubre del año 1999:  
SELECT * FROM pedidos WHERE fecha_pedido LIKE '1999-10-%';  
  
SELECT * FROM pedidos  
WHERE YEAR(fecha_pedido) = 1999  
AND MONTH(fecha_pedido) IN (10,11,12);
```

```
SELECT id_Familia, SUM(Stock)  
FROM productos  
GROUP BY id_Familia;
```

```
SELECT Id_Familia SUM(Stock)  
FROM Productos  
GROUP BY Id_Familia  
HAVING SUM (Stock) > 100;
```

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no  
FROM empleados  
INNER JOIN departamentos  
ON empleados.dep_no = departamentos.dep_no;
```

#el equivalente con NATURAL JOIN:

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no  
FROM empleados  
NATURAL JOIN departamentos ;
```

```
SELECT dep_no, apellido1, dnombre  
FROM empleados  
INNER JOIN departamentos  
ON empleados.dep_no = departamentos.dep_no;  
#produce error de atributo ambiguo porque existe en ambas tablas y el servidor no sabe cual  
mostrar
```

2. INTRODUCCION

El lenguaje de consulta estructurado (SQL) es un lenguaje de base de datos normalizado, utilizado por los diferentes motores de bases de datos para realizar determinadas operaciones sobre los datos o sobre la estructura de los mismos. Pero como sucede con cualquier sistema de normalización hay excepciones para casi todo; de hecho, cada motor de bases de datos tiene sus peculiaridades y lo hace diferente de otro motor, por lo tanto, el lenguaje SQL normalizado (ANSI) no nos servirá para resolver todos los problemas, aunque si se puede asegurar que cualquier sentencia escrita en ANSI será interpretable por cualquier motor de datos.

2.1 Breve Historia

La historia de SQL (que se pronuncia deletreando en inglés las letras que lo componen, es decir "ese-cu-ele" y no "siquel" como se oye a menudo) empieza en 1974 con la definición, por parte de Donald Chamberlin y de otras personas que trabajaban en los laboratorios de investigación de IBM, de un lenguaje para la especificación de las características de las bases de datos que adoptaban el modelo relacional. Este lenguaje se llamaba SEQUEL (Structured English Query Language) y se implementó en un prototipo llamado SEQUEL-XRM entre 1974 y 1975. Las experimentaciones con ese prototipo condujeron, entre 1976 y 1977, a una revisión del lenguaje (SEQUEL/2), que a partir de ese momento cambió de nombre por motivos legales, convirtiéndose en SQL. El prototipo (System R), basado en este lenguaje, se adoptó y utilizó internamente en IBM y lo adoptaron algunos de sus clientes elegidos. Gracias al éxito de este sistema, que no estaba todavía comercializado, también otras compañías empezaron a desarrollar sus productos relacionales basados en SQL. A partir de 1981, IBM comenzó a entregar sus productos relacionales y en 1983 empezó a vender DB2. En el curso de los años ochenta, numerosas compañías (por ejemplo Oracle y Sybase, sólo por citar algunos) comercializaron productos basados en SQL, que se convierte en el estándar industrial de hecho por lo que respecta a las bases de datos relacionales.

En 1986, el ANSI adoptó SQL (sustancialmente adoptó el dialecto SQL de IBM) como estándar para los lenguajes relacionales y en 1987 se transformó en estándar ISO. Esta versión del estándar va con el nombre de SQL/86. En los años siguientes, éste ha sufrido diversas revisiones que han conducido primero a la versión SQL/89 y, posteriormente, a la actual SQL/92.

El hecho de tener un estándar definido por un lenguaje para bases de datos relacionales abre potencialmente el camino a la intercomunicabilidad entre todos los productos que se basan en él. Desde el punto de vista práctico, por desgracia las cosas fueron de otro modo. Efectivamente, en general cada productor adopta e implementa en la propia base de datos sólo el corazón del lenguaje SQL (el así llamado Entry level o al máximo el Intermediate level), extendiéndolo de manera individual según la propia visión que cada cual tenga del mundo de las bases de datos.

Actualmente, está en marcha un proceso de revisión del lenguaje por parte de los comités ANSI e ISO, que debería terminar en la definición de lo que en este momento se conoce como SQL3. Las características principales de esta nueva encarnación de SQL deberían ser su transformación en un lenguaje stand-alone (mientras ahora se usa como lenguaje hospedado en otros lenguajes) y la introducción de nuevos tipos de datos más complejos que permitan, por ejemplo, el tratamiento de datos multimediales.

2.2 Componentes del SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

2.3 Comandos

Existen dos tipos de comandos SQL:

- los comandos DML (Data Manipulation Language) que permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.
- los comandos DDL (Data Definition Language) que permiten crear y definir nuevas bases de datos, campos e índices.

Comandos DDL	
Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices
DROP	Empleado para eliminar tablas e índices
ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Comandos DML	
Comando	Descripción
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados
DELETE	Utilizado para eliminar registros de una tabla de una base de datos

2.4 Cláusulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

2.5 Operadores Lógicos

Operador	Uso
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve el valor verdadero sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve el valor verdadero si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

2.6 Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor ó Igual que
>=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo
IN	Utilizado para especificar registros de una base de datos

2.7 Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

En Mysql estas funciones no tienen en cuenta los valores NULL. En Access si algún valor es nulo la función devuelve valor NULL

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

2.8 Precedencia de los operadores

(los operadores que están encima se ejecuta antes, los del mismo nivel se evalúan de izquierda a derecha).

INTERVAL
BINARY, COLLATE

!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&

= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR,
= (assignment), :=

2.9 Orden de ejecución de los comandos

Dada una sentencia SQL de selección que incluye todas las posibles cláusulas, el orden de ejecución de las mismas es el siguiente:

- 1) Cláusula FROM
- 2) Cláusula WHERE
- 3) Cláusula GROUP BY
- 4) Cláusula HAVING
- 5) Cláusula SELECT
- 6) Cláusula DISTINCT
- 7) Cláusula ORDER BY
- 8) Cláusula LIMIT

3. Consultas de Selección (SELECT FROM)

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos, esta información es devuelta en forma filas y columnas., donde cada fila representa a un ejemplar y cada columna se asocia a un atributo.

3.1 Consultas básicas

La sintaxis básica de una consulta de selección es la siguiente:

```
SELECT atributo1, atributo2, ...  
FROM tabla;
```

En donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

```
SELECT nombre, telefono  
FROM clientes;  
# Esta consulta devuelve los campos nombre y teléfono de la tabla clientes.
```

Para mostrar todos los atributos, se puede usar * en lugar de nombrar uno por uno todos los atributos en el SELECT:

```
SELECT *  
FROM empleados;  
# muestra todos los atributos de la tabla
```

No es conveniente abusar del * ya que obligamos al motor de la base de datos a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados.

3.2 Reunir varias tablas: Consultas de Unión Internas (INNER JOIN, NATURAL JOIN)

3.2.1 INNER JOIN

Las uniones entre tablas se realizan normalmente mediante la cláusula **INNER JOIN** que combina registros de dos tablas siempre que haya concordancia de valores en un campo común (clave primaria de una tabla con clave ajena de otra tabla). Su sintaxis es:

```
SELECT columna1, columna2,...
FROM tabla1
    INNER JOIN tabla2
    ON tabla1.atributo1 = tabla2.atributo2
```

Los atributos que se usan para combinar las tablas son la **clave primaria en una tabla y clave ajena en la otra tabla**.

Por ejemplo se puede utilizar **INNER JOIN** con las tablas *departamentos* y *empleados* para mostrar los datos de los empleados junto con información del departamento en que trabajan. El ejemplo siguiente muestra cómo podría combinar las tablas *empleados* y *departamentos* basándose en el campo *dep_no* (que es **PRIMARY KEY** de la tabla departamentos y **FOREIGN KEY** en la tabla empleados):

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no
FROM empleados
    INNER JOIN departamentos
    ON empleados.dep_no = departamentos.dep_no;
```

#	dep_no	nombre	apellido
1	30	María	Alonso
2	10	Rosa María	López
3	30	Ana	Martín
4	30	Luis Manuel	Garrido
5	10	Juan Rodrigo Fernando	Rey
6	10	Ana Patricia	Rev

En el ejemplo anterior, para incluir el atributo *dep_no* en el SELECT es necesario especificar la tabla en que debe buscarlo (*departamentos.dep_no*), porque ese atributo existe en las dos tablas que se han reunido y provocaría un error de ambigüedad si no se especificase la tabla, como sucede en este ejemplo **erróneo**:

```
SELECT dep_no, apellido1, dnombre
FROM empleados
    INNER JOIN departamentos
    ON empleados.dep_no = departamentos.dep_no;
#produce error de atributo ambigüo porque existe en ambas tablas y el servidor no sabe cual
mostrar
```

Si empleamos la cláusula **INNER JOIN** en la consulta, se reunirán sólo aquellos registros de las

dos tablas que están relacionados entre si. Si utilizamos LEFT JOIN actuará como un INNER JOIN pero se incluirán además los registros de la tabla de la izquierda que no se relacionan con ningún registro de la tabla derecha. Si utilizamos RIGHT JOIN actuará como un INNER JOIN pero se incluirán además los registros de la tabla de la derecha que no se relacionan con ningún registro de la tabla izquierda.

1. LEFT JOIN y RIGHT JOIN se explican en otro apartado

```
SELECT productos.producto_no, productos.descripcion, pedidos.pedido_no, pedidos.producto_no
FROM pedidos INNER JOIN productos
ON productos.producto_no = pedidos.producto_no;
```

Veamos las dos tablas y el resultado de la consulta anterior: no aparecen los productos que no han sido comprados nunca (que no se relacionan con algún pedido)

A	B	C	D
1	SELECT producto_no, descripcion FROM productos;	1	SELECT pedido_no, producto_no FROM pedidos;
2	prod descrip	2	pedido_no producto_no
3	'10' 'MESA DESPACHO MOD. GAVIOTA'	3	'1002' '10'
4	'20' 'SILLA DIRECTOR MOD. BUFALO'	4	'1008' '10'
5	'30' 'ARMARIO NOGAL DOS PUERTAS'	5	'1017' '20'
6	'40' 'MESA MODELO UNIÓN'	6	'1004' '40'
7	'50' 'ARCHIVADOR CEREZO'	7	'1001' '50'
8	'60' 'CAJA SEGURIDAD MOD B222'	8	'1007' '50'
9	'70' 'DESTRUCTORA DE PAPEL A3'	9	'7777' '555'
10	'80' 'MODULO ORDENADOR MOD. ERGOS'		
11			

3.2.2 Como reunir 3 o más tablas

Se pueden reunir 3 o más tablas de modo similar a cómo se unen 2: tenemos que añadir en cada paso una tabla que esté directamente relacionada con las anteriores.

Ejemplo de cómo se unen las tablas: empleados, departamentos, clientes y pedidos:

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no
FROM empleados
INNER JOIN departamentos
ON empleados.dep_no = departamentos.dep_no
INNER JOIN clientes
ON clientes.vendedor_no = empleados.emp_no
INNER JOIN pedidos
ON pedidos.cliente_no = clientes.cliente_no;
```

- Primero unimos 2 tablas que tienen relación directa (empleados y departamentos) lo que da lugar a una supertabla A,
- Después unimos esa supertabla A con la tabla clientes (que se relaciona con alguna de las anteriores, concretamente con empleados), dando lugar a otra supertabla B (más grande que la anterior).
- Después unimos esa supertabla B con la tabla pedidos (que se relaciona con alguna de las anteriores, concretamente con clientes), dando lugar a otra supertabla C (más grande que la anterior). De este modo la supertabla C está formada por la reunión de 4 tablas.

3.2.3 Otras maneras de reunir 2 tablas:

3.2.3.1 NATURAL JOIN

NATURAL JOIN permite reunir 2 tablas sin especificar que atributos se usan para la unión: el servidor utiliza aquellos atributos que tienen el mismo nombre en ambas tablas:

Sólo se puede usar cuando se cumplen ciertas condiciones (ver más abajo).

Como no se especifica que atributos se usan para la unión: el servidor utiliza aquellos atributos que tienen el mismo nombre en ambas tablas.

Restricciones:

- Si los atributos necesarios tienen nombres distintos **no puede usarse este método**.
- Si existen atributos con el mismo nombre pero que no están relacionados, entonces **tampoco** podemos usar NATURAL JOIN.

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no  
FROM empleados  
NATURAL JOIN departamentos ;
```

#el equivalente con INNER JOIN:

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no  
FROM empleados  
INNER JOIN departamentos  
ON empleados.dep_no = departamentos.dep_no;
```

3.2.3.2 Producto cartesiano con WHERE

Este otro método es un poco peligroso, utiliza el producto cartesiano en el FROM (separa las tablas con coma), y después se queda sólo con las combinaciones correctas añadiendo una cláusula WHERE. Decimos que es un poco peligroso porque si nos olvidamos de la cláusula WHERE se combinan todas las filas de todas las tablas, lo cual no tiene sentido.

```
SELECT apellido1, apellido2, dnombre, departamentos.dep_no  
FROM empleados , departamentos  
WHERE empleados.dep_no = departamentos.dep_no ;
```

3.3 ORDER BY: Ordenar los registros

Adicionalmente se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la cláusula **ORDER BY** Lista de Campos. En donde Lista de campos representa los campos a ordenar. Ejemplo:

```
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY Nombre;
```

Esta consulta devuelve los campos CodigoPostal, Nombre, Telefono de la tabla Clientes ordenados por el campo Nombre.

Se pueden ordenar los registros por más de un campo, como por ejemplo:

```
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY CodigoPostal, Nombre;
```

Incluso se puede especificar el orden de los registros: ascendente mediante la cláusula (**ASC** -se toma este valor por defecto) ó descendente (**DESC**)

```
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY CodigoPostal DESC , Nombre ASC;
```

Los NULL se ponen al final de la lista ordenada si el orden es decreciente y al principio si el orden es creciente

En lugar del nombre del atributo se puede poner qué posición ocupa ese atributo en el SELECT, eso es obligatorio cuando en lugar de un atributo simple en el SELECT se muestra una expresión, y opcional cuando el SELECT muestra un atributo sencillo:

```
#aquí es obligatorio referirse en el ORDER BY a la columna COUNT mediante su posición
SELECT oficio, COUNT(*) AS 'Número de empleados con ese oficio'
FROM empleados
GROUP BY oficio
HAVING COUNT(*) > 2
ORDER BY 2 DESC, 1
;
```

```
#aquí es obligatorio referirse en el ORDER BY a la columna COUNT mediante su posición
SELECT oficio, COUNT(*) AS 'Número de empleados con ese oficio'
FROM empleados
GROUP BY oficio
HAVING COUNT(*) > 2
ORDER BY 2 DESC, oficio
;
```

```
#aquí no es optativo referirse en el ORDER BY a las columnas mediante su posición
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY 1 DESC , 2 ASC;
```

```
#aquí no es optativo referirse en el ORDER BY a las columnas mediante su posición
SELECT CodigoPostal, Nombre, Telefono
FROM Clientes
ORDER BY CodigoPostal DESC , 2 ASC;
```


3.3.1 LIMIT

```
LIMIT [ número de filas a descartar, ] númeroDeFilasDeseadas  
LIMIT 5  
LIMIT 4, 2
```

Devuelve un cierto número de registros que se encuentran al principio de un rango ordenado por una cláusula ORDER BY. Supongamos que queremos recuperar los nombres de los 5 productos de precio más alto:

```
SELECT *  
FROM productos  
ORDER BY precio_actual DESC  
LIMIT 5  
# si existiesen 6 productos de precio más alto porque hay varios que empatan, solo se muestran 5,  
el sistema descarta una fila.
```

Si no se incluye la cláusula ORDER BY, la consulta devolverá un conjunto arbitrario de 5 registros de la tabla productos.

En el siguiente ejemplo se saltan las 4 primeras filas y se devuelven las 2 siguientes:

```
SELECT *  
FROM productos  
ORDER BY precio_actual DESC  
LIMIT 4, 2
```

3.3.2 DISTINCT /DISTINCTROW

En MySQL ambos comandos (DISTINCT /DISTINCTROW) son idénticos, en otros SGBD son distintos.

Este comando fuerza que no se muestren filas repetidas.

Por ejemplo para mostrar en que localidades tenemos clientes y que no aparezcan repetidas:

```
SELECT DISTINCT localidad FROM clientes;
```

DISTINCT se aplica a todos los atributos mostrados, por ello estas dos instrucciones son iguales:

```
SELECT DISTINCT (oficio), dep_no FROM empleados; #DISTINCT no se aplica sólo a oficio.  
SELECT DISTINCT oficio, dep_no FROM empleados;
```

3.4 Alias: AS

En determinadas circunstancias es necesario asignar un nombre a alguna columna determinada de un conjunto devuelto, otras veces por simple capricho o por otras circunstancias. Para resolver todas ellas tenemos la palabra reservada **AS** que se encarga de asignar el nombre que deseamos a la columna deseada. Tomado como referencia el ejemplo anterior podemos hacer que la columna devuelta por la consulta, en lugar de llamarse apellido (igual que el campo devuelto) se llame Empleado. En este caso procederíamos de la siguiente forma:

```
SELECT dnombre AS 'nombre del departamento' , localidad
FROM departamentos;
```

3.5 Indicar en que tabla o base de datos se encuentra la información

En MySQL para indicar la ubicación completa de una tabla o de un atributo se puede especificar, separando con punto la ubicación:

```
baseDeDatos.tabla.atributo
```

```
baseDeDatos.tabla
```

```
tabla.atributo
```

■ ejemplos:

```
SELECT dep_no, nombre
FROM bdempleados.empleados;
# en el SELECT especifica en que tabla se encuentra el atributo dep_no
```

```
SELECT empleados.dep_no, nombre
FROM empleados;
# en el SELECT especifica en que tabla se encuentra el atributo dep_no
```

Ejemplo de consulta que accede a tablas situadas en dos bases de datos distintas (db1, db2):

```
SELECT db1.tablaA.*, db2.tablaB.atributoX
FROM db1.tablaA INNER JOIN db2.tablaB ON tablaA.id = tablaB.id;
```

De este modo una sola consulta puede acceder a varias bases de datos distintas.

4. Criterios de Selección (WHERE)

En el capítulo anterior se vio la forma de recuperar los registros de las tablas, las formas empleadas devolvían todos los registros de la mencionada tabla. A lo largo de este capítulo se estudiarán las posibilidades de filtrar los registros con el fin de recuperar solamente aquellos que cumplan unas condiciones preestablecidas.

Antes de comenzar el desarrollo de este capítulo hay que recalcar tres detalles de vital importancia. El primero de ellos es que cada vez que se desee establecer una condición referida a un campo de texto la condición de búsqueda debe ir encerrada entre comillas simples; la segunda es que no se puede establecer condiciones de búsqueda en los campos memo y; la tercera y última hace referencia a las fechas. Las fechas se deben escribir siempre entre comillas y en formato 'AAAA/MM/DD' en donde AAAA representa el año, MM representa el mes, y DD representa el día. Por ejemplo si deseamos referirnos al día 3 de Septiembre de 1995 deberemos hacerlo de la siguiente forma '1995/09/03'.

4.1 La cláusula WHERE

La cláusula WHERE puede usarse para determinar qué filas de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. Si no se emplea esta cláusula, la consulta devolverá todas las filas de la tabla. WHERE es opcional, pero cuando aparece debe ir a continuación de FROM.

```
SELECT Apellidos, Salario
FROM empleados
WHERE Salario > 21000;
```

```
SELECT Id_Producto, Existencias
FROM productos
WHERE Existencias <= 3;
```

```
SELECT *
FROM pedidos
WHERE Fecha_Envio = '1994/10/25';
```

```
SELECT Apellidos, Nombre
FROM empleados
WHERE Apellidos = 'King';
```

```
SELECT Apellidos, Nombre
FROM empleados
WHERE Apellidos LIKE 'S_';
```

```
SELECT Apellidos, Salario
FROM empleados
WHERE Salario BETWEEN 200 AND 300;
```

```
SELECT Apellidos, Salario
FROM empleados
WHERE Apellidos BETWEEN 'Lon' AND 'Tol';
```

```
SELECT Id_Pedido, Fecha_Pedido
FROM pedidos
WHERE Fecha_Pedido BETWEEN '1994/1/1' AND '1994/06/30';
```

```
SELECT Apellidos, Nombre, Ciudad
FROM empleados
WHERE Ciudad IN ('Sevilla', 'Los Angeles', 'Barcelona');
```

4.2 OPERADORES ARITMETICOS EN MySQL

/	División con decimales en el resultado
-	Resta
+	Suma
*	Multipliación
DIV	División entera (divide 2 enteros y proporciona el cociente sin decimales (entero resultante de truncar los decimales, sin redondear) Ej: 7 DIV 2 produce 3
%, MOD	Operador Modulo (divide 2 enteros y proporciona el resto de esa división). Ej: 7 % 2 produce 1

```
SELECT apellidos, salario + comisión
FROM empleados;
```

```
SELECT apellidos, salario * 1.05
FROM empleados;
```

4.3 Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor o Igual que
>=	Mayor o Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores (incluye los extremos)
LIKE	Utilizado en la comparación de un patrón que usa comodines: %, _ Si no se usan comodines debe usarse = por ser más eficiente
IN	Utilizado para especificar varios valores posibles.

4.4 Operadores Lógicos

OPERADOR	significado
AND, &&	Y
OR, 	O
XOR	XOR (OR EXCLUSIVO)
NOT	NO (INVIERTE EL VALOR LÓGICO)
IS NULL	comprueba si es NULL
IS NOT NULL	comprueba si NO es NULL

Los operadores lógicos soportados por SQL son: AND, OR, XOR, IS y NOT. A excepción del último todos poseen la siguiente sintaxis:

```
<expresión1> operador <expresión2>
```

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico. La **tabla de verdad** adjunta muestra los diferentes posibles resultados:

<expresión1>	Operador	<expresión2>	Resultado
Verdad	AND	Falso	Falso
Verdad	AND	Verdad	Verdad
Falso	AND	Verdad	Falso
Falso	AND	Falso	Falso
Verdad	OR	Falso	Verdad
Verdad	OR	Verdad	Verdad
Falso	OR	Verdad	Verdad
Falso	OR	Falso	Falso
Verdad	XOR	Verdad	Falso
Verdad	XOR	Falso	Verdad
Falso	XOR	Verdad	Verdad
Falso	XOR	Falso	Falso
	NOT	Verdad	Falso
	NOT	Falso	Verdad

```
SELECT *
FROM empleados
WHERE edad > 25 AND edad < 50;
```

```
SELECT *
FROM empleados
WHERE (edad > 25 AND edad < 50) OR sueldo = 100;
```

```
SELECT *  
FROM empleados  
WHERE NOT estado = 'Soltero';
```

```
SELECT *  
FROM empleados  
WHERE estado <> 'Soltero';
```

```
SELECT *  
FROM empleados  
WHERE (sueldo > 100 AND sueldo < 500) OR (provincia = 'Madrid' AND estado = 'casado');
```

```
SELECT *  
FROM empleados  
WHERE comision IS NULL;
```

4.5 Intervalos de Valores

Para indicar que deseamos recuperar los registros según el intervalo de valores de un campo emplearemos el operador BETWEEN cuya sintaxis es:

```
campo [NOT] BETWEEN valor1 AND valor2  
# el corchete indica que la cláusula NOT es opcional.
```

En este caso la consulta devolvería los registros que contengan en "campo" un valor incluido en el intervalo valor1, valor2 (ambos inclusive). valor1 debe ser menor o igual que valor2. Si antepone la condición NOT devolverá aquellos valores no incluidos en el intervalo.

```
SELECT *  
FROM pedidos  
WHERE codigoPostal BETWEEN '28000' AND '28999';
```

```
SELECT *  
FROM empleados  
WHERE salario BETWEEN 1000 AND 1500;
```

```
SELECT *  
FROM empleados  
WHERE salario NOT BETWEEN 1000 AND 1500;  
#VERDADERO PARA: 998, 999, 1501, 1502,...
```

4.6 IS NULL / IS NOT NULL

■ IS NULL

IS NULL: La única manera de averiguar si una celda está vacía es con esta cláusula:

```
SELECT *  
FROM empleados  
WHERE comison IS NULL;  
# mostrar a los empleados con el atributo comisión vacío
```

El siguiente código es **erróneo**, no se puede usar = para encontrar valores NULL

```
WHERE comison = NULL;  
# ERROR
```

■ IS NOT NULL

IS NOT NULL: La única manera de averiguar si una celda no está vacía es con esta cláusula:

```
SELECT *  
FROM empleados  
WHERE comison IS NOT NULL;  
# mostrar a los empleados con el atributo comisión NO vacío
```


4.7 El Operador LIKE

Se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL. Su sintaxis es:

expresión **LIKE** *modelo*

En donde *expresión* es una cadena modelo o campo contra el que se compara expresión. Se puede utilizar el operador LIKE para encontrar valores en los campos que coincidan con el modelo especificado. Por modelo puede especificar un valor completo ('Ana María'), o se pueden utilizar caracteres comodín para encontrar un rango de valores.

Los COMODINES que se pueden usar con **LIKE** son estos 2:

CARÁCTER	FUNCIÓN
%	0, 1, 2,... caracteres (equivale al * de Windows)
_	1 caracter (equivalen al ? de Windows)

Ejemplos:

```
SELECT * FROM clientes WHERE codigo LIKE '15%'
# Busca los clientes cuyo código comienza por 15, su longitud puede ser de 2, 3, 4, ... caracteres.

SELECT * FROM productos WHERE codigo LIKE '__AM_'
# Busca los productos cuyo código tiene 5 caracteres, de los cuales el 3º y 4º son AM.

SELECT * FROM productos WHERE codigo LIKE '__AM%'
# Busca los productos cuyo código tiene 4 o más caracteres, de los cuales el 3º y 4º son AM.

SELECT * FROM productos WHERE codigo LIKE '%H'
# Busca los productos cuyo código termina en H, su longitud puede ser de 1, 2, ... caracteres.
```

Cuando se buscan cadenas de texto sin comodines, entonces el servidor es más rápido si la búsqueda se realiza con el operador = y no con el operador LIKE:

```
SELECT * FROM clientes WHERE ciudad = 'FERROL'
# Es preferible esta búsqueda con =, frente a la que emplea LIKE más abajo

SELECT * FROM clientes WHERE ciudad LIKE 'FERROL'
```

4.8 El Operador IN

Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los en una lista. Su sintaxis es:

expresión [NOT] IN (*valor1*, *valor2*...)

```
SELECT * FROM pedidos
WHERE Provincia IN ('Madrid', 'Barcelona', 'Sevilla');
# Busca los pedidos cuya provincia es una de las siguientes: 'Madrid', 'Barcelona', 'Sevilla'
```

La expresión con IN anterior equivale a esta otra:

```
SELECT * FROM pedidos
WHERE Provincia = 'Madrid' OR Provincia = 'Barcelona' OR Provincia = 'Sevilla'
```

```
SELECT * FROM empleados
WHERE numeroHijos IN (2, 4);
# Escoge a los empleados con 2 o 4 hijos, los que tienen 3 no valen.
```

Como utilizar IN con varios atributos simultáneamente:

```
SELECT * FROM clientes
WHERE (localidad, vendedor_no) IN (
```

```
    SELECT localidad, vendedor_no
    FROM clientes
    WHERE cliente_no = 101
);
```

La consulta anterior busca los clientes que son de la misma localidad que el cliente 101 y además son atendidos por el mismo vendedor que atiende al cliente 101:

4.9 Cuándo usar = / IN, <> / NOT IN, > / > ALL() / > ANY()

= (1 valor)	IN (0,1,2... valores)
<>(1valor)	NOT IN (0,1,2... valores)
> (1 valor)	> ALL (0,1,2... valores) <i>mayor que todos los valores del listado</i> > SOME (0,1,2... valores) <i>mayor que algún valor del listado</i> > ANY (0,1,2... valores) <i>mayor que algún valor del listado</i>
<,<=,>=,	se hacen de la misma forma que >

4.10 IFNULL, COALESCE

IFNULL(atributo, valor si nulo)

IFNULL(atributo, valor) convierte un atributo NULL en el valor indicado.

Un ejemplo típico es convertir el valor NULL en 0 para que al sumar ese atributo con otro número, el resultado no sea NULL, porque debemos recordar que: $\text{NULL} + 5000 = \text{NULL}$.
Ejemplo:

```
SELECT IFNULL(salario, 0) + IFNULL(complemento, 0)
```

sin embargo sería **ERRÓNEO** solucionarlo así, porque se suma antes de transformar los NULL en 0:

#error:

```
SELECT IFNULL(salario + complemento, 0)
```

#debería ser así:

```
SELECT IFNULL(salario, 0) + IFNULL(complemento, 0)
```

COALESCE (value1, value2, value3, value4...)

COALESCE () admite varios parámetros de forma genérica, y en ese caso: devuelve el primer valor no NULL de la lista, o NULL cuando no exista ningún valor distinto a NULL en la lista:

COALESCE (atributo, valor si nulo)

en este uso su función es similar a la de IFNULL.

```
SELECT COALESCE (salario, 0) + COALESCE (complemento, 0)
```

4.11 Fechas, como trabajar con fechas

<http://forums.mysql.com/read.php?71,154336,154380>
<https://dev.mysql.com/doc/refman/5.6/en/date-and-time-functions.html>
 Realizar cálculos con fechas: [http://dev.mysql.com/doc/refman/5.7/en/date-](http://dev.mysql.com/doc/refman/5.7/en/date-calculations.htm)
[calculations.htm](http://dev.mysql.com/doc/refman/5.7/en/date-calculations.htm)

El único formato con el que se pueden almacenar las fechas en MySQL es el americano:

'YYYY/MM/DD'	'YYYY-MM-DD'	'YYYY.MM.DD'	'YYYY@MM@DD'
(año/mes/día)			
'2013/05/21'	'2013-05-21'	'2013.05.21'	

Para insertar una fecha es obligatorio proporcionarla en el formato YYYY-MM-DD, como separador se pueden usar 4 símbolos: (-./@). Ejemplo:

```
INSERT INTO mytabla (fecha) VALUES ('2013-05-21'), ('2013.05.22'), ('2013/05/23'), ('2013@05@24'));
```

4.11.1 Ejemplos de consultas por fechas:

Buscar fechas aprovechando que son similares a cadenas de texto

```
#buscar una fecha concreta:
SELECT * FROM pedidos WHERE fecha_pedido = '1999-10-6';

#buscar todos los días del mes de octubre del año 1999:
SELECT * FROM pedidos WHERE fecha_pedido LIKE '1999-10-%';

SELECT * FROM pedidos WHERE fecha_pedido LIKE '1999-1-%';
/* Selecciona el mes: 1 */

SELECT * FROM pedidos WHERE fecha_pedido LIKE '1999-1%';
/* CUIDADO, selecciona los meses: 1, 10, 11, 12, que posiblemente no es lo que deseamos*/

#buscar todos los días del año 1999:
SELECT * FROM pedidos WHERE fecha_pedido LIKE '1999-%';
```

Buscar un rango de fechas:

```
#Buscar los pedidos realizados de octubre a diciembre del año 1999: 3 soluciones posibles:

SELECT * FROM pedidos
WHERE fecha_pedido >= '1999-10-1'
AND fecha_pedido <= '1999-12-31';

SELECT * FROM pedidos
WHERE fecha_pedido BETWEEN '1999-10-1' AND '1999-12-31';

SELECT * FROM pedidos
WHERE YEAR(fecha_pedido) = 1999
AND MONTH(fecha_pedido) IN (10,11,12);
```

Más búsquedas por rango de fechas:

```
#Buscar los pedidos realizados en los años 1997 a 1999: 2 soluciones posibles:

SELECT * FROM pedidos
WHERE fecha_pedido >= '1997-1-1'
AND fecha_pedido < '2000-1-1';
```

```
SELECT * FROM pedidos
WHERE YEAR(fecha_pedido) IN (1997, 1998, 1999);
```

4.11.2 Cambiar el formato de salida de una fecha (poco importante)

Este apartado es poco importante porque simplemente tiene efectos cosméticos y no se suele emplear en SQL (son otros programas, que muestran la información a los usuarios, los que suelen modificar el formato).

Para mostrar una fecha almacenada tenemos varias opciones:

- Mostrarla en el **formato americano**
- Especificar un **formato distinto**: función **DATE_FORMAT()**

Código de ejemplo:

```
CREATE TABLE mytabla (
  id int(11) NOT NULL auto_increment,
  fecha varchar(10) default NULL,
  PRIMARY KEY (id)
);

INSERT INTO mytabla (fecha) VALUES ('2013/5/23'), ('2013/5/24'), ('2013/5/25'),
('2013/5/26'), ('2013/5/27');

SELECT * FROM mytabla;

SELECT id, DATE_FORMAT(fecha, '%d/%m/%y') AS fecha FROM mytabla;

SELECT id, DATE_FORMAT(fecha, '%d/%m/%Y') AS fecha FROM mytabla;

SELECT id, DATE_FORMAT(fecha, '%e/%c/%Y') AS fecha FROM mytabla;
```

4.11.3 Funciones para trabajar con fechas

Funciones destacadas para trabajar con fechas:

```
YEAR( ), MONTH( ), DAY( ), DATEDIFF( ), NOW( )
```

NAME	DESCRIPTION
ADDDATE()	Add time values (intervals) to a date value
ADDTIME()	Add time
CURDATE()	Return the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Return the current time
DATE_ADD()	Add time values (intervals) to a date value
DATE_FORMAT()	Format date as specified

NAME	DESCRIPTION
DATE_SUB()	Subtract a time value (interval) from a date
DATE()	Extract the date part of a date or datetime expression
DATEDIFF()	Subtract two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Return the name of the weekday
DAYOFMONTH()	Return the day of the month (0-31)
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
EXTRACT()	Extract part of a date
FROM_DAYS()	Convert a day number to a date
FROM_UNIXTIME()	Format UNIX timestamp as a date
GET_FORMAT()	Return a date format string
HOUR()	Extract the hour
LAST_DAY	Return the last day of the month for the argument
LOCALTIME() , LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP() , LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Create a date from the year and day of year
MAKETIME()	Create time from hour, minute, second
MICROSECOND()	Return the microseconds from argument
MINUTE()	Return the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Return the name of the month
NOW()	Return the current date and time
PERIOD_ADD()	Add a period to a year-month
PERIOD_DIFF()	Return the number of months between periods
QUARTER()	Return the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format
SECOND()	Return the second (0-59)
STR_TO_DATE()	Convert a string to a date
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments
SUBTIME()	Subtract times
SYSDATE()	Return the time at which the function executes

NAME	DESCRIPTION
TIME_FORMAT()	Format as time
TIME_TO_SEC()	Return the argument converted to seconds
TIME()	Extract the time portion of the expression passed
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Subtract an interval from a datetime expression
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()	Return a UNIX timestamp
UTC_DATE()	Return the current UTC date
UTC_TIME()	Return the current UTC time
UTC_TIMESTAMP()	Return the current UTC date and time
WEEK()	Return the week number
WEEKDAY()	Return the weekday index
WEEKOFYEAR()	Return the calendar week of the date (1-53)
YEAR()	Return the year
YEARWEEK()	Return the year and week
Name	Description

4.11.4 DATE_FORMAT()

http://www.w3schools.com/sql/func_date_format.asp

Código de campo cambiado

Función que cambia el formato en que se muestra una fecha.

DATE_FORMAT(date,format)

Ejemplo:

```
SELECT DATE_FORMAT(NOW(), '%b %d %Y %h:%i %p')
SELECT DATE_FORMAT(NOW(), '%m-%d-%Y')
SELECT DATE_FORMAT(NOW(), '%d %b %Y')
SELECT DATE_FORMAT(NOW(), '%d %b %Y %T:%f')
```

```
-- salida:
Nov 04 2014 11:45 PM
11-04-2014
04 Nov 14
04 Nov 2014 11:45:34:243
```

Tabla con los códigos de formato:

FORMATO	DESCRIPCIÓN
%a	Abbreviated weekday name (Sun-Sat)
%b	Abbreviated month name (Jan-Dec)
%c	Month, numeric (0-12)
%D	Day of month with English suffix (0th, 1st, 2nd, 3rd,)
%d	Day of month, numeric (00-31)
%e	Day of month, numeric (0-31)
%f	Microseconds (000000-999999)
%H	Hour (00-23)
%h	Hour (01-12)
%I	Hour (01-12)
%i	Minutes, numeric (00-59)
%j	Day of year (001-366)
%k	Hour (0-23)
%l	Hour (1-12)
%M	Month name (January-December)
%m	Month, numeric (00-12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00-59)
%s	Seconds (00-59)
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00-53) where Sunday is the first day of week
%u	Week (00-53) where Monday is the first day of week
%V	Week (01-53) where Sunday is the first day of week, used with %X
%v	Week (01-53) where Monday is the first day of week, used with %x
%W	Weekday name (Sunday-Saturday)
%w	Day of the week (0=Sunday, 6=Saturday)
%X	Year for the week where Sunday is the first day of week, four digits, used with %V

FORMATO	DESCRIPCIÓN
%x	Year for the week where Monday is the first day of week, four digits, used with %v
%Y	Year, numeric, four digits
%y	Year, numeric, two digits

4.12 Realizar cálculos con fechas

<http://dev.mysql.com/doc/refman/5.7/en/date-calculations.htm>

4.13 Expresiones regulares

<https://www.geeksforgeeks.org/mysql-regular-expressions-regexp/>
<https://dev.mysql.com/doc/refman/8.0/en/regexp.html>
<https://www.w3resource.com/mysql/string-functions/mysql-regexp-function.php>

4.13.1

Pattern	What the Pattern matches
*	Zero or more instances of string preceding it
+	One or more instances of strings preceding it
.	Any single character
?	Match zero or one instances of the strings preceding it.
^	caret(^) matches Beginning of string
\$	End of string
[abc]	Any character listed between the square brackets
[^abc]	Any character not listed between the square brackets
[A-Z]	match any upper case letter.
[a-z]	match any lower case letter
[0-9]	match any digit from 0 through to 9.
[[:<:]]	matches the beginning of words.
[[:>:]]	matches the end of words.
[[:class:]]	matches a character class i.e. [:alpha:] to match letters, [:space:] to match white space, [:punct:] is match punctuations and [:upper:] for upper class letters.
p1 p2 p3	Alternation; matches any of the patterns p1, p2, or p3
{n}	n instances of preceding element
{m,n}	m through n instances of preceding element

La barra invertida (\) se usa como un carácter de escape. Si queremos usarlo como parte del patrón en una expresión regular, debemos usar barras diagonales inversas dobles (\\)

Carbonizarse	Descripción	Ejemplo
*	El metacaracter de asterisco (*) se usa para coincidir con cero (0) o más instancias de las cadenas que lo preceden	SELECCIONE * FROM películas DONDE el título REGEXP 'da *'; dará todas las películas que contengan caracteres "da". Por ejemplo, Da Vinci Code, Daddy's Little Girls.
+	El metacarácter más (+) se usa para coincidir con una o más instancias de cadenas que lo preceden.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP 'mon +'; dará todas las películas que contengan caracteres "mon". Por ejemplo, Ángeles y Demonios.
?	El metacarácter de pregunta(?) Se usa para hacer coincidir cero (0) o una instancia de las cadenas que lo preceden.	SELECCIONAR * FROM `categories` WHERE `category_name` REGEXP 'com?'; dará todas las categorías que contienen cadena com. Por ejemplo, comedia, comedia romántica.
.	El metacarácter punto (.) Se usa para hacer coincidir cualquier carácter individual a excepción de una nueva línea.	SELECCIONE * FROM películas WHERE `year_released` REGEXP '200.'; dará todas las películas lanzadas en los años comenzando con los caracteres "200" seguidos de cualquier carácter individual. Por ejemplo, 2005,2007,2008 etc.
[abc]	La charlist [abc] se usa para hacer coincidir cualquiera de los caracteres incluidos.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '[vwxyz]'; dará todas las películas que contengan cualquier carácter en "vwxyz". Por ejemplo, X-Men, código Da Vinci.Daddy's Little Girls, etc.
[^abc]	La charlist [^abc] se usa para unir cualquier caracter	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '^[^vwxyz]'; dará

	excluyendo los que están encerrados.	todas las películas que contengan caracteres distintos de los de "vwxyz".
[ARIZONA]	El [AZ] se utiliza para hacer coincidir cualquier letra mayúscula.	SELECCIONAR * FROM `miembros` WHERE `postal_address` REGEXP '[AZ]'; dará a todos los miembros que tengan una dirección postal que contenga cualquier carácter de la A a la Z. Por ejemplo, Janet Jones con número de miembro 1.
[Arizona]	El [az] se usa para hacer coincidir cualquier letra minúscula	SELECT * FROM `members` WHERE `postal_address` REGEXP '[az]'; dará a todos los miembros que tengan direcciones postales que contengan cualquier carácter de aaz. . Por ejemplo, Janet Jones con membresía número 1.
[0-9]	El [0-9] se usa para unir cualquier dígito desde 0 hasta 9.	SELECCIONAR * FROM `miembros` DONDE `contact_number` REGEXP '[0-9]'; dará a todos los miembros que hayan enviado números de contacto que contengan caracteres "[0-9]". Por ejemplo, Robert Phil.
^	El símbolo de intercalación (^) se utiliza para comenzar el partido al principio.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '^ [cd]'; da todas las películas con el título que comienza con cualquiera de los caracteres en "cd". Por ejemplo, Code Name Black, Daddy's Little Girls y Da Vinci Code.
	La barra vertical () se usa para aislar alternativas.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '^ [cd] ^ [u]'; da todas las películas con el título que comienza con cualquiera de los caracteres en "cd" o "u". Por ejemplo, Code Name Black, Daddy's Little Girl, Da Vinci Code y Underworld – Awakening.

[[:<:]]	El [[:<:]] coincide con el comienzo de las palabras.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '[[:<:]] para'; da todas las películas con títulos que comienzan con los personajes. Por ejemplo: Olvidarse de Sarah Marshal.
[[:>:]]	El [[:>:]] coincide con el final de las palabras.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP 'ack [[:>:]]'; da todas las películas con títulos que terminan con los caracteres "ack". Por ejemplo, Code Name Black.
[:clase:]	La [:clase:] coincide con una clase de caracteres, es decir[: alfa:] para unir letras, [: espacio:] para hacer coincidir el espacio en blanco, [: puntuacion:] es puntuacion de coincidencia y [: superior:] para letras de clase superior.	SELECCIONAR * FROM `películas` DONDE `título` REGEXP '[[: alfa:]]'; da todas las películas con títulos que contienen solo letras. Por ejemplo, Olvidando a Sarah Marshal, X-Men, etc. Película como Pirates of the Caribbean 4 será omitida por esta consulta.

La barra invertida (\) se usa como un carácter de escape. Si queremos usarlo como parte del patrón en una expresión regular, debemos usar barras diagonales inversas dobles (\\)

5. FUNCIONES DE AGREGADO: SUM, AVG, MAX, MIN, COUNT

ADVERTENCIAS: Si utilizamos una función de agregado (MAX, MIN, COUNT, AVG, SUM,...) todas las demás columnas del SELECT tienen que ser funciones de agregado. Es decir:

```
SELECT AVG(salario), MIN(salario)
FROM empleados # está BIEN: todas son funciones de agregado
```

```
SELECT salario, apellido
FROM empleados
```

```
SELECT AVG(salario), apellido
FROM empleados # está MAL, incluye un atributo puro
```

5.1 AVG

Calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta. Su sintaxis es la siguiente

```
AVG(expr)
```

En donde *expr* representa el campo que contiene los datos numéricos para los que se desea calcular la media o una expresión que realiza un cálculo utilizando los datos de dicho campo. La media calculada por AVG es la media aritmética (la suma de los valores dividido por el número de valores). La función AVG no incluye ningún campo NULL en el cálculo. Si todos los sumandos son NULL, entonces devuelve NULL

En MySQL esta función no tienen en cuenta los valores nulos. En Access si algún valor es nulo la función devuelve valor nulo.

En MySQL el paréntesis tiene que ir pegado a la función (sin espacio en blanco), si no produce error:

```
AVG(salario) # bien
AVG (salario) # ERROR: hay espacios en blanco
```

```
SELECT AVG(precio_actual)
FROM productos;
```

```
SELECT AVG(salario) AS 'Salario Medio'
FROM empleados WHERE salario > 1000
```

5.2 COUNT

Calcula el número de registros devueltos por una consulta. Su sintaxis es la siguiente

COUNT(*expr*)

En donde *expr* contiene el nombre del campo que desea contar, los operandos de *expr* pueden incluir el nombre de un campo de una tabla, asterisco, una constante o una función. Puede contar cualquier tipo de datos incluso texto.

COUNT(*) #cuenta el número de filas de la tabla, incluyendo aquellos que contienen campos NULL.

COUNT(*atributo*) #cuenta el número de filas en las que el atributo no es NULL.

COUNT(DISTINCT *atributo*) cuenta valores distintos y no nulos del *atributo*.

SELECT COUNT(*) AS Total FROM pedidos;

La función COUNT no cuenta los registros que tienen campos NULL a menos que *expr* sea el carácter comodín asterisco (*)

COUNT(*) cuenta el número total de registros que devuelve la consulta.

COUNT(*) es considerablemente más rápida que COUNT(*atributo*). No se debe poner el asterisco entre comillas ('*').

Aunque *expr* puede realizar un cálculo sobre un campo, COUNT simplemente cuenta el número de registros sin tener en cuenta qué valores se almacenan en los registros.

Ejemplo que usa una versión reducida de la tabla empleados:

COMISION	OFICIO
100	vendedor
NULL	vendedor
30	NULL
NULL	director
40	contable

SELECT COUNT(*), COUNT(*comision*), COUNT(*oficio*), COUNT(DISTINCT *oficio*) FROM empleados;

#Resultado:
5, 3, 4, 3

5.3 MAX, MIN

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo específico de una consulta. Su sintaxis es:

```
MIN(expr)
MAX(expr)
```

```
SELECT MIN(salario) FROM empleados;
SELECT MIN(gastos) FROM pedidos WHERE pais = 'España';
SELECT MAX(gastos) AS 'Máximo gasto' FROM pedidos WHERE pais = 'España';
```

Donde *expr* es el campo sobre el que se desea realizar el cálculo. *expr* puede incluir el nombre de un campo de una tabla, una constante o una función.

Se pueden usar atributos numéricos, fechas o texto:

```
#MIN(fecha) fecha más antigua, MAX (fecha) fecha más reciente
SELECT MIN(fecha_nacimiento) FROM empleados;
SELECT MAX(fecha_nacimiento) FROM empleados;

#MIN(apellido) apellido con letra más baja, MAX (apellido) apellido con letra más alta
SELECT MIN(apellido) FROM empleados;
SELECT MAX(apellido) FROM empleados;
```

En MySQL esta función **no tienen en cuenta los valores nulos**

5.4 SUM

Devuelve la suma del conjunto de valores contenido en un campo específico de una consulta. Su sintaxis es:

```
SUM(expr)
```

```
SELECT SUM(salario)
FROM empleados;

SELECT SUM(precioUnidad * cantidad) AS 'total'
FROM detallePedido;
```

En donde *expr* respresenta el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de *expr* pueden incluir el nombre de un campo de una tabla, una constante o una función.

En MySQL esta función **no tienen en cuenta los valores nulos**. Si todos los sumandos son NULL, entonces devuelve NULL

5.5 STDEV, STDEVP

Devuelve estimaciones de la desviación estándar para la población (el total de los registros de la tabla) . Su sintaxis es:

```
STDEV(expr)
```

```
SELECT StDev(gastos) AS 'Desviacion' FROM pedidos WHERE pais = 'España';
```

En donde *expr* representa el nombre del campo que contiene los datos que desean evaluarse o una expresión que realiza un cálculo utilizando los datos de dichos campos.

5.6 VARIANCE

Devuelve una estimación de la varianza de una población

```
VARIANCE(expr)
```

```
SELECT VARIANCE(gastos) AS 'Varianza' FROM pedidos WHERE pais = 'España';
```

5.7 Efecto de los valores NULL sobre las funciones de agregado:

MAX, MIN, AVG, SUM no tienen en cuenta los valores NULL en sus cálculos.

Ejemplo que usa una versión reducida de la tabla empleados:

COMISION	OFICIO
100	vendedor
NULL	vendedor
30	NULL
NULL	director
40	contable

```
SELECT COUNT(comision), COUNT(oficio), COUNT(DISTINCT oficina) FROM empleados;
```

#Resultado:

5, 3, 4, 3

```
SELECT SUM(comision), AVG(comision), MAX(comision), MIN(comision) FROM empleados;
```

#Resultado:

170, 56.6666, 100, 30

6. Agrupamiento de Registros (GROUP BY)

6.1 GROUP BY

Agrupar los registros en *cajas* (también podemos llamarlas *grupos*): crea una *caja (grupo)* para cada valor distinto del atributo que aparece en la cláusula GROUP BY. cada fila de la table se coloca en la *caja (grupo)* que le corresponde. Tras la ejecución del GROUP BY ya no tenemos una tabla, sino que tenemos varias cajas, cada fila de la tabla original se ha colocado en la caja que le corresponde, por ese motivo ya no podemos poner cualquier cosa en el SELECT:

Los campos que aparezcan en la lista de campos de SELECT deben o bien aparecer en la cláusula GROUP BY o bien aparecer como argumentos de una función SQL de agregado (MAX, MIN,...).

La sintaxis de una consulta que incluye la cláusula GROUP BY es:

```
SELECT campos FROM tabla WHERE criterio
GROUP BY campos por los que se agrupa.
```

ejemplo:

```
#Agrupar los distintos productos según la familia a la que pertenecen; para cada familia mostrar
cómo se llama la familia y cuál es la suma de los stocks de los productos de esa familia.
```

```
SELECT id_Familia, SUM(Stock)
FROM productos
GROUP BY id_Familia;
```

```
#esto sería incorrecto, porque en el SELECT aparece el atributo nombre fuera de una función de
agregado.
```

```
SELECT id_Familia, nombre, SUM(Stock)
FROM productos
GROUP BY id_Familia;
```

GROUP BY es opcional.. Los valores NULL en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores NULL no se evalúan en ninguna de las funciones SQL agregadas.

Es opcional utilizar la cláusula WHERE para excluir filas antes de agruparlas.

Se utiliza la cláusula HAVING para eliminar algunos grupos que han sido creados con la cláusula GROUP BY.

En la lista de campos del GROUP BY puede incluirse cualquier campo de las tablas que aparecen en la cláusula FROM, incluso si el campo no está incluido en la instrucción SELECT.

6.1.1 HAVING

Una vez que GROUP BY ha agrupado los registros, HAVING muestra sólo los grupos creados por la cláusula GROUP BY que satisfagan las condiciones de la cláusula HAVING.

HAVING es en cierto sentido similar a WHERE: WHERE determina qué filas se seleccionan. Una vez que las filas se han agrupado utilizando GROUP BY, HAVING determina qué grupos se van a mostrar.

```
#Agrupar los distintos productos según la familia a la que pertenecen; para cada familia mostrar
cómo se llama la familia y cuál es la suma de los stocks de los productos de esa familia. pero
(HAVING) mostrar sólo los grupos que tienen una suma de STOCK superior a 100.
```

```
SELECT Id_Familia SUM(Stock)
FROM Productos
GROUP BY Id_Familia
```

```
HAVING SUM (Stock) > 100;
```

6.2 SELECT INTO OUTFILE

<https://www.mysqlfaq.net/mysql-faq/Export-Data/How-to-use-SELECT-INTO-OUTFILE-statement-to-export-data>
<https://stackoverflow.com/questions/32737478/how-should-i-resolve-secure-file-priv-in-mysql>

Guarda en un fichero los datos que genera un SELECT

```
SELECT * INTO OUTFILE 'clientesFerrol.txt'  
FROM bdempleados.clientes  
WHERE localidad = 'Ferrol';
```

si produce error, podemos probar esto:

```
SHOW VARIABLES LIKE "secure_file_priv";  
'secure_file_priv', '/var/lib/mysql-files/'  
  
# con lo que sabemos que podemos guardar la salida en la carpeta '/var/lib/mysql-files/'
```

```
SELECT emp_no, salario, fecha_alta  
INTO OUTFILE '/var/lib/mysql-files/file1.csv'  
FROM empleados;
```

```
SELECT emp_no, salario, fecha_alta  
INTO OUTFILE '/var/lib/mysql-files/file1.csv'  
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'  
LINES TERMINATED BY '\n'  
FROM empleados;
```

7. Consultas de Acción (INSERT, UPDATE, DELETE)

Las consultas de acción son aquellas que no devuelven ningún registro (FILA), son las encargadas de acciones como añadir, borrar y modificar registros (FILAS) en las tablas.

7.1 INSERT INTO

INSERT INTO Agrega una fila a una tabla. Esta operación puede obtener los datos de orígenes distintos:

- Insertar en una tabla filas cuyos **valores especifcamos nosotros en el comando**.
- Insertar en una tabla filas que **se obtienen mediante una operación SELECT realizada sobre otra tabla**.

```
# espcificamos nosotros los valores
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', 'Becario', '2010/6/22');
```

```
# los datos provenientes de un SELECT de otra tabla
INSERT INTO clientes
SELECT * FROM clientesViejos;
```

Recordar además que los datos a insertar deben escribirse así:

- atributos de tipo **numérico** (INT, DOUBLE, DECIMAL, BOOLEAN): sin comillas.
- atributos de tipo **texto** (VARCHAR, TEXT...): escritos entre comillas.
- atributos de tipo **fecha** (DATE, DATETIME): escritos entre comillas y las partes de una fecha se ordenan como 'aaaa/mm/dd' o 'aaaa-mm-dd'...

```
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', 'Becario', '2010/6/22', '2010/6/22 22:56:45.234');
```

7.1.1 insertar una fila en una tabla:

7.1.1.1 Opción A: Indicar valores para todos los atributos de la fila:

```
INSERT INTO table_name VALUES
(valor1, valor2, valor3, ...);
```

- En este caso hay que dar valor a todos los atributos.
- En caso de no querer darle valor a un atributo es obligatorio asignarle **NULL** (sin comillas)

```
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', 'Becario', '2010/6/22');
```

```
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', NULL, '2010/6/22');
```

7.1.1.2 Opción B: Indicar sólo algunos atributos de la fila:

Cuando sólo se va a asignar valor de algunos atributos de la fila, entonces debemos **especificar previamente el nombre de los atributos a los que se dará valor**. Es obligatorio dar valor a aquellos atributos que tienen la restricción NOT NULL y no se les ha especificado un valor DEFAULT.

```
INSERT INTO table_name (column1, column2, column3 ...)
VALUES (value1, value2, value3 ...);
```

```
#proporciono sólo 3 atributos (la tabla tiene más). Los atributos que no proporciono, o bien tienen
valor por defecto (DEFAULT) o bien no son obligatorios (no son NOT NULL).
INSERT INTO empleados (nombre, apellido, cargo)
VALUES ('Luis', 'Sánchez', 'Becario');
```

7.1.2 Insertar varias filas con un único comando INSERT:

Para insertar 2 o más filas podemos:

- Insertar cada fila con su propio INSERT
- Utilizar un único INSERT que inserte varias filas (es más rápido para el servidor).

Sintaxis:

```
#Opción A:
INSERT INTO table_name VALUES
(value1,value2,value3,..., valueN),
(value1,value2,value3,..., valueN);

#Opción B:
INSERT INTO table_name (column1,column2,column3,...) VALUES
(value1,value2,value3,..., valueJ),
(value1,value2,value3,..., valueJ);
```

Ejemplos:

```
#utilizar varios INSERT, cada uno inserta una fila
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', 'Becario', '2018/8/12');
INSERT INTO empleados
VALUES (5678, 'Ana', 'Gago', 'Comercial', '2020/6/22');
```

```
#utilizar un solo INSERT (las filas se separan con coma).
INSERT INTO empleados
VALUES (1234, 'Luis', 'Sánchez', 'Becario', '2018/8/12'),
(5678, 'Ana', 'Gago', 'Comercial', '2020/6/22');
```

Insertar varias filas mediante un único comando INSERT es más rápido que generar un comando INSERT por cada fila: El motivo es que se reduce la sobrecarga de comunicaciones entre cliente y servidor:

<https://dev.mysql.com/doc/refman/8.0/en/insert-optimization.html>

Existen más mecanismos para acelerar la inserción de filas en una tabla:

<https://dev.mysql.com/doc/refman/8.0/en/insert-optimization.html>
<https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-bulk-data-loading.html>

7.1.3 Insertar una fila en una tabla que incluye un atributo que tiene la propiedad AUTO_INCREMENT

Como regla general: a un atributo con la propiedad **AUTO_INCREMENT** no debemos asignarle valor (es el propio servidor quien debe hacerlo), por tanto tenemos estas 3 opciones para insertar una fila que incluye un atributo con la propiedad **AUTO_INCREMENT**:

- **Opción A:** Indicar el valor NULL para el atributo y el sistema le asignará a esa celda el número que le corresponda.
- **Opción B:** no incluir el atributo en la lista de atributos que vamos a dar.
- **Opción C:** indicar un valor para el atributo y entonces el servidor asignará ese valor y después continuará a partir de ese número cuando en el futuro se inserten nuevas filas (el valor debe ser mayor que todos los existentes para que no haya colisiones en el futuro).

```

CREATE TABLE Persona(
  IdPersona INT NOT NULL AUTO_INCREMENT,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  PRIMARY KEY (IdPersona)
)
#Insertar datos:

#opción A
INSERT INTO Persona
VALUES (NULL, 'Rodriguez', 'Manuel', 'Calle del sol 45', 'Ferrol');

#opción B
INSERT INTO Persona (apellido, nombre, direccion, localidad)
VALUES ('Rodriguez', 'Manuel', 'Calle del sol 45', 'Ferrol');

#opción C, non a utilizaremos nuca:
INSERT INTO Persona
VALUES (5467, 'Rodriguez', 'Manuel', 'Calle del sol 45', 'Ferrol');

```

7.1.4 Explicación mediante ejemplos de varias alternativas de INSERT

```

DROP TABLE IF EXISTS A;

CREATE TABLE A(
  b UNSIGNED INT NOT NULL AUTO_INCREMENT,
  c VARCHAR(3) NOT NULL,
  d VARCHAR(15),
  PRIMARY KEY (b)
);

```

```

INSERT INTO A VALUES
(NULL, 'AAA', 'Ferrol');

INSERT INTO A(c, d) VALUES
('BBB', 'Narón');

INSERT INTO A VALUES
(13, 'E13', NULL);

INSERT INTO A VALUES
(25, 'C25', 'Cedeira');

INSERT INTO A VALUES
(NULL, 'DDD', 'Ferrol');

```

b	c	d
1	AAA	Ferrol
2	BBB	Narón
13	E13	
25	C25	Cedeira
26	DDD	Ferrol

```
SELECT * FROM A;
```

b	c	d
1	AAA	Ferrol
2	BBB	Narón
13	E13	
25	C25	Cedeira
26	DDD	Ferrol

7.1.5 Insertar filas obteniendo los datos mediante una consulta a otras tablas

Sintaxis:

```
#Opción A:
INSERT INTO table_name
sentencia SELECT;

#Opción B, especificando las columnas:
INSERT INTO table_name (column1, column2, column3,...)
sentencia SELECT;
```

Ejemplos:

```
#Opción A: escoger todos los atributos
INSERT INTO AntiguosEmpleados
SELECT *
FROM empleados
WHERE fecha_Contratacion < '2015/1/25';

#Opción B: escoger algunos atributos
INSERT INTO AntiguosEmpleados (emp_no, apellido1, apellido2, nombre, telefono)
SELECT emp_no, apellido1, apellido2, nombre, telefono
FROM empleados
WHERE fecha_Contratacion < '2015/1/25';
```

7.1.6 LOAD DATA INFILE: Insertar filas procedentes de un fichero

1. The LOAD DATA INFILE statement reads rows from a text file into a table at a very high speed.
2. LOAD DATA INFILE is the complement of [SELECT... INTO OUTFILE](#).

```
LOAD DATA INFILE '/var/lib/mysql-files/pedidos2000servidor.txt' INTO TABLE
bdempleados.pedidos2000;
```

3. El fichero no puede tener líneas en blanco ni por el medio ni al final (no se puede meter carácter de nueva línea al final de la última línea).
4. For security reasons, when reading text files located on the server, the files must either reside in the database directory or be readable by the user account used to run the server.
5. Also, to use LOAD DATA INFILE on server files, you must have the FILE privilege.

1) Ejemplo

```
USE bdempleados;

LOAD DATA INFILE '/var/lib/mysql-files/pedidos2000servidor.txt' INTO TABLE
bdempleados.pedidos2000;
```

7.1.7 REPLACE: utiliza los datos de la tabla B para actualizar o insertar datos en la tabla A

REPLACE inserta las filas de la tabla B en la tabla A, funciona como INSERT' excepto cuando la TABLA A tiene una fila cuya **PRIMARY KEY** (o algún índice **UNIQUE**)

coincide con la fila de la tabla B que se está insertando: en ese caso la fila original de la tabla A es borrada antes de insertar la fila procedente de la tabla B.

```
#Opción A: escoger todos los atributos
REPLACE INTO tabla_A
SELECT *
FROM tabla_B
;

#Opción B: escoger algunos atributos
REPLACE INTO tabla_A (dni, telefono, direccion)
SELECT (dni, telefono, direccion)
FROM tabla_B
;
```

7.1.8 Deshabilitar el chequeo de las FOREIGN KEYs para poder insertar ciertas filas que incumplen restricciones de FOREIGN KEY:

Veamos el siguiente caso: deseamos insertar estas 2 filas pero no se puede porque hay un error de FOREIGN KEY

```
INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO)
VALUES ('9001', '14256821Y', 'Luis', 'Yañez', 'Rioboo', 'ADMINISTRATIVO', '9002', '2016-
04-05', 150000, 0, '10', '652986532');

INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO)
VALUES ('9002', '45325698P', 'Ana', 'Rinlo', 'Rois', 'INGENIERA', '8904', '2016-04-05',
150000, 0, '10', '547852146');
```

¿A qué se debe este error?:

El problema es que la primera fila a insertar incumple la FOREIGN KEY en el atributo JEFE: intentamos insertar un empleado (9001) cuyo jefe (9002) todavía no existe: no existe el empleado 9002 cuando intentamos insertar al empleado 9001

```
#Manera de evitarlo 1:
#deshabilitar chequeo de las FOREIGN KEY
SET FOREIGN_KEY_CHECKS=0;

#insertardatos
INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO)
VALUES ('9001', '14256821Y', 'Luis', 'Yañez', 'Rioboo', 'ADMINISTRATIVO', '9002', '2016-04-05',
'150000', '0', '10', '652986532');

INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO)
VALUES ('9002', '45325698P', 'Ana', 'Rinlo', 'Rois', 'INGENIERA', '8904', '2016-04-05', '150000', '0',
'10', '547852146');

#habilitar chequeo de las FOREIGN KEY
SET FOREIGN_KEY_CHECKS=1;

#el problema es que al volver a habilitar el chequeo de las FOREIGN KEY no se comprueba si las
filas introducidas previamente tienen algún error en sus FOREIGN KEY.
#Tenemos que comprobar si alguna clave ajena es errónea, para ello buscamos esas posibles filas
erróneas mediante esta consulta: buscar aquellos empleados cuyo jefe no existe.
SELECT *
```

```
FROM empleados
WHERE jefe NOT IN (
    SELECT emp_no
    FROM empleados
);
```

#Manera de evitarlo 2:

#ordenar los INSERT de modo que no se infrinjan las FOREIGN KEY (si fuesen muchas filas sería muy complicado ordenarlas todas, y en algunos casos sería imposible si hubiese referencias cruzadas)

```
INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO) VALUES ('9002', '45325698P',
'Ana', 'Rinlo', 'Rois', 'INGENIERA', '8904', '2016-04-05', '150000', '0', '10', '547852146');
INSERT INTO bdempleados.empleados (EMP_NO, DNI, NOMBRE, APELLIDO1, APELLIDO2, OFICIO,
JEFE, FECHA_ALTA, SALARIO, COMISION, DEP_NO, TELEFONO) VALUES ('9001', '14256821Y',
'Luís', 'Yañez', 'Rioboo', 'ADMINISTRATIVO', '9002', '2016-04-05', '150000', '0', '10',
'652986532');
```

7.1.8.1 SET FOREIGN_KEY_CHECKS = 0 consideraciones a tener en cuenta

```
SET foreign_key_checks = 0; #deshabilitar
```

```
INSERT...
```

```
SET foreign_key_checks = 1;#rehabilitar
```

Deshabilitar las claves ajenas: el servidor deja de comprobar si se cumplen las restricciones de las claves ajenas en todas las tablas de todas las bases de datos.

En algunas ocasiones se utiliza "SET foreign_key_checks = 0" para introducir datos sin tener que preocuparse del orden en que se introducen las filas: de esta manera no importa que introduzcamos una fila que hace referencia a otra fila que todavía no hemos introducido.

Pero CUIDADO!!!:

Al rehabilitar las FOREIGN KEY, el sistema no comprueba si los datos ya existentes en la tabla cumplen las FOREIGN KEY que se acaban de rehabilitar, con lo que podríamos tener datos que no cumplieren las FOREIGN KEY.

Sin embargo: si una tabla ya contiene datos y después añadimos una nueva FOREIGN KEY, en este caso el sistema comprobará si todos los datos previos de la tabla cumplen esa FOREIGN KEY, y si alguno de ellos no la cumple, entonces se produce un error y no se crea la FOREIGN KEY.

- SET foreign_key_checks = 0:
 - Sí Deshabilita la comprobación de las claves FOREIGN KEY existentes en las tablas.
 - No Deshabilita la comprobación UNIQUE de las claves primarias.
 - No Deshabilita las restricciones UNIQUE.
- SET foreign_key_checks = 1:

- Vuelve a habilitar la comprobación de las claves FOREIGN KEY existentes en las tablas
- Pero **no comprueba si los datos ya existentes en la tabla cumplen las FOREIGN KEY.**

Si deshabilitamos las FOREIGN KEY con `SET foreign_key_checks = 0;` permanecerán deshabilitadas hasta que se habiliten con el comando `SET foreign_key_checks = 1;` o hasta que se reinicie el servidor.

7.2 UPDATE

UPDATE es una consulta de actualización que cambia los valores de uno o varios campos de una tabla especificada basándose en un criterio. Su sintaxis es:

```
UPDATE tabla
SET Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN
WHERE Criterio;
```

UPDATE es especialmente útil cuando se desea cambiar un gran número de filas o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez.

El ejemplo siguiente incrementa 2.5% el salario de los empleados que trabajan en el departamento 20:

```
#cambio 1 atributo de la tabla en las filas que cumplen la condición WHERE
UPDATE empleados
SET salario = salario * 1.025
WHERE dep_no = 20;
```

Otro ejemplo:

```
#cambio 2 atributos (salario y comisión) de la tabla empleados, en las filas que cumplen la
condición WHERE
UPDATE empleados
SET salario = salario * 1.01, comisión = comisión + 100
WHERE dep_no = 30;
```

UPDATE no muestra ningún resultado. Si quisiera saber qué registros se van a cambiar, habría que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

```
#consulta que valores tienen las filas antes de haberlas cambiado:
SELECT *
FROM productos
WHERE Proveedor = 8 AND Familia = 3;
```

```
#realizo el cambio:
UPDATE productos
SET Precio = Precio * 1.1
WHERE Proveedor = 8 AND Familia = 3;
```

```
#consulta que valores nuevos tienen las filas tras haberlas cambiado:
SELECT *
FROM productos
WHERE Proveedor = 8 AND Familia = 3;
```

Si en una operación UPDATE suprimimos la cláusula WHERE todos los registros de la tabla señalada serán actualizados, ejemplo:

```
#se cambia en todas las filas porque no incluyo condición WHERE
UPDATE empleados
SET Salario = Salario * 1.1;
```

7.2.1 Restricción del Workbench que impide UPDATE o DELETE debido al **SAFE UPDATES** cuando no se incluye una KEY en el WHERE o no se usa LIMIT

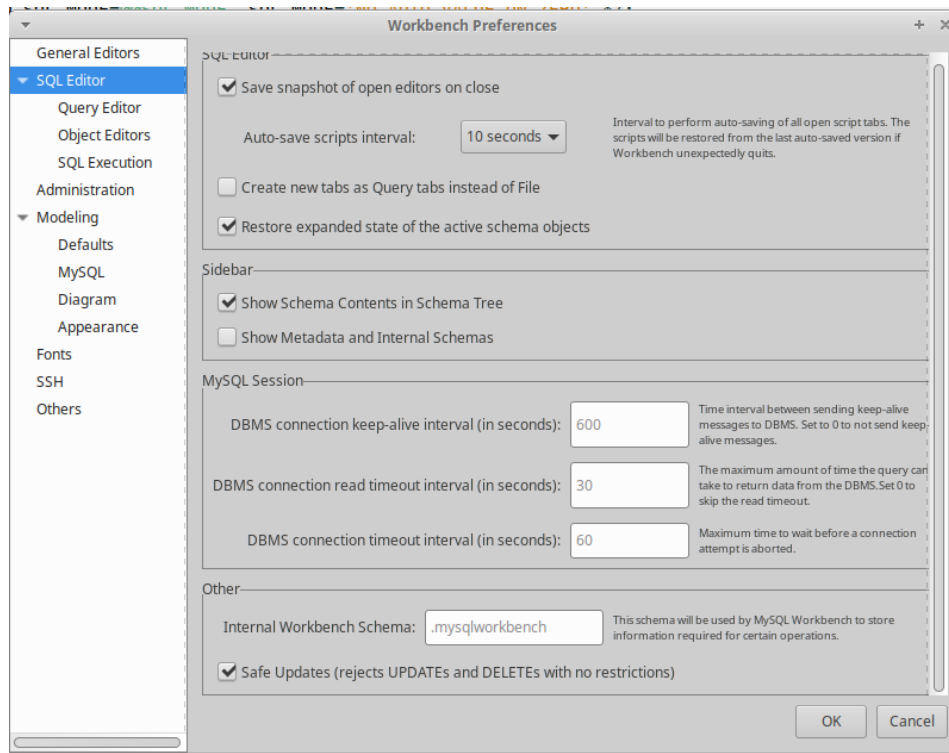
6. Por defecto Workbench tiene un comportamiento restrictivo, porque impide el UPDATE o DELETE cuando se cumplen estas 2 condiciones simultáneamente:
 - El UPDATE (DELETE) no contiene un WHERE que incluya una KEY
 - El UPDATE (DELETE) no incluye una cláusula LIMIT

El mensaje que muestra Workbench en este caso es éste:

```
You are using safe update mode and you tried to update a table without a WHERE that uses
a KEY column. To disable safe mode, toggle the option in Preferences -> SQL Queries
and reconnect.
```

7. La razón de esta restricción es evitar que un UPDATE (DELETE) pueda modificar más filas de las deseadas, eso podría suceder en caso de que no tengamos un buen dominio del lenguaje SQL y seleccionemos mal las filas a modificar. Una vez modificadas las filas, no hay manera de deshacer el cambio (salvo que hayamos iniciado una transacción y realicemos un ROLLBACK o que tengamos un BACKUP que restaurar).
8. Esta restricción es **exclusiva del Workbench**, no está presente si nos conectamos al servidor desde un terminal del S.O.
9. Podemos **configurar Workbench** para que no imponga esta restricción y permita cualquier tipo de UPDATE, esto se configura en:

```
WorkBench - Edit - Preferences - SQL Editor - "Save Updates" (Rejects UPDATES and DELETES
with no restrictions)
Hay que desmarcar esa opción.
```



10. Reiniciar Workbench para que surtan efecto los cambios.

7.3 DELETE

Crea una consulta de eliminación que elimina los registros de una o más de las tablas listadas en la cláusula FROM que satisfagan la cláusula WHERE. Esta consulta elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto (eso se haría poniendo ese campo a NULL con UPDATE). Su sintaxis es:

```
DELETE FROM tabla WHERE criterio;
```

```
DELETE FROM empleados
WHERE cargo = 'vendedor';
```

Si en una operación DELETE suprimimos la cláusula WHERE todos los registros (filas) de la tabla señalada serán borrados, ejemplo:

```
# se borran todas las filas porque no incluyo condición WHERE
DELETE FROM empleados ;
```

DELETE es especialmente útil cuando se desea eliminar varios registros. En una instrucción DELETE con múltiples tablas, debe incluir el nombre de tabla (Tabla.*). Si especifica más de una

tabla desde la que eliminar registros, todas deben ser tablas de muchos a uno. Si desea eliminar todos los registros de una tabla, **TRUNCATE** es más eficiente que ejecutar DELETE.

Se puede utilizar DELETE para eliminar registros de una única tabla o desde varios lados de una relación uno a muchos. Las operaciones de eliminación en cascada en una consulta únicamente eliminan desde varios lados de una relación. Por ejemplo, en la relación entre las tablas Clientes y Pedidos, la tabla Pedidos es la parte de muchos por lo que las operaciones en cascada solo afectaran a la tabla Pedidos. Una consulta de borrado elimina los registros completos, no únicamente los datos en campos específicos. Si desea eliminar valores en un campo especificado, crear una consulta de actualización que cambie los valores a NULL.

Una vez que se han eliminado los registros utilizando una consulta de borrado, no puede deshacer la operación. Si desea saber qué registros se eliminarán, primero examine los resultados de una consulta de selección que utilice el mismo criterio y después ejecute la consulta de borrado. Mantenga copias de seguridad de sus datos en todo momento. Si elimina los registros equivocados podrá recuperarlos desde las copias de seguridad.

7.3.1 TRUNCATE

<http://dev.mysql.com/doc/refman/5.6/en/truncate-table.html>
<http://mysql.conclase.net/curso/?sqlsen=TRUNCATE>

1. Borra todos los datos de una tabla, pero **No puede realizarse TRUNCATE sobre tablas InnoDB en las que otra tabla tiene claves ajenas que apuntan a la tabla a truncar.** (por ejemplo no puedo truncar la tabla *departamentos* porque la tabla *empleados* tienen una clave ajena que hace referencia a la tabla *departamentos*).

```
TRUNCATE TABLE table_name
```

2. TRUNCATE es similar en ciertos aspectos a esta otra sentencia:

```
DELETE FROM table_name
```

3. Pero TRUNCATE tiene algunas **diferencias frente a DELETE**:

- Realmente se considera que TRUNCATE es una sentencia DDL (Data Definition Language), pues es equivalente a una sentencia DROP TABLE seguida por una sentencia CREATE TABLE. Es más rápida que la sentencia DELETE FROM cuando queremos borrar todos los datos de una tabla.

2. Para entenderlo mejor vamos a poner un símil:

3. DELETE abre un fichero y borra línea a línea todo su contenido (imagínate si tiene miles de filas el tiempo que consume borrarlas todas) mientras que TRUNCATE borra el fichero y crea otro nuevo vacío con el mismo nombre.

- TRUNCATE no puede realizarse sobre tablas *InnoDB* en las que otra tabla tiene claves ajenas que apuntan a la tabla a truncar. Sin embargo sí se permiten FOREIGN KEY entre columnas de la propia tabla.
- No permite *ROLL BACK*
- No dispara *TRIGGERS ON DELETE*
- El valor *AUTO_INCREMENT* se reinicializa a su valor inicial.
- No se devuelve el número de filas eliminadas.
- Mientras el fichero de definición de tabla 'table_name.frm' sea válido, la tabla puede ser recreada como una tabla vacía con **TRUNCATE TABLE**, aunque los ficheros de datos o índices estén corruptos.

8. Subconsultas

EJEMPLO: *Queremos conocer los empleados cuyo salario supere el salario medio:*

- Esta sería la manera de hacerlo ejecutando dos consultas, una tras otra a mano y copiando a mano el resultado de la primera en la segunda:

```
SELECT AVG (salario) FROM empleados;
```

#La consulta anterior informa de que el salario medio es de 249.000

#Ahora utilizamos ese valor 249.000 para la consulta final:

```
SELECT * FROM empleados
WHERE salario > 249000;
```

- Esta es la manera de juntarlo todo en una consulta con subconsulta:

```
SELECT *
FROM empleados
WHERE salario >
(
  SELECT AVG (salario)
  FROM empleados
)
;
```

Una **subconsulta** es una instrucción SELECT anidada dentro de otra instrucción SELECT, SELECT...INTO, INSERT...INTO, DELETE, o UPDATE o dentro de otra subconsulta.

Puede utilizar tres formas de sintaxis para crear una subconsulta:

comparación [ANY | ALL | SOME] (instrucción sql)

expresión [NOT] IN (instrucción sql)

[NOT] EXISTS (instrucción sql)

En donde:

Comparación:

Es una expresión y un operador de comparación que compara la expresión con el resultado de la subconsulta.

expresión

Es una expresión por la que se busca el conjunto resultante de la subconsulta.

instrucción sql

Es una instrucción SELECT, que sigue el mismo formato y reglas que cualquier otra instrucción SELECT. Debe ir entre paréntesis.

Esta tabla resume las diferentes opciones que podemos usar para combinar los resultados de la subconsulta con la consulta principal:

= (1 valor)	IN (0,1,2,... valores)
<>(1valor)	NOT IN (0,1,2,... valores)
> (1 valor)	> ALL (0,1,2,... valores) <i>mayor que todos los valores del listado</i> > SOME (0,1,2,... valores) <i>mayor que algún valor del listado</i> > ANY (0,1,2,... valores) <i>mayor que algún valor del listado</i>
<,<=,>=,	se hacen de la misma forma que >

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción `SELECT` o en una cláusula `WHERE` o `HAVING`. En una subconsulta, se utiliza una instrucción `SELECT` para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula `WHERE` o `HAVING`.

Se puede utilizar el predicado `ANY` o `SOME`, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta. El ejemplo siguiente devuelve los ordenadores portátiles cuyo precio es menor que el de cualquier ordenador de sobremesa:

```
SELECT *
FROM productos
WHERE tipo = 'ordenador portátil' AND precio < ANY (
    SELECT precio
    FROM productos
    WHERE tipo = 'ordenador sobremesa'
);
```

El predicado `ALL` se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Si se cambia `ANY` por `ALL` en el ejemplo anterior, la consulta devolverá únicamente aquellos ordenadores portátiles cuyo precio es menor que el de todos los ordenadores de sobremesa. Esto es mucho más restrictivo.

```
SELECT *
FROM productos
WHERE tipo = 'ordenador portátil' AND precio < ALL (
    SELECT precio
    FROM productos
    WHERE tipo = 'ordenador sobremesa'
);
```

El predicado `IN` se emplea para recuperar únicamente aquellos registros de la consulta principal para los que existe algún registro de la subconsulta que contiene un valor igual. El ejemplo siguiente devuelve todos los desempleados que viven en localidades en los que hay ofertas de empleo:

```
SELECT *
FROM trabajadores
WHERE estado = 'desempleado' AND localidad IN (
    SELECT localidad
    FROM ofertas_de_empleo
);
```

Inversamente se puede utilizar `NOT IN` para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual. El ejemplo siguiente devuelve todos los desempleados que viven en localidades en los que no hay ofertas de empleo:

```
SELECT *
FROM trabajadores
WHERE estado = 'desempleado' AND localidad NOT IN (
    SELECT localidad
    FROM ofertas_de_empleo
);
```

El predicado **EXISTS** (con la palabra reservada `NOT` opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro. Realmente al utilizar

EXISTS estamos creando una subconsulta correlacionada. El ejemplo siguiente devuelve todos los desempleados que viven en localidades en los que hay ofertas de empleo:

```
SELECT *
FROM trabajadores
WHERE estado = 'desempleado' AND EXISTS (
    SELECT localidad
    FROM ofertas_de_empleo
    WHERE ofertas_de_empleo.localidad =
    trabajadores.localidad
);
```

Se puede utilizar también alias del nombre de la tabla en una subconsulta para referirse a tablas listadas en la cláusula FROM fuera de la subconsulta. El ejemplo siguiente devuelve los nombres de los empleados cuyo salario es igual o mayor que el salario medio de todos los empleados con el mismo título. A la tabla Empleados de la consulta externa se le ha dado el alias E1 y al interna E2:

```
SELECT Apellido, Nombre, Titulo, Salario
FROM Empleados AS E1
WHERE Salario >= (
    SELECT AVG(Salario)
    FROM Empleados AS E2
    WHERE E1.Titulo = E2.Titulo
)
ORDER BY Titulo;
```

Sería suficiente con haber dado alias a una sola tabla Empleados para distinguirlas, por lo que también valdría esta:

```
SELECT Apellido, Nombre, Titulo, Salario
FROM Empleados AS E1
WHERE Salario >= (
    SELECT AVG(Salario)
    FROM Empleados
    WHERE E1.Titulo = Empleados.Titulo
)
ORDER BY Titulo;
```

Obtiene una lista de las compañías y los contactos de todos los clientes que han realizado un pedido en el segundo trimestre de 1993.

```
SELECT Nombre, Apellidos
FROM clientes AS C
WHERE EXISTS (
    SELECT *
    FROM pedidos AS P
    WHERE P.ID_Empleado = C.ID_Empleado
);
```


En el ejemplo anterior, la palabra reservada **AS** es opcional porque las tablas tienen nombres distintos, por tanto equivale a:

```
SELECT Nombre, Apellidos
FROM clientes
WHERE EXISTS (
    SELECT *
    FROM pedidos
    WHERE pedidos.ID_Empleado = clientes.ID_Empleado
);
```

Obtiene una lista con el nombre, cargo y salario de todos los agentes de ventas cuyo salario es mayor que el de todos los jefes y directores.

```
SELECT Apellidos, Nombre, Cargo, Salario
FROM Empleados
WHERE Cargo LIKE "Agente Ven*" AND Salario > ALL (
    SELECT Salario
    FROM Empleados
    WHERE (Cargo LIKE "%Jefe%") OR (Cargo LIKE "%Director%")
);
```

Obtiene una lista con el nombre y el precio unitario de todos los aceites con el mismo precio que el 'aceite de oliva Picual Orujo 0.8'.

```
SELECT DISTINCT NombreProducto, Precio_Unidad
FROM Productos
WHERE nombre LIKE '%aceite%' AND Precio_Unidad = (
    SELECT Precio_Unidad
    FROM Productos
    WHERE nombre LIKE 'aceite de oliva Picual Orujo 0.8'
);
```

```
SELECT DISTINCT Nombre_Contacto, Nombre_Compañía, Cargo_Contacto, Telefono
FROM Clientes
WHERE ID_Cliente IN (
    SELECT DISTINCT ID_Cliente
    FROM Pedidos
    WHERE Fecha_Pedido >= '04/1/93' AND Fecha_Pedido <='07/1/93'
);
```

Selecciona el nombre de todos los empleados que han reservado al menos un pedido.

```
SELECT DISTINCT Pedidos.Id_Producto, Pedidos.Cantidad,
    (SELECT DISTINCT Productos.Nombre
    FROM Productos
    WHERE Productos.Id_Producto = Pedidos.Id_Producto) AS ElProducto
FROM Pedidos
WHERE Pedidos.Cantidad > 150
ORDER BY Pedidos.Id_Producto;
```

Recupera el Código del Producto y la Cantidad pedida de la tabla pedidos, extrayendo el nombre del producto de la tabla de productos.

```
SELECT DISTINCT pedidos.producto_no, pedidos.unidades,  
  (SELECT DISTINCT productos.descripcion  
   FROM productos  
   WHERE productos.producto_no = pedidos.producto_no) AS  
  'ElProducto'  
FROM pedidos  
WHERE pedidos.unidades > 1  
ORDER BY pedidos.producto_no;
```

9. Consultas de Unión Externas

Se utiliza la operación UNION para crear una consulta de unión, combinando los resultados de dos o más SELECT. Su sintaxis es:

```
SELECT ...  
UNION [ALL | DISTINCT] SELECT ...  
[UNION [ALL | DISTINCT] SELECT ...]
```

Puede combinar los resultados de dos o más consultas SELECT, en cualquier orden, en una única operación UNION. El ejemplo siguiente combina a los clientes de la tablas clientes_actuales y clientes_datos_de_baja:

```
SELECT * FROM clientes_actuales WHERE LOCALIDAD = 'LUGO'  
UNION  
SELECT * FROM clientes_datos_de_baja WHERE LOCALIDAD = 'LUGO';
```

UNION: No se devuelven filas duplicadas.

UNION ALL: Sí se devuelven filas duplicadas.

Todas las consultas en una operación UNION deben pedir el mismo número de campos, no obstante los campos no tienen por qué tener el mismo tamaño o el mismo tipo de datos.

Se puede utilizar una cláusula GROUP BY y/o HAVING en cada consulta para agrupar los datos devueltos.

Puede utilizar una cláusula ORDER BY al final del último argumento consulta para visualizar los datos devueltos en un orden específico.

```
SELECT [Nombre de compañía], Ciudad  
FROM Proveedores  
WHERE País = 'Brasil'  
UNION  
SELECT [Nombre de compañía], Ciudad  
FROM Clientes  
WHERE País = 'Brasil';  
#Recupera los nombres y las ciudades de todos proveedores y clientes de Brasil
```

```
SELECT [Nombre de compañía], Ciudad  
FROM Proveedores  
WHERE País = 'Brasil'  
UNION  
SELECT [Nombre de compañía], Ciudad  
FROM Clientes  
WHERE País = 'Brasil'  
ORDER BY Ciudad;
```

#Recupera los nombres y las ciudades de todos proveedores y clientes radicados en Brasil, ordenados por el nombre de la ciudad

```
SELECT [Nombre de compañía], Ciudad  
FROM Proveedores
```

```
WHERE País = 'Brasil'
UNION
SELECT [Nombre de compañía], Ciudad
FROM Clientes
WHERE País = 'Brasil'
UNION
SELECT [Apellidos], Ciudad
FROM Empleados
WHERE Región = 'América del Sur';
#Recupera los nombres y las ciudades de todos los proveedores y clientes de brasil y los apellidos y
las ciudades de todos los empleados de América del Sur
```

10. INNER JOIN, NATURAL JOIN, LEF JOIN, RIGHT JOIN, FULL JOIN ...

■ Introducción

1. En este documento vamos a ver las distintas maneras que existen para juntar 2 o más tablas.
2. Hasta ahora siempre hemos utilizado INNER JOIN, y es la manera más habitual de hacerlo, Porque siempre se puede utilizar.
3. Existe otra opción que consigue el mismo resultado que INNER JOIN y es el NATURAL JOIN pero tiene el inconveniente de que no siempre se puede utilizar. Veremos cómo funciona y cuando se puede utilizar.
4. Finalmente veremos otro tipo de uniones entre tablas: LEF JOIN, RIGHT JOIN, FULL JOIN... Estos tipos de uniones permiten conseguir resultados distintos a los que se consiguen al juntar 2 tablas con INNER JOIN.

10.1 INNER JOIN: juntar 2 tablas para tareas normales

<http://stackoverflow.com/questions/8696383/difference-between-natural-join-and-inner-join>
<http://dev.mysql.com/doc/refman/5.7/en/join.html>

Código de campo cambiado

INNER JOIN es el comando que hemos utilizado hasta ahora para reunir 2 tablas. Es el comando más utilizado

Existe la alternativa de utilizar NATURAL JOIN (la veremos), pero no siempre se puede utilizar.

Hay 4 formas equivalentes de realizar INNER JOIN:

```
SELECT apellido1, empleados.dep_no, localidad
FROM empleados INNER JOIN departamentos
ON empleados.dep_no = departamentos.dep_no
;

SELECT apellido1, empleados.dep_no, localidad
FROM empleados INNER JOIN departamentos
USING (dep_no)
;

SELECT apellido1, empleados.dep_no, localidad
FROM empleados NATURAL JOIN departamentos
;

SELECT apellido1, empleados.dep_no, localidad
FROM empleados, departamentos
WHERE empleados.dep_no = departamentos.dep_no
;
```

In MySQL, JOIN, CROSS JOIN, and INNER JOIN are syntactic equivalents (they can replace each other).

In standard SQL, they are not equivalent. INNER JOIN is used with an ON clause, CROSS JOIN is used otherwise.

10.2 NATURAL JOIN

<https://www.tutorialesprogramacionya.com/mysqlya/temarios/descripcion.php?inicio=63&cod=61&punto=67>

Ejemplo:

```
SELECT apellido, empleados.dep_no, localidad
FROM empleados NATURAL JOIN departamentos
;
```

El comando **NATURAL JOIN**:

- también permite juntar 2 tablas, consigue el mismo resultado que el INNER JOIN
- Pero sólo se puede utilizar cuando las tablas que queremos reunir cumplen ciertas condiciones que veremos (INNER JOIN se puede utilizar siempre).
- necesitamos escribir menos código.

Con NATURAL JOIN podemos ahorrarnos escribir la cláusula ON (que sí requiere INNER JOIN), pero sólo se puede utilizar cuando los campos por los cuales se juntan las tablas tienen el mismo nombre. El NATURAL JOIN funciona así:

- El servidor buscará que atributos tienen el mismo nombre en ambas tablas y el escribirá por nosotros la cláusula ON.
- Si las 2 tablas no tienen atributos que se llamen igual no puede juntarlas y no funcionará.
- Si las 2 tablas tienen atributos que se llamen igual pero no sirven para unir las, el intentará unir las con esos atributos equivocados y no funcionará.

Por tanto sí podríamos juntar las tablas:

```
empleados - departamentos
pedidos - clientes
pedidos - productos
```

Pero no podríamos juntar estas tablas porque no tienen atributos que se llamen igual:

```
clientes - empleados
```

10.2.1 Otras opciones (menos interesantes) para realizar INNER JOIN:

además de las ya vistas, existen otras dos opciones para juntar dos tablas:

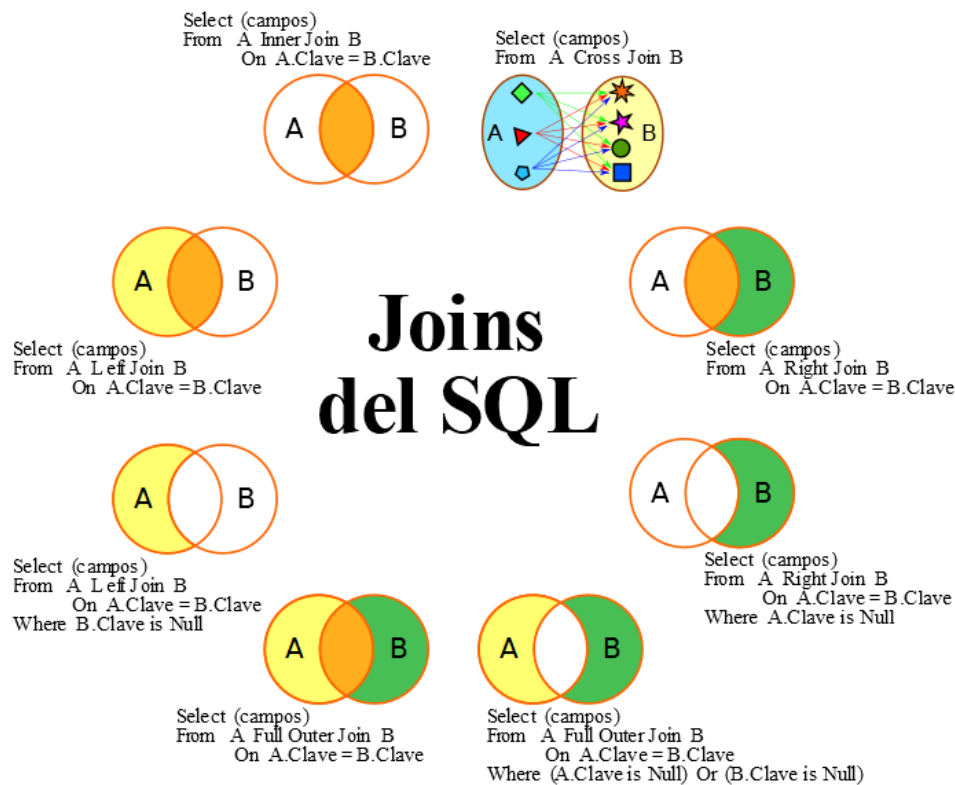
```
SELECT apellido, empleados.dep_no, localidad
FROM empleados INNER JOIN departamentos
    USING (dep_no)
;

SELECT apellido, empleados.dep_no, localidad
FROM empleados, departamentos
WHERE empleados.dep_no = departamentos.dep_no
;
```

10.3 Otros tipos de JOIN: CROSS JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN

<http://stackoverflow.com/questions/448023/what-is-the-difference-between-left-right-outer-and-inner-joins>
<http://dev.mysql.com/doc/refman/5.7/en/join.html>

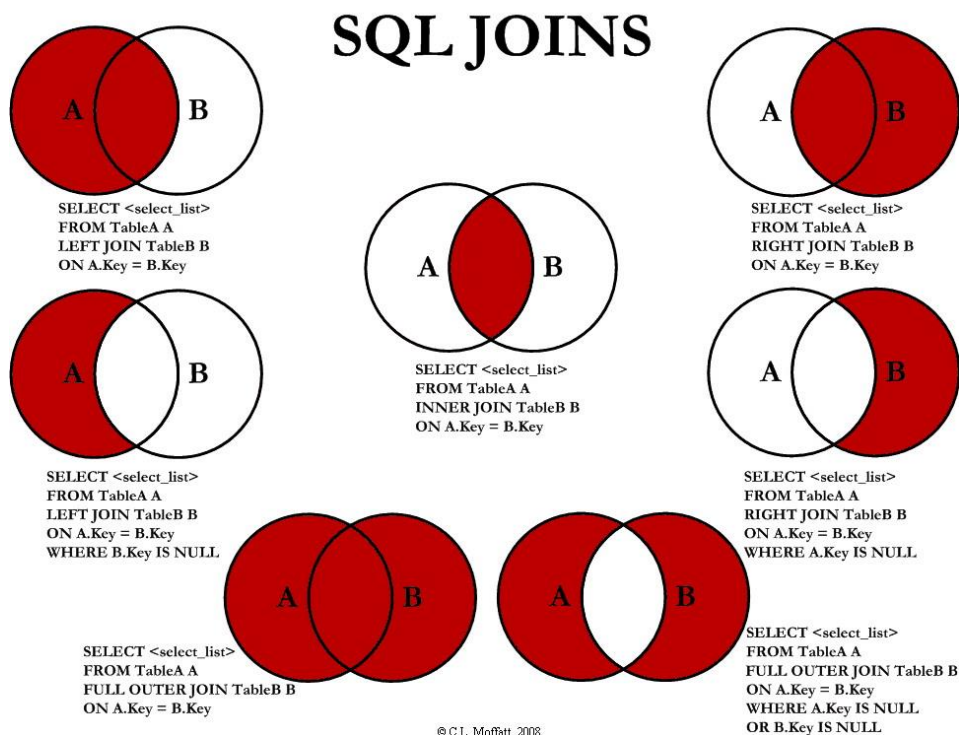
Código de campo cambiado



https://es.wikipedia.org/wiki/Sentencia_JOIN_en_SQL

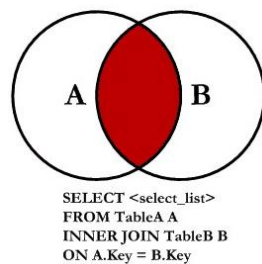
El siguiente diagrama muestra mediante teoría de conjuntos Cuál es el resultado de unir 2 tablas A y B mediante cada uno de los tipos de JOIN existentes:

- cada diagrama muestra debajo la sentencia JOIN que se ha utilizado.
- En color rojo se muestran que datos de ambas tablas sobreviven a la unión (JOIN).



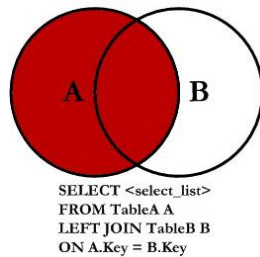
■ **Explicación previa de algunos tipos de JOIN del diagrama:**

- **INNER JOIN:** Comienza por entender el caso **INNER JOIN**: al juntar 2 tablas A y B, resulta una supertabla que contiene sólo aquellos elementos de la tabla A que se casan con elementos de la tabla B:



- **LEFT JOIN:** el caso **Tabla_A LEFT JOIN Tabla_B**: al juntar 2 tablas A y B, resulta una supertabla que contiene:

- Aquellos elementos de la tabla A que se casan con elementos de la tabla B
- Aquellos elementos de la tabla que aparece a la izquierda (tabla_A) que no se han podido casar con elementos de la tabla de la derecha (tabla_B):



- RIGHT JOIN: El caso `Tabla_A RIGHT JOIN Tabla_B` es similar al LEFT pero en lugar de utilizarlo hacia la izquierda lo hace hacia la derecha.

Ejemplos:

#Mostrar a todos los empleados:

```
SELECT emp_no, dep_no FROM empleados;
```

#Mostrar a todos los empleados, al lado de cada empleado mostrar los datos de su departamento. Cuando un empleado no esté asignado a algún departamento NO se quiere mostrar al empleado:

```
SELECT emp_no, dep_no, dnombre FROM empleados INNER JOIN departamentos ON
empleados.dep_no = departamentos.dep_no;
```

#Mostrar a todos los empleados, al lado de cada empleado mostrar los datos de su departamento. Cuando un empleado no esté asignado a algún departamento Si se quiere mostrar al empleado (lógicamente en ese caso los datos del departamento van a aparecer en blanco):

(2 SOLUCIONES EQUIVALENTES)

```
SELECT emp_no, empleados.dep_no, dnombre FROM empleados LEFT JOIN departamentos ON
empleados.dep_no = departamentos.dep_no;
```

```
SELECT emp_no, empleados.dep_no, dnombre FROM departamentos RIGHT JOIN empleados ON
empleados.dep_no = departamentos.dep_no;
```

tabla empleados	SELECT emleados INNER JOIN departamentos	SELECT empleados LEFT JOIN departamentos																																																																																																																																							
<table><tr><th>#</th><th>emp_no</th><th>dep_no</th></tr><tr><td>1</td><td>7499</td><td>30</td></tr><tr><td>2</td><td>7521</td><td>10</td></tr><tr><td>3</td><td>7654</td><td>30</td></tr><tr><td>4</td><td>7698</td><td>30</td></tr><tr><td>5</td><td>7782</td><td>10</td></tr><tr><td>6</td><td>7839</td><td>10</td></tr><tr><td>7</td><td>7844</td><td>30</td></tr><tr><td>8</td><td>7876</td><td>20</td></tr><tr><td>9</td><td>7900</td><td>20</td></tr><tr><td>10</td><td>8898</td><td>30</td></tr><tr><td>11</td><td>8902</td><td>NULL</td></tr><tr><td>12</td><td>8904</td><td>NULL</td></tr></table>	#	emp_no	dep_no	1	7499	30	2	7521	10	3	7654	30	4	7698	30	5	7782	10	6	7839	10	7	7844	30	8	7876	20	9	7900	20	10	8898	30	11	8902	NULL	12	8904	NULL	<table><tr><th>#</th><th>emp_no</th><th>dep_no</th><th>nombre</th></tr><tr><td>1</td><td>7499</td><td>30</td><td>VENTAS</td></tr><tr><td>2</td><td>7521</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>3</td><td>7654</td><td>30</td><td>VENTAS</td></tr><tr><td>4</td><td>7698</td><td>30</td><td>VENTAS</td></tr><tr><td>5</td><td>7782</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>6</td><td>7839</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>7</td><td>7844</td><td>30</td><td>VENTAS</td></tr><tr><td>8</td><td>7876</td><td>20</td><td>INVESTIGACION</td></tr><tr><td>9</td><td>7900</td><td>20</td><td>INVESTIGACION</td></tr><tr><td>10</td><td>8898</td><td>30</td><td>VENTAS</td></tr></table>	#	emp_no	dep_no	nombre	1	7499	30	VENTAS	2	7521	10	CONTABILIDAD	3	7654	30	VENTAS	4	7698	30	VENTAS	5	7782	10	CONTABILIDAD	6	7839	10	CONTABILIDAD	7	7844	30	VENTAS	8	7876	20	INVESTIGACION	9	7900	20	INVESTIGACION	10	8898	30	VENTAS	<table><tr><th>#</th><th>emp_no</th><th>dep_no</th><th>nombre</th></tr><tr><td>1</td><td>7499</td><td>30</td><td>VENTAS</td></tr><tr><td>2</td><td>7521</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>3</td><td>7654</td><td>30</td><td>VENTAS</td></tr><tr><td>4</td><td>7698</td><td>30</td><td>VENTAS</td></tr><tr><td>5</td><td>7782</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>6</td><td>7839</td><td>10</td><td>CONTABILIDAD</td></tr><tr><td>7</td><td>7844</td><td>30</td><td>VENTAS</td></tr><tr><td>8</td><td>7876</td><td>20</td><td>INVESTIGACION</td></tr><tr><td>9</td><td>7900</td><td>20</td><td>INVESTIGACION</td></tr><tr><td>10</td><td>8898</td><td>30</td><td>VENTAS</td></tr><tr><td>11</td><td>8902</td><td>NULL</td><td>NULL</td></tr><tr><td>12</td><td>8904</td><td>NULL</td><td>NULL</td></tr></table>	#	emp_no	dep_no	nombre	1	7499	30	VENTAS	2	7521	10	CONTABILIDAD	3	7654	30	VENTAS	4	7698	30	VENTAS	5	7782	10	CONTABILIDAD	6	7839	10	CONTABILIDAD	7	7844	30	VENTAS	8	7876	20	INVESTIGACION	9	7900	20	INVESTIGACION	10	8898	30	VENTAS	11	8902	NULL	NULL	12	8904	NULL	NULL
#	emp_no	dep_no																																																																																																																																							
1	7499	30																																																																																																																																							
2	7521	10																																																																																																																																							
3	7654	30																																																																																																																																							
4	7698	30																																																																																																																																							
5	7782	10																																																																																																																																							
6	7839	10																																																																																																																																							
7	7844	30																																																																																																																																							
8	7876	20																																																																																																																																							
9	7900	20																																																																																																																																							
10	8898	30																																																																																																																																							
11	8902	NULL																																																																																																																																							
12	8904	NULL																																																																																																																																							
#	emp_no	dep_no	nombre																																																																																																																																						
1	7499	30	VENTAS																																																																																																																																						
2	7521	10	CONTABILIDAD																																																																																																																																						
3	7654	30	VENTAS																																																																																																																																						
4	7698	30	VENTAS																																																																																																																																						
5	7782	10	CONTABILIDAD																																																																																																																																						
6	7839	10	CONTABILIDAD																																																																																																																																						
7	7844	30	VENTAS																																																																																																																																						
8	7876	20	INVESTIGACION																																																																																																																																						
9	7900	20	INVESTIGACION																																																																																																																																						
10	8898	30	VENTAS																																																																																																																																						
#	emp_no	dep_no	nombre																																																																																																																																						
1	7499	30	VENTAS																																																																																																																																						
2	7521	10	CONTABILIDAD																																																																																																																																						
3	7654	30	VENTAS																																																																																																																																						
4	7698	30	VENTAS																																																																																																																																						
5	7782	10	CONTABILIDAD																																																																																																																																						
6	7839	10	CONTABILIDAD																																																																																																																																						
7	7844	30	VENTAS																																																																																																																																						
8	7876	20	INVESTIGACION																																																																																																																																						
9	7900	20	INVESTIGACION																																																																																																																																						
10	8898	30	VENTAS																																																																																																																																						
11	8902	NULL	NULL																																																																																																																																						
12	8904	NULL	NULL																																																																																																																																						

10.4 Tipos de JOIN

- **INNER JOIN** junta dos tablas A y B, sólo devuelve las filas para las que existe una coincidencia en los atributos de reunión.
- **OUTER JOIN** compara dos tablas A y B, devuelve las filas para las que existe una coincidencia en los atributos de reunión y además añade otras filas: cuando una fila de una tabla A no se corresponde con ninguna fila en la otra tabla B, se devuelve esa fila de A y en las columnas de B aparece NULL. Lo mismo sucede cuando una fila de la tabla B no se corresponde con ninguna fila en la otra tabla A, se devuelve esa fila de B y en las columnas de A aparece NULL. Existen **3 tipos de OUTER JOIN**:
 - **LEFT JOIN (equivale a LEFT OUTER JOIN)**: Devuelve todos los registros de la tabla A y NULL en los atributos de la tabla B si no tiene coincidencias
 - **RIGHT JOIN (equivale a RIGHT OUTER JOIN)**: Devuelve todos los registros de la tabla B y NULL en los atributos de la tabla así no tiene coincidencias
 - **FULL JOIN no existe en MySQL (equivale a FULL OUTER JOIN)**: Devuelve todos los registros de ambas tablas, e insertar un valor NULL en cualquiera de las tablas si no hay ninguna coincidencia. Equivale a combinar LEFT y RIGHT
- **CROSS JOIN** compara dos tablas y devuelve todas las combinaciones posibles de filas de ambas tablas. Se obtiene una gran cantidad de filas que no tienen sentido. **Tiene poca utilidad.** Equivale a producto cartesiano de matemáticas.

4. En MySQL no existe FULL JOIN, en su lugar debemos usar un truco: unir el resultado de un LEFT JOIN con el resultado de un RIGHT JOIN mediante el comando UNION. Se muestra ejemplo un poco más abajo

5. En MySQL, JOIN, CROSS JOIN, e INNER JOIN son equivalentes sintácticamente (se pueden sustituir unos por otros). En SQL estándar, esos comandos no son equivalentes: INNER JOIN se utiliza obligatoriamente con una cláusula ON y CROSS JOIN se

utiliza	en	el	resto	de	casos.
Http://dev.mysql.com/doc/refman/5.6/en/join.html					

- A menudo se omite la palabra clave **OUTER**, de este modo, por ejemplo, estos comandos serían equivalentes: "LEFT JOIN", "LEFT OUTER JOIN".

```
# ejemplo de cómo realizar algo equivalente al FULL OUTER JOIN en MySQL
# Combinamos el resultado de un LEFT y un RIGHT

SELECT *
FROM empleados LEFT JOIN departamentos ON empleados.dep_no = departamentos.dep_no
UNION
SELECT *
FROM empleados RIGHT JOIN departamentos ON empleados.dep_no = departamentos.dep_no
;
```

#UNION: une el resultado de 2 consultas sin mostrar filas duplicadas
 #UNION ALL: une el resultado de 2 consultas (incluye filas duplicadas)

10.5 Ejemplos de distintos tipos de JOIN:

Analizaremos un caso práctico:

DESCRIPCIÓN DEL PROBLEMA:

- 1) Se trata de un instituto con alumnos (alumnos) que pueden usar taquillas (taquillas):
- 2) Cada estudiante puede ser asignado a una taquilla, por lo que hay una clave ajena Numero_Taquilla en la Tabla de alumnos.
- 3) Un alumno nunca tiene más de una taquilla.
- 4) Una taquilla podría estar asignada a más de un estudiante (comparten la taquilla)
- 5) Es posible que algún estudiante no tenga taquilla.
- 6) Puede haber taquillas que no tienen estudiantes asignados.

Ejemplo, digamos que: hay 100 estudiantes, 70 de los cuales tienen taquillas. Se dispone de un total de 50 taquillas, 40 de las cuales con al menos 1 estudiante y 10 taquillas no usa ningún estudiante.

2) INNER JOIN

```
SELECT *
FROM alumnos INNER JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
```

responde a la pregunta "muéstrame todos los estudiantes con armarios junto con los datos de su armario".
 Cualquier estudiante sin armarios, o cualquier taquilla sin estudiantes desaparecen.
 Devuelve 70 filas

3) LEFT JOIN (LEFT OUTER JOIN)

```
SELECT *
FROM alumnos LEFT JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
```

responde a la pregunta:
 "muéstrame todos los estudiantes, con sus correspondientes armarios; muestra a todos los alumnos, aunque no tengan armario".
 Esto podría ser una lista general de los estudiantes, o podría ser utilizado para identificar a los estudiantes sin casillero.
 Devuelve 100 filas

4) RIGHT JOIN (RIGHT OUTER JOIN)

```
SELECT *
FROM alumnos RIGHT JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
```

Responde a la pregunta:

"muéstrame todas las taquillas (tengan o no estudiantes asignados), al lado de cada taquilla muestra los datos de los estudiantes asignados a la taquilla, y si no tiene ningún estudiante asignado muestra los atributos del estudiante como NULL".

Esto podría ser utilizado para conocer los armarios que tienen alumnos junto con los armarios sin alumnos.

Devuelve 80 filas (lista de 70 estudiantes en los casilleros, además de los 10 casilleros con ningún estudiante)

5) FULL JOIN (FULL OUTER JOIN)

```
SELECT *
FROM alumnos FULL JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
```

#Como en MYSQL no existe FULL JOIN, lo solucionamos así:

```
SELECT *
FROM alumnos LEFT JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
UNION ALL
SELECT *
FROM alumnos RIGHT JOIN taquillas ON alumno.numero_taquilla = taquilla.numero_taquilla
```

Responde a la pregunta:

"muéstrame todos los estudiantes y todos los armarios".

Devuelve 110 filas (100 todos los estudiantes, incluyendo los que no tienen taquillas, además las 10 taquillas sin estudiante).

CROSS JOIN es absurdo en este escenario.

Muestra una lista gigante de toda posible vinculación entre estudiantes y el armario, aunque no sea real.

Devuelve 5000 filas (100 x 50 filas: 100 estudiantes x 50 taquillas).

11. CREATE TABLE

```

DROP TABLE IF EXISTS departamentos;

CREATE TABLE IF NOT EXISTS departamentos (
  DEP_NO SMALLINT UNSIGNED NOT NULL,
  DNOMBRE VARCHAR(65) NOT NULL,
  LOCALIDAD VARCHAR(65),
  PRIMARY KEY (DEP_NO)
);

DROP TABLE IF EXISTS empleados;

CREATE TABLE IF NOT EXISTS empleados (
  EMP_NO SMALLINT UNSIGNED NOT NULL,
  DNI VARCHAR(9) NOT NULL,
  NOMBRE VARCHAR(65) NOT NULL,
  APELLIDO1 VARCHAR(65) NOT NULL,
  APELLIDO2 VARCHAR(65),
  OFICIO VARCHAR(65),
  JEFE SMALLINT UNSIGNED,
  FECHA_ALTA DATE,
  SALARIO DECIMAL(8,2) UNSIGNED,
  COMISION DECIMAL(8,2) UNSIGNED,
  DEP_NO SMALLINT UNSIGNED,
  TELEFONO VARCHAR(15),
  FOTO VARCHAR(255),
  PRIMARY KEY (EMP_NO),
  UNIQUE UQ_EMP_DNI (DNI),
  UNIQUE UQ_EMP_FOTO (FOTO),
  CONSTRAINT FK_EMP_DEP_NO
    FOREIGN KEY (DEP_NO)
      REFERENCES departamentos (DEP_NO)
      ON UPDATE CASCADE,
  CONSTRAINT FK_EMP_JEFE
    FOREIGN KEY (JEFE)
      REFERENCES empleados (EMP_NO)
      ON DELETE SET NULL
      ON UPDATE CASCADE
);

```

11.1 Índices: PRIMARY KEY, UNIQUE, INDEX

11.1.1 Orden de Colocación en CREATE TABLE de los distintos índices:

Al crear la tabla no es necesario seguir un orden en la creación de los índices, pero lo habitual es seguir este orden para facilitar la lectura:

- Primero se coloca la restricción **PRIMARY KEY**
- Después se colocan los índices **UNIQUE**
- Por último, se colocan los índices **INDEX**

6. This helps the MySQL optimizer to prioritize which index to use and also more quickly to detect duplicated UNIQUE keys.

11.1.2 PRIMARY KEY

Es importante elegir bien la clave primaria:

- Debemos elegir un campo cuyos valores no vaya a sufrir modificaciones, ya que será necesario una reordenación de los registros cada vez que cambiamos un valor.
- Además es mejor que tenga que el atributo tenga un tamaño pequeño ya que si, por ejemplo, es una cadena de 100 caracteres, estamos guardando en todos los índices secundarios 100 caracteres en cada entrada del índice.

11.1.2.1 3 maneras distintas de definir la PRIMARY KEY:

```
CREATE TABLE actor (  
  actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
  first_name VARCHAR(45) NOT NULL,  
  last_name VARCHAR(45) NOT NULL,  
  PRIMARY KEY (actor_id)  
);
```

```
CREATE TABLE actor (  
  actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  first_name VARCHAR(45) NOT NULL,  
  last_name VARCHAR(45) NOT NULL  
);
```

```
CREATE TABLE actor (  
  actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
  first_name VARCHAR(45) NOT NULL,  
  last_name VARCHAR(45) NOT NULL,  
  CONSTRAINT pk_actor_id PRIMARY KEY (actor_id)  
);
```

- Una tabla puede tener **ninguna o una sola** PRIMARY KEY.
- **PRIMARY KEY** lleva implícita la restricción **UNIQUE**.
- **PRIMARY KEY** lleva implícita la restricción **NOT NULL**, pero **deberíamos escribir explícitamente el NOT NULL** para evitar inconvenientes, **por los siguientes motivos relacionados con el valor por defecto de un atributo**:

- **Valor por defecto de un atributo**

<http://dev.mysql.com/doc/refman/5.5/en/data-type-defaults.html>

- Si insertamos una fila y no damos valor al atributo PRIMARY KEY, entonces MySQL **SÍ** le asigna al atributo PRIMARY KEY el **valor por defecto propio de su tipo de datos**. Evidentemente si ese valor por defecto ya existiese previamente en otro registro, no podrá introducirse la fila (porque PRIMARY KEY no permite valores duplicados de la clave).

Veamos en un ejemplo, por qué debemos poner explícitamente NOT NULL a un atributo que forme parte de la PRIMARY KEY:

#El comando **SHOW CREATE TABLE** muestra la estructura real de la tabla (con las cláusulas que añade MySQL de forma implícita).

-- Ejemplo 1: tabla creada correctamente (especificando NOT NULL en el atributo que es PRIMARY KEY):

-- Si hemos especificado NOT NULL en el atributo de la PRIMARY KEY

```
CREATE TABLE t (i INT NOT NULL,
PRIMARY KEY (i))
;
```

SHOW CREATE TABLE t;

```
CREATE TABLE `t` (
  `i` int(11) NOT NULL,
  PRIMARY KEY (`i`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

-- Ejemplo 2: la misma tabla creada INCORRECTAMENTE (NO especificando NOT NULL en el atributo que es PRIMARY KEY):

-- NO hemos especificado NOT NULL en el atributo de la PRIMARY KEY

```
CREATE TABLE t (i INT,
PRIMARY KEY (i))
;
```

SHOW CREATE TABLE t;

```
CREATE TABLE `t` (
  `i` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`i`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

11.1.3 UNIQUE [INDEX|KEY]

http://www.w3schools.com/sql/sql_unique.asp

- UNIQUE Crea una restricción que impide que el atributo (o atributos) se repita en varias filas.
- UNIQUE **Sí** permite que el atributo sea NULL en varias filas (en caso de que no exista una restricción NOT NULL que impida los nulos)
- La restricción PRIMARY KEY lleva implícita la restricción UNIQUE y NOT NULL.
- Se pueden tener varias restricciones UNIQUE en una tabla pero una sola restricción de tipo PRIMARY KEY
- Si introducimos un registro y no damos valor al atributo UNIQUE, MySQL **NO** le asigna al atributo UNIQUE el valor por defecto propio de su tipo de datos (a diferencia de lo que sucede con PRIMARY KEY).

Comandos para añadir restricción UNIQUE, tenemos varias posibilidades:

SQL Server / Oracle / MS Access (también MySQL):

```
CREATE TABLE Persons
(
  P_Id int NOT NULL UNIQUE,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255)
```

)

MySQL:

```
CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  UNIQUE (P_Id)
)
```

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  CONSTRAINT uc_PersonID UNIQUE (P_Id,LastName)
)
```

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (P_Id)
```

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT uc_PersonID UNIQUE (P_Id,LastName)
```

MySQL:

```
ALTER TABLE Persons
DROP INDEX uc_PersonID
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT uc_PersonID
```

MySQL:

```
-- crear un índice sobre el atributo nombre
CREATE INDEX index_nombre ON clientes (nombre);

--crear un índice sobre el atributo nombre, pero solamente utilizar los 10 primeros
caracteres del nombre para formar el índice
CREATE INDEX index_nombre ON clientes (nombre(10));
```

11.1.4 INDEX|KEY

<http://www.dimensis.com/consejos-1.html>
<http://stackoverflow.com/questions/3844899/difference-between-key-primary-key-unique-key-and-index-in-mysql>

Código de campo cambiado

Código de campo cambiado

<https://dev.mysql.com/doc/refman/5.6/en/optimization-indexes.html>

Código de campo cambiado

7. KEY e INDEX son sinónimos.

5. Los índices (INDEX) son usados para encontrar rápidamente los registros que tengan un determinado valor en alguna de sus columnas. Sin un índice, MySQL tiene que iniciar con el primer registro y leer a través de toda la tabla para encontrar los registros relevantes. Aún en tablas pequeñas, de unos 1000 registros, es por lo menos 100 veces más rápido leer los datos usando un índice, que haciendo una lectura secuencial.
6. Cuando indexamos una columna en particular, MySQL crea otra estructura de datos (un índice) que usa para almacenar información extra acerca de los valores en la columna indexada.
7. MySQL almacena todas las claves del índice en una estructura de datos de árbol. Esta estructura de datos de árbol le permite a MySQL encontrar claves muy rápidamente.
8. Cuando MySQL encuentre que hay un índice en una columna, lo usará en vez de hacer un escaneo completo de la tabla. Esto reduce de manera importante los tiempos de CPU y las operaciones de entrada/salida en disco, a su vez que se mejora la concurrencia porque MySQL bloqueará la tabla únicamente para obtener las filas que necesite (en base a lo que encontró en el índice).
9. Cuando tenemos grandes cantidades de datos en nuestras tablas, la mejora en la obtención de los datos puede ser muy significativa.
10. A PRIMARY KEY is a unique index where all key columns must be defined as NOT NULL. If they are not explicitly declared as NOT NULL, MySQL declares them so implicitly (and silently). A table can have only one PRIMARY KEY. The name of a PRIMARY KEY is always PRIMARY, which thus cannot be used as the name for any other kind of index.
11. a UNIQUE index permits multiple NULL values for columns that can contain NULL.
12. Un índice puede incluir hasta 16 atributos.
13. MySQL can use multiple-column indexes for queries that test all the columns in the index, or queries that test just the first column, the first two columns, the first three columns, and so on. If you specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

<https://dev.mysql.com/doc/refman/5.6/en/multiple-column-indexes.html>

Código de campo cambiado

Suppose that a table has the following specification:

```
CREATE TABLE test (  
  id          INT NOT NULL,  
  last_name   CHAR(30) NOT NULL,  
  first_name  CHAR(30) NOT NULL,  
  PRIMARY KEY (id),  
  INDEX name (last_name,first_name)  
);
```

The name index is an index over the last_name and first_name columns. The index can be used for lookups in queries that specify values in a known range for combinations of last_name and first_name values. It can also be used for queries that specify just a last_name value because that column is a leftmost prefix of the index (as described later in this section).

Therefore, the name index is used for lookups in the following queries:

```
SELECT * FROM test WHERE last_name='Widenius';
```

```
SELECT * FROM test  
WHERE last_name='Widenius' AND first_name='Michael';
```

```
SELECT * FROM test
WHERE last_name='Widenius'
AND (first_name='Michael' OR first_name='Monty');
```

```
SELECT * FROM test
WHERE last_name='Widenius'
AND first_name >='M' AND first_name < 'N';
```

However, the name index is not used for lookups in the following queries:

```
SELECT * FROM test WHERE first_name='Michael';
```

```
SELECT * FROM test
WHERE last_name='Widenius' OR first_name='Michael'
```

14. Cuando se crea un índice para un atributo BLOB o TEXT, es necesario especificar cuantos caracteres iniciales del atributo se utilizan para crear el índice (máximo 676 bytes). Ejemplo:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

8. Prefix limits are measured in bytes, while the prefix length in **CREATE TABLE** statements is interpreted as number of characters. Take this into account when specifying a prefix length for a column that uses a multibyte character set.

6) Ventajas:

- **Concurrencia:** mientras MySQL está leyendo datos de una tabla, esta queda bloqueada para escritura, aunque sí permite lecturas concurrentes. Si la consulta es más rápida entonces el bloqueo es más breve.
- **Disco duro:** cuantos menos datos se lean del disco duro menos cargamos el acceso al HD y puede hacer más tareas.
- **CPU:** analizar cada uno de los registros de una tabla da más trabajo que consultar el atributo de un índice. Cuanto menos cargamos la CPU más tareas podrá realizar servidor.

7) Inconvenientes

A pesar de que los índices aceleran la consulta (SELECT) de datos, tienen como inconveniente el que **ralentizan las operaciones de inserción, borrado y modificación** de datos:

Cada vez que insertamos, borramos un registro de la tabla, el SGBD debe modificar también todos los índices de esta tabla.

Si modificamos un atributo de un registro y ese atributo forma parte de un índice, el SGBD debe modificar ese índice.

8) En qué atributos crear índices:

- atributos utilizados en **JOIN, WHERE, ORDER BY**

9) Ejemplos:

```
CREATE TABLE IF NOT EXISTS autobus (
matricula VARCHAR(9) NOT NULL,
cif_empresa_propietaria VARCHAR(9) NOT NULL,
dni_conductor VARCHAR(9),
PRIMARY KEY (matricula),
```

```
INDEX indice_conductor (dni_conductor)  
);
```

11.2 FOREIGN KEY

11.2.1 ON DELETE/UPDATE [RESTRICT | CASCADE | SET NULL | NO ACTION]

<http://dev.mysql.com/doc/refman/5.6/en/create-table-foreign-keys.html>

Código de campo cambiado

- **CASCADE:** Cuando se Borra o actualiza un registro en la tabla padre, automáticamente se elimina o actualizan las filas coincidentes en la tabla de la clave ajena. `ON DELETE CASCADE` y `ON UPDATE CASCADE` son compatibles.

9. Actualmente, CASCADE no activa TRIGGERS

- **SET NULL:** Cuando se Borra o actualiza un registro en la tabla padre, se establece la columna de clave ajena a NULL. `ON DELETE SET NULL` y `ON UPDATE SET NULL` son compatibles.

10. CUIDADO: Si se especifica un **SET NULL** en la clave ajena, entonces no se puede usar la restricción **NOT NULL** en los atributos de esa clave ajena.

- **RESTRICT:** Rechaza la operación de eliminación o actualización en la tabla padre cuando existe un registro relacionado en la tabla secundaria. Especificar `RESTRICT` (o `NO ACTION`) es lo mismo que omitir `DELETE` u `ON UPDATE`. Es preferible usar **NO ACTION** porque Workbench 6.2 y 6.3 no muestran correctamente la restricción en el diagrama gráfico si hacemos ingeniería inversa sobre una BD.
- **NO ACTION:** Una palabra clave de SQL estándar. En MySQL, equivalente a `RESTRICT`.

11. Cuando no se especifica `ON DELETE` o `ON UPDATE`, la acción predeterminada es **RESTRICT**.

11.2.2 Errores típicos al crear una FOREIGN KEY

Ejemplo de FOREIGN KEY correctas:

```
# Foreign Key formada por 1 atributo:
CONSTRAINT FK_CAMA_NUMAND FOREIGN KEY (Numero_Andar) REFERENCES Andar (Numero) ON UPDATE
CASCADE ON DELETE NO ACTION

# Foreign Key formada por 2 atributos:
CONSTRAINT fk_doente_numcama_numplanta FOREIGN KEY (NUMERO_CAMA, NUMERO_PLANTA)
REFERENCES CAMAS (NUMERO_CAMA, NUMERO_PLANTA) ON UPDATE CASCADE ON DELETE NO ACTION
```

15. No es obligatorio poner **NOT NULL** a un atributo que es FOREIGN KEY; se pondrá **NOT NULL** si deseamos impedir se pueda dejar el atributo en blanco y que por tanto tenga que casarse obligatoriamente con una fila de la tabla principal.

16. El orden es indiferente en:

```
#correcto:  
ON UPDATE ... ON DELETE ...  
  
#correcto:  
ON DELETE ...ON UPDATE ...
```

17. La FOREIGN KEY debe apuntar a una PRIMARY KEY o a un UNIQUE en la tabla referenciada
18. Los tipos de dato en ambas tablas deben de ser similares:
- Los datos de tipo numérico deben ser del mismo tamaño y signo.
 - La longitud de los tipos texto no tiene por qué ser la misma, aunque es recomendable.
19. Si ponemos ON DELETE SET NULL, el atributo no puede tener la restricción NOT NULL
20. Si la PRIMARY KEY de la tabla referenciada es SERIAL, el atributo que lo referencia debe de ser BIGINT UNSIGNED
21. Si La FOREIGN KEY está formada por 2 o más atributos, se crea una sola restricción con esos atributos, no varias restricciones:

```
# Foreign Key formada por 2 atributos:  
CONSTRAINT fk_doente_numcama_numplanta FOREIGN KEY (NUMERO_CAMA, NUMERO_PLANTA)  
REFERENCES CAMAS (NUMERO_CAMA, NUMERO_PLANTA) ON UPDATE CASCADE ON DELETE NO ACTION
```

22. MySQL requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. In the referencing table, there must be an index where the foreign key columns are listed as the *first* columns in the same order (el índice debe incluir los atributos de la FOREIGN KEY como sus primeros atributos, y en el mismo orden, pero podría incluir atributos extra a su izquierda). Such an index is created on the referencing table automatically if it does not exist. This index might be silently dropped later, if you create another index that can be used to enforce the foreign key constraint. *index_name*, if given, is used as described previously.

11.2.3 Ejemplos correctos e incorrectos de FOREIGN KEY que provienen de PRIMARY KEY formada por más de un atributo

Si la PRIMARY KEY está formada por más de un atributo, entonces la FOREIGN KEY hay que crearla con una única cláusula FOREIGN KEY que debe traer juntos todos los atributos (y en el mismo orden en que se encuentran en la PRIMARY KEY).

Veamos este ejemplo:

#dada la tabla A, ¿cuáles de las tablas A1, A2, A3 crean bien la FOREIGN KEY?.

```
CREATE TABLE A(
a INTEGER,
b INTEGER,
c INTEGER,
PRIMARY KEY (a,b));

CREATE TABLE A1(
a INTEGER,
b INTEGER,
d VARCHAR(33),
PRIMARY KEY (a,b,d),
FOREIGN KEY AA_a (a) REFERENCES A(a) ON UPDATE CASCADE ON DELETE CASCADE,
FOREIGN KEY AA_b (b) REFERENCES A(b) ON UPDATE CASCADE ON DELETE CASCADE);

CREATE TABLE A2(
a INTEGER,
b INTEGER,
d VARCHAR(33),
PRIMARY KEY (a,b,d),
FOREIGN KEY AA_ab (a,b) REFERENCES A(a,b) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE A3(
a INTEGER,
b INTEGER,
d VARCHAR(33),
PRIMARY KEY (a,b,d),
FOREIGN KEY AA_ba (b,a) REFERENCES A(b,a) ON UPDATE CASCADE ON DELETE CASCADE
);
```

#Respuesta: la A2 es la única correcta.

#Dado que la PK de A está formada por (a,b), la FK debe ser una sola FK formada por (a,b)

#la tabla A1 tiene dos FK porque trocea la PK de A y no se puede hacer.

#la tabla A3 invierte el orden de los atributos de la PK de A1 y no se puede hacer.

Ejemplo:

coche

número matricula	letra matrícula
1111	ABC
2222	DBS

```
primary key (número_matricula, letra_matricula)
```

Multa

número matricula	letra matrícula	fecha	importe	causa
1111	ABC	04/02/2022	150	ir a 60 en zona 50
1111	DBS	14/04/2023	200	ir a 70 en zona 50

```
#incorrecto, no son 2 FK, es una FK que trae a la PK
foreign key numero (número_matricula)
foreign key letra (letra_matricula)
```

```
#correcto:
foreign key numero_letra (número_matricula, letra_matricula)
```

11.3 NULL / NOT NULL

NULL indica que un atributo puede tomar valores nulos.

Cuando de un atributo no se especifica si puede o no puede tomar valores nulos, entonces

por defecto MySQL considera que puede tomar valores nulos:

```
# son equivalentes
dni VARCHAR(9) NULL
dni VARCHAR(9)
```

NOT NULL indica que un atributo NO puede tomar valores nulos:

```
dni VARCHAR(9) NOT NULL
```

11.4 DEFAULT

<http://dev.mysql.com/doc/refman/5.5/en/data-type-defaults.html>

DEFAULT indica el valor por defecto de un atributo. Es el valor que le asigna automáticamente MySQL a un atributo cuando nosotros introducimos un nuevo registro y no indicamos valor para ese atributo.

El valor por defecto de un atributo tiene que ser una constante (con la única excepción de los **TIMESTAMP** que admiten como valor por defecto **CURRENT_TIMESTAMP** -fecha y hora del instante en que se introduce el registro-).

1. Para asignar por defecto la fecha actual a una variable de tipo fecha es necesario crear un **TRIGGER**.

Código de campo cambiado

2. Para asignar por defecto la hora actual a una variable de tipo hora es necesario crear un TRIGGER.

Ejemplo:

```
provincia VARCHAR(9) DEFAULT 'A CORUÑA',
numero_de_hijos TINYINT DEFAULT 0,
fecha DATETIME DEFAULT CURRENT_TIMESTAMP,
salario DOUBLE() DOUBLE DEFAULT NULL,
salario DOUBLE() DOUBLE,
```

Si no se especifica un valor DEFAULT para un atributo entonces MySQL determina su **valor por defecto** del siguiente modo:

- Si el atributo **SÍ admite el valor NULL**: entonces se le asigna implícitamente una cláusula DEFAULT NULL.
- Si el atributo **NO admite el valor NULL**: entonces no se le asigna implícitamente una cláusula DEFAULT, y lo que sucederá al realizar un INSERT, REPLACE o UPDATE depende del modo activo de SQL:
 - **Modo SQL estricto activado y tabla transaccional**: se produce un error y se produce un ROLL BACK de la sentencia problemática.
 - **Modo SQL estricto activado y tabla NO transaccional**: se produce un error, pero si estamos realizando una tarea que afecta a más de una fila, la operación se realizará correctamente para todas las filas excepto la última.
 - **Modo SQL estricto desactivado**: MySQL asigna al atributo el valor implícito de su tipo de dato.
- **Clave primaria**: Si un atributo forma parte de una clave primaria y no se indica explícitamente la cláusula NOT NULL, entonces MySQL le asigna implícitamente la cláusula NOT NULL (porque todos los atributos de una PRIMARY KEY tienen que ser NOT NULL) y además le asigna a ese atributo el valor implícito por defecto de su tipo de dato; si queremos evitar esto, debemos incluir la cláusula NOT NULL en el atributo que forma parte de la clave primaria.

For data entry into a NOT NULL column that has no explicit DEFAULT clause, if an INSERT or REPLACE statement includes no value for the column, or an UPDATE statement sets the column to NULL, MySQL handles the column according to the SQL mode in effect at the time:

- If strict SQL mode is enabled, an error occurs for transactional tables and the statement is rolled back. For non-transactional tables, an error occurs, but if this happens for the second or subsequent row of a multiple-row statement, the preceding rows will have been inserted.
- If strict mode is not enabled, MySQL sets the column to the implicit default value for the column data type.

(ver apartado modos de MySQL: modo estricto y otros)

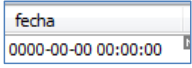
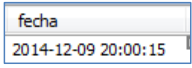
BLOB y TEXT no admiten DEFAULT.

11.4.1 Valor implícito por defecto de cada tipo de dato.

dev.mysql.com/doc/refman/5.0/en/data-type-defaults.html

Cuando MySQL debe asignar un valor por defecto a un atributo, el valor asignado depende del tipo de atributo.

El valor por defecto para cada tipo de atributo es el siguiente:

TIPO DE DATOS	VALOR POR DEFECTO PARA ESE TIPO DE DATOS
Numéricos:	0
Texto:	cadena de texto vacía (se representa como 2 comillas simples seguidas): ''
Fecha, hora (excepto TIMESTAMP)	0, (codificado en forma de fecha u hora) 
TIMESTAMP	fecha y hora actuales. 

11.4.2 DATETIME, DATE, TIME, TIMESTAMP: asignar valores DEFAULT

10) DATETIME

Para asignar por defecto la fecha y hora actuales a un atributo DATETIME:

```
fecha DATETIME DEFAULT CURRENT_TIMESTAMP,
```

11) DATE

Asignar valor por defecto a DATE:

3. DATA y TIME necesitan un trigger, no admiten un método tan fácil como DATETIME

```
# borro el trigger si existe.
DROP TRIGGER IF EXISTS trigger_muestra_before;
DROP TRIGGER IF EXISTS ejemplo01.trigger_muestra_before; # puedo hacer referencia al
trigger usando el schema.
```

```
DELIMITER//
CREATE TRIGGER trigger_muestra_before BEFORE INSERT ON muestra
FOR EACH ROW
BEGIN
```

```
IF ( ISNULL(new.dia) ) THEN
SET new.dia=CURDATE();
END IF;
```

```
END;
//
DELIMITER ;
```

12) TIME

Asignar valor por defecto a TIME:

```
# borro el trigger si existe.
DROP TRIGGER IF EXISTS trigger_muestra_before;

DELIMITER//
CREATE TRIGGER trigger_muestra_before BEFORE INSERT ON muestra
FOR EACH ROW
BEGIN
    IF ( ISNULL(new.hora) ) THEN
        SET new.hora=CURTIME();
    END IF;

END;
//
DELIMITER;
```

13) DATE y TIME

Asignar valor por defecto a DATE y TIME: Hay que meterlos en el mismo trigger

4. Motivo: Before MySQL 5.7.2, there cannot be multiple triggers for a given table that have the same trigger event and action time. For example, you cannot have two BEFORE UPDATE triggers for a table. To work around this, you can define a trigger that executes multiple statements by using the BEGIN ... END compound statement construct after FOR EACH ROW. (An example appears later in this section.)

```
# borro el trigger si existe.
DROP TRIGGER IF EXISTS trigger_muestra_before;

DELIMITER//
CREATE TRIGGER trigger_muestra_before BEFORE INSERT ON muestra
FOR EACH ROW
BEGIN
    IF ( ISNULL(new.dia) ) THEN
        SET new.dia=CURDATE();
    END IF;

    IF ( ISNULL(new.hora) ) THEN
        SET new.hora=CURTIME();
    END IF;

END;
//
DELIMITER ;
```

```

1  # borro el trigger si existe.
2  DROP TRIGGER IF EXISTS trigger_muestra_before;
3
4  delimiter $$
5  CREATE TRIGGER trigger_muestra_before BEFORE INSERT ON muestra
6  FOR EACH ROW
7  BEGIN
8
9  IF ( ISNULL(new.dia) ) THEN
10     SET new.dia=CURDATE();
11  END IF;
12
13  IF ( ISNULL(new.hora) ) THEN
14     SET new.hora=CURTIME();
15  END IF;
16
17  END;
18  $$
19  delimiter ;
20
21

```

14) TIMESTAMP

Asignar valor por defecto:

```
fechaYHora_de_entrada TIMESTAMP DEFAULT NOW(),
```

11.5 AUTO_INCREMENT = 77

fuerza a que el valor inicial de un atributo de tipo auto_increment sea 77

```
ALTER TABLE Personas AUTO_INCREMENT=77
```

11.6 CHECK

<https://dev.mysql.com/doc/refman/8.0/en/create-table-check-constraints.html>
<https://mysqlserverteam.com/mysql-8-0-16-introducing-check-constraint/>
<https://www.w3resource.com/mysql/creating-table-advance/constraint.php>

5. Antes de MySQL 8.0.16 sólo se permitía escribir restricciones mediante CHECK para ser compatible con el estándar SQL, pero **las ignoraba**, por tanto no tenía utilidad especificarlas. En esos casos podíamos intentar crear un **TRIGGER para realizar la función del CHECK**.

En MySQL 8.0.16 se permite este CHECK:

```
[CONSTRAINT [name_constraint]] CHECK (expr) [[NOT] ENFORCED]
```

A CHECK constraint is specified as either a table constraint or column constraint:

- A **table constraint** does not appear within a column definition and can refer to any table column or columns. Forward references are permitted to columns appearing later in the table definition.
- A **column constraint** appears within a column definition and can refer only to that column.

■ Ejemplos:

Consider this table definition:

```
CREATE TABLE t1
(
  CHECK (c1 <> c2),
  c1 INT CHECK (c1 > 10),
  c2 INT CONSTRAINT c2_positive CHECK (c2 > 0),
  c3 INT CHECK (c3 < 100),
  CONSTRAINT c1_nonzero CHECK (c1 <> 0),
  CHECK (c1 > c3)
);
```

The definition includes table constraints and column constraints, in named and unnamed formats:

- The first constraint is a table constraint: It occurs outside any column definition, so it can (and does) refer to multiple table columns. This constraint contains forward references to columns not defined yet. No constraint name is specified, so MySQL generates a name.
- The next three constraints are column constraints: Each occurs within a column definition, and thus can refer only to the column being defined. One of the constraints is named explicitly. MySQL generates a name for each of the other two.
- The last two constraints are table constraints. One of them is named explicitly. MySQL generates a name for the other one.

As mentioned, MySQL generates a name for any CHECK constraint specified without one. To see the names generated for the preceding table definition, use SHOW CREATE TABLE. The SQL standard specifies that all types of constraints (primary key, unique index, foreign key, check) belong to the same namespace. In MySQL, each constraint type has its own namespace per schema (database). Consequently, CHECK constraint names must be unique per schema; no two tables in the same schema can share a CHECK constraint name. (Exception: A TEMPORARY table hides a non-TEMPORARY table of the same name, so it can have the same CHECK constraint names as well.)

■ Cuándo se comprueba el CHECK y que sucede si falla:

CHECK constraints are evaluated for INSERT, UPDATE, REPLACE, LOAD DATA, and LOAD XML statements and an error occurs if a constraint evaluates to FALSE

■ Qué comandos puede incluir un CHECK

CHECK condition expressions must adhere to the following rules. An error occurs if an expression contains disallowed constructs.

- Nongenerated and generated columns are permitted, except columns with the AUTO_INCREMENT attribute and columns in other tables.
- Literals, deterministic built-in functions, and operators are permitted. A function is deterministic if, given the same data in tables, multiple invocations produce the same result, independently of the connected user. Examples of functions that are nondeterministic and fail this definition: CONNECTION_ID(), CURRENT_USER(), NOW().

- Stored functions and loadable functions are not permitted.
- Stored procedure and function parameters are not permitted.
- Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
- Subqueries are not permitted.
- **ENFORCEMENT**

The optional ENFORCEMENT clause indicates whether the constraint is enforced (si está activa o no).

11.6.1 Comandos para añadir, modificar, borrar CHECKS

```
ALTER TABLE Persons
ADD CHECK (Age>=18);

ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');

ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

11.7 Crear una tabla nueva a partir de otra tabla existente, pero sin las KEYS de la original

<https://dev.mysql.com/doc/refman/8.0/en/create-table-select.html>

```
CREATE TABLE IF NOT EXISTS empleados_antiguos_reducida
(SELECT emp_no, apellido1, apellido2, nombre
FROM empleados
WHERE emp_no = 8904
);
```

11.8 Crear una tabla temporal (en memoria RAM) que incluya ciertas filas de otra tabla.

Esa tabla temporal tiene estas propiedades:

- sólo existen en la RAM, no se guarda en disco.
- sólo es accesible por la sesión del usuario que la ha creado.
- El sistema la borra definitivamente de la RAM una vez que el usuario cierra esa sesión.

Es útil para guardar datos temporales que se pueden aprovechar después en otros comandos.

```
CREATE TEMPORARY TABLE IF NOT EXISTS pedidos_ano_2000
(SELECT *
FROM pedidos
WHERE YEAR(FECHA_PEDIDO) = '2000'
);
```

12. ALTER TABLE

<https://dev.mysql.com/doc/refman/8.0/en/alter-table.html>
https://www.w3schools.com/sql/sql_alter.asp
<https://conclase.net/mysql/curso/sqlsen/ALTER TABLE>

12.1.1 Modificar los atributos de una tabla:

Renombrar tabla:

```
ALTER TABLE customers  
  RENAME clientes;
```

Añadir columna:

```
ALTER TABLE clientes  
  ADD Email varchar(255);
```

Cambiar tipo de datos de columna:

```
ALTER TABLE clientes  
  MODIFY COLUMN Email varchar(320);
```

Renombrar columna:

```
ALTER TABLE clientes  
  RENAME COLUMN Email TO CorreoElectronico;
```

Borrar columna:

```
ALTER TABLE clientes  
  DROP COLUMN CorreoElectronico;
```

Añadir PRIMARY KEY:

```
ALTER TABLE personas  
  ADD PRIMARY KEY (a);
```

Añadir ÍNDICE UNIQUE:

```
ALTER TABLE personas  
  ADD UNIQUE (P_Id)
```

#otra forma, dando nombre al índice (a la restricción) para poder borrarlo después si fuese necesario:

```
ALTER TABLE personas  
  ADD CONSTRAINT uq_personas_P_Id UNIQUE (P_Id)
```

Añadir ÍNDICE normal:

```
ALTER TABLE personas  
  ADD INDEX (localidad);
```

#otra forma, dando nombre al índice (a la restricción) para poder borrarlo después:

```
ALTER TABLE personas  
  ADD CONSTRAINT ind_personas_localidad INDEX (localidad)
```

Borrar restricción de clave primaria:

```
ALTER TABLE personas
```

```
DROP PRIMARY KEY;
```

Borrar índice unique o normal:

```
ALTER TABLE personas
DROP INDEX ind_personas_locali;
ALTER TABLE Persons
DROP INDEX uq_personas_P_Id
```

```
#Otra forma (DROP CONSTRAINT):
ALTER TABLE personas
DROP CONSTRAINT ind_personas_locali;
ALTER TABLE Persons
DROP CONSTRAINT uq_personas_P_Id;
```

12.1.2 Añadir una Foreign Key a una tabla

<https://tableplus.com/blog/2019/08/add-foreign-key-to-existing-table-mysql.html>

```
#añadimos a la tabla existente el atributo que será FK:
ALTER TABLE students ADD COLUMN student_id INT NOT NULL;

#modificamos la tabla existente añadiendo la restricción de FOREIGN KEY:
ALTER TABLE students ADD CONSTRAINT fk_student_id FOREIGN KEY (student_id) REFERENCES
points(id);
```

13. ANEXOS

13.1 Resolución de Problemas

13.1.1 Buscar Información duplicada en un campo de una tabla.

Para generar este tipo de consultas lo más sencillo es utilizar el asistente de consultas de Access, editar la sentencia SQL de la consulta y pegarla en nuestro código. No obstante este tipo de consulta se consigue de la siguiente forma:

```
SELECT DISTINCTROW Lista de Campos a Visualizar FROM Tabla
WHERE CampoDeBusqueda In (SELECT CampoDeBusqueda FROM Tabla As psudónimo
GROUP BY CampoDeBusqueda HAVING Count(*)>1 ) ORDER BY CampoDeBusqueda;
```

Un caso práctico, si deseamos localizar aquellos empleados con igual nombre y visualizar su código correspondiente, la consulta sería la siguiente:

```
SELECT DISTINCTROW Empleados.Nombre, Empleados.IdEmpleado FROM Empleados WHERE Empleados.Nombre In (SELECT
Nombre FROM Empleados As Tmp GROUP BY Nombre HAVING Count(*)>1)
ORDER BY Empleados.Nombre;
```

13.1.2 Recuperar Registros de una tabla que no contengan registros relacionados en otra.

Este tipo de consulta se emplea en situaciones tales como saber que productos no se han vendido en un determinado periodo de tiempo,

```
SELECT DISTINCTROW Productos.IdProducto, Productos.Nombre FROM Productos LEFT JOIN Pedidos ON Productos.IdProducto
= Pedidos.IdProduct WHERE (Pedidos.IdProducto Is Null) AND (Pedidos.Fecha Between #01-01-98# And #01-30-98#);
```

La sintaxis es sencilla, se trata de realizar una unión interna entre dos tablas seleccionadas mediante un LEFT JOIN, estableciendo como condición que el campo relacionado de la segunda sea Null.

