ntic master
revolucionamos la comunicación

UNIVERSIDAD COMPLUTENSE MADRID

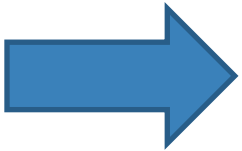
Bases de Datos
4.- SQL

Isabel Riomoros

1 .- Introducción a las Bases de Datos

2.- Modelo Entidad-Relación

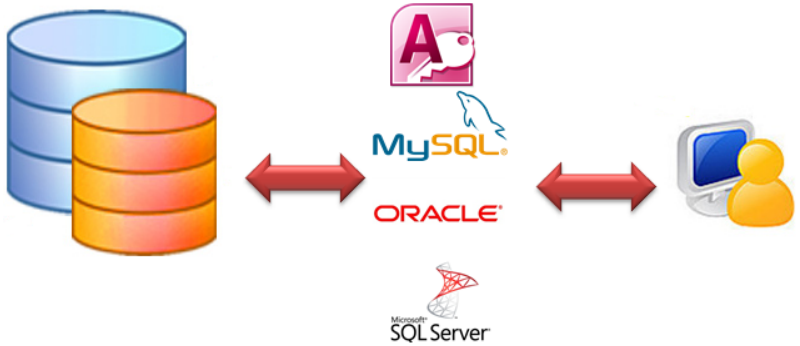
3.- Modelo Relacional



4.- SQL

4

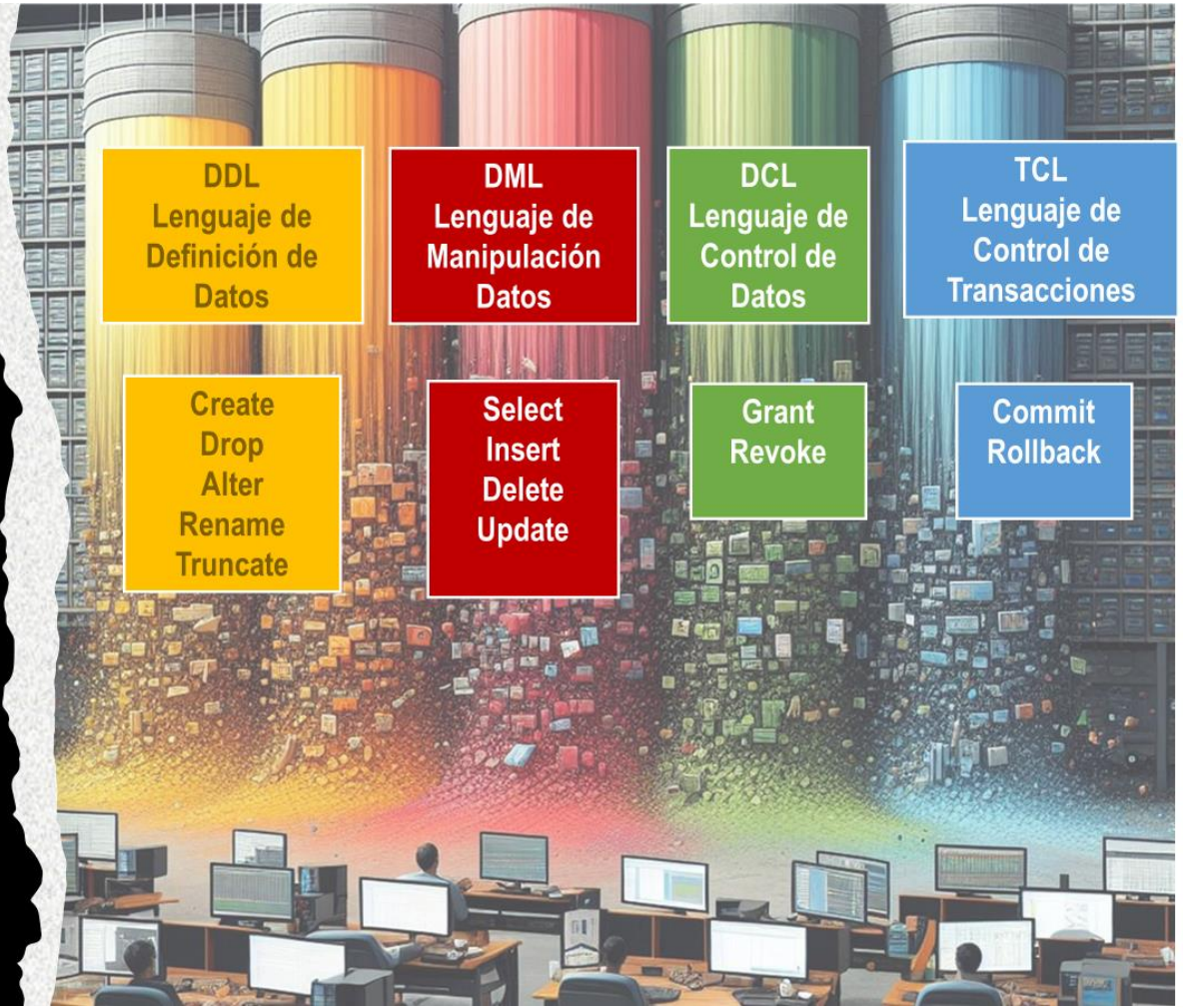
SQL



SQL (Structured Query Language) es un lenguaje estándar e interactivo de acceso a bases de datos relacionales, permite realizar distintas operaciones en ellas, gracias a la utilización del álgebra y del cálculo relacional. SQL nos ofrece la posibilidad de realizar consultas con el fin de recuperar información de las bases de datos, realizando este proceso de forma sencilla. Las consultas permiten seleccionar, insertar, actualizar, averiguar la ubicación de los datos, …

Las bases de datos que utilizaremos para los ejemplos será la del banco y la tienda de informática

TIPOS DE COMANDOS SQL



SQL - (*Structured Query Language*).

codArticulo	denom	precio	unidades
0001	Ord. Sobremesa	600.00	12
0002	Ord. Portátil	1000.00	6
0003	Tarjeta Red	20.00	25
0004	Impresora Láser	200.00	4
0005	Ratón USB	7.00	50
0006	Monitor TFT	250.00	10
0007	Router inalámbrico	100.00	30
NULL	NULL	NULL	NULL

- Las distintas implementaciones de **SQL** pueden **diferenciarse en detalles**, o pueden admitir sólo un subconjunto del lenguaje completo.

- **El resultado** de **ejecutar** una instrucción **SQL** es **una tabla** (tabla resultado) con los registros que cumplen la instrucción ejecutada.

Algunas consideraciones

En SQL **no** se distingue entre mayúsculas y minúsculas. El final de una instrucción o sentencia lo marca el signo **de punto y coma.**

Las sentencias SQL (SELECT, INSERT, …) se pueden escribir en varias líneas siempre que las palabras no sean partidas.

Los comentarios en el código SQL pueden ser de 2 tipos:

```
/*
  Esto es un comentario
  de varias líneas.
   Fin.
*/
 -- Esto es un comentario de una línea
# Esto también es un comentario de línea
```

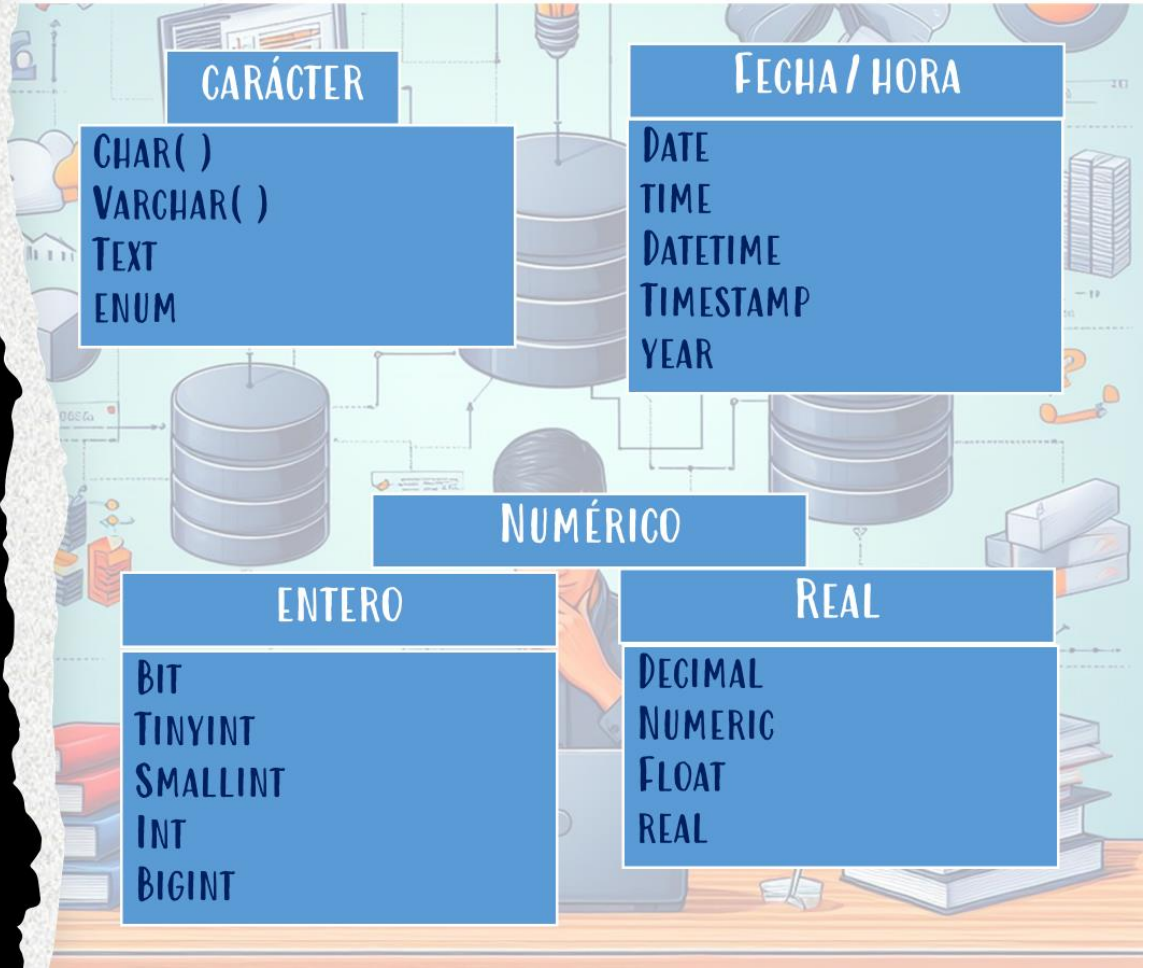
Algunas consideraciones

Las relaciones de cada base de datos deben especificarse en el sistema en términos de un lenguaje de definición de datos (LDD)

Además de las relaciones(tablas), se define la información relativa a ellas:

- Esquema de cada relación
- Dominio de valores asociados a cada atributo
- Restricciones de integridad
- Índices que se mantienen para cada relación
- Información de seguridad y autorización de cada relación
- Estructura de almacenamiento físico de cada relación en disco

TIPOS DE DATOS SQL



Tipos de datos carácter

CHAR- Permite almacenar una cadena de datos con longitud fija.

Siempre reservará espacio para la longitud definida, aunque no se utilice. La longitud máxima es de 255.

VARCHAR- Permite almacenar una cadena de datos (caracteres, números y caracteres especiales) con longitud variable.

No reserva el espacio de la longitud máxima definida, ya que ocupa espacio del tamaño real de los datos. La longitud máxima es de 255.

TEXT- Permite almacenar una cadena de caracteres de longitud máxima de 65,535.

ENUM- Permite definir una lista de posibles valores que pueden almacenarse. La longitud máxima es de 65,535. Si se intenta introducir un valor que no esté incluido en la lista, se insertará valor vacío. Enum(valor1,valor2,..)

Tipos de datos en numéricos: enteros

BIT- Un número entero que puede ser cero o uno

TINYINT - Permite números desde -128 hasta 127.

Si se define como **UNSIGNED** (sin signo) permite números desde 0 hasta 255.

SMALLINT- Permite números desde -32768 hasta 32767.

Si se define como **UNSIGNED** (sin signo) permite números desde 0 hasta 65535.

INT- Permite números desde -2147483648 hasta 2147483647.

Si se define como UNSIGNED (sin signo) permite números desde 0 hasta 4294967295.

BIGINT- Permite números desde -92233720368547758O8 hasta 9223372036854775807.

Si se define como UNSIGNED (sin signo) permite números desde 0 hasta 18446744073709551615.

Tipos de datos numéricos: reales

Float: Número pequeño en coma flotante de precisión simple.
Los valores válidos van desde -3.402823466E+38 a -1.175494351E-38, 0 y desde 1.175494351E-38 a 3.402823466E+38.

xReal, Double:
Número en coma flotante de precisión doble.
Los valores permitidos van desde -1.7976931348623157E+308 a -2.2250738585072014E-308, 0 y desde 2.2250738585072014E-308 a 1.7976931348623157E+308

Decimal, Dec, Numeric:
Número en coma flotante desempaquetado.

Para los tipos **decimal y numeric** se puede especificar el número máximo de dígitos y el número de decimales. Por ejemplo: (6,2) tendrá 4 dígitos enteros y 2 decimales dando un tamaño total de 6, por ejemplo, 1234.56.

Tipos de datos fecha

Mysql no comprueba de una manera estricta si una fecha es válida o no. Comprueba que el mes está comprendido entre 0 y 12 y que el día está comprendido entre 0 y 31.

Date: guarda una fecha. El formato es año-mes-día

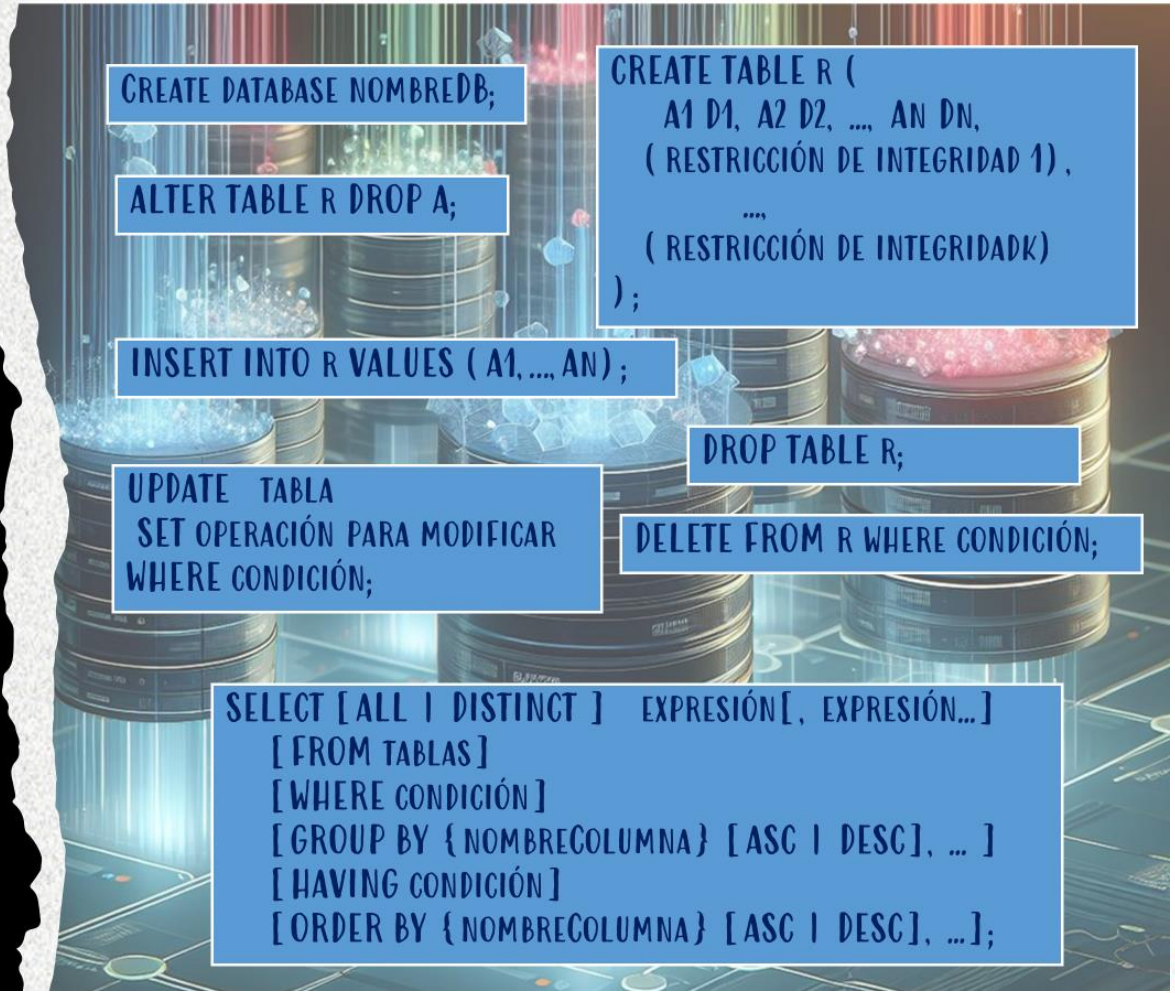
Time: guarda una hora. El formato es 'HH:MM:SS'

DateTime: Combinación de fecha y hora.
El formato es año-mes-dia horas:minutos:segundos

TimeStamp: Combinación de fecha y hora.

Year: guarda un año. El campo puede tener tamaño dos o tamaño 4 dependiendo de si queremos guardar el año con dos o cuatro dígitos.

Sintáxis básica SQL



```
CREATE DATABASE NOMBREDB;
```

```
ALTER TABLE R DROP A;
```

```
CREATE TABLE R (
    A1 D1, A2 D2, ..., AN DN,
  ( RESTRICCIÓN DE INTEGRIDAD 1) ,
        ...,
  ( RESTRICCIÓN DE INTEGRIDADK)
) ;
```

```
INSERT INTO R VALUES ( A1, ..., AN) ;
```

```
DROP TABLE R;
```

```
UPDATE  TABLA
 SET OPERACIÓN PARA MODIFICAR
WHERE CONDICIÓN;
```

```
DELETE FROM R WHERE CONDICIÓN;
```

```
SELECT [ALL | DISTINCT ]  EXPRESIÓN[, EXPRESIÓN...]
    [FROM TABLAS]
    [WHERE CONDICIÓN]
    [GROUP BY {NOMBRECOLUMNA} [ASC | DESC], ... ]
    [HAVING CONDICIÓN]
    [ORDER BY {NOMBRECOLUMNA} [ASC | DESC], ...];
```

Definición básica de esquemas SQL, crear tabla

Para crear una tabla utilizamos el comando **create table:**

CREATE TABLE r (
A_1 D_1,
A_2 D_2,
...,
A_n D_n,
(restricción de integridad $_1$),
...,
restricción de integridad$_k$)
);

r *es el nombre de la* ***relación o tabla***

Cada A_i es un **atributo** del esquema de relación r

D_i es el **tipo del dominio** del atributo A_i

Ejemplos de creación de tablas

```
create table sucursal (
        nombre_sucursal    char(15) ,
        ciudad-sucursal    char(30),
        activos            numeric(16,2) not null
        Primary key nombre_sucursal)
```

```
CREATE TABLE  provincia(
   id SMALLINT AUTO_INCREMENT,
   nombre VARCHAR(30) NOT NULL,
   superficie INTEGER ,
   habitantes INTEGER DEFAULT 0,
   idComunidad SMALLINT NOT NULL,
  Check(superficie>0),
   PRIMARY KEY (id),
   FOREIGN KEY (idComunidad) REFERENCES
comunidad(id)
   ON DELETE CASCADE
   ON UPDATE CASCADE
);
```

```
create table hotel(
   idHotel int auto_increment primary key,
   nomHotel varchar(50) not null,
   direccion varchar(50),
   numH int default 0,
   idPueblo int,
   foreign key (idPueblo) references pueblo(codPueblo)
   on delete cascade
   on update cascade
);
```

```
CREATE TABLE cliente (
  idCliente  char(3),
  nombreC varchar(40) not null,
  direccion varchar(40) ,
  telefono numeric(9,0) not null,
  primary key(idCliente)
);
```

Restricciones de integridad

Las restricciones de integridad protegen contra problemas accidentales en la base de datos, asegurando que los cambios con autorización en la base de datos no generan pérdidas en la consistencia de los datos.

- Obligatoriedad**, NOT NULL**
- Clave primaria**, PRIMARY KEY** (A_1, ..., A_n)- los atributos han de ser no nulos y únicos
- Clave ajena**, FOREIGN KEY** (A_1) **REFERENCES** r (A)
- Verificación de condiciones**, CHECK**
- **AUTO_INCREMENT**
- Valores por defecto**, DEFAULT**

❖ Una cuenta debe tener un saldo mayor que 10.000,00€

Check(saldo>10000)

❖ El salario de un empleado del banco no puede ser menor de 4,00€ la hora ni mayor que 20.

Check(salario >4 and salario>20)

❖ Un cliente debe tener un número de teléfono (no nulo)

Telefono varchar(9) not null,

Claves En SQL

Las claves proporcionan una forma rápida y eficiente de buscar datos en una tabla

- **Clave candidata** es un conjunto no vacío de atributos que identifican unívoca y mínimamente cada tupla de una relación.
- Una **clave primaria** es un atributo o conjunto de atributos que el diseñador ha elegido para identificar de manera única una fila de una tabla.
- Una **clave ajena o externa** es un atributo en una tabla que se corresponde con la clave primaria de otra tabla.

Como elegir la clave primaria

Para elegir la clave primaria puedes crear un atributo numérico nuevo(que no está en el modelo relacional) o elegir uno de los campos varchar que tengas como clave.

En este dilema para decidir cuál es la mejor opción hay mucha polémica y cada opción tiene sus ventajas e inconvenientes.

- **Si añades un atributo numérico**, al añadir un campo que no existe en la tabla estás añadiendo peso (si es un int son 4 bytes), según va creciendo la base de datos y tienes más claves ajenas te ahorras un espacio, las comparaciones y los joins son más rápidos que utilizando varchar. Como inconveniente que son menos intuitivas y cuando haces consultas y quieres sacar el nombre tienes que cruzar con la tabla dónde es clave primaria.
- **Si pones un atributo varchar** son más fáciles de leer y entender para las personas que manejen la base de datos (a la base de datos le da igual realiza la consulta si está bien sintácticamente) No se añade el nuevo campo a la tabla. Las comparaciones y las uniones de tablas son más lentas. Un gran inconveniente sería si tuvieras que cambiarla, imagina que en la tabla persona el nombre fuera la clave y en vez de Alejandro te quisieras llamar Alejandro de Jesús, tendrías que cambiarlo en todas las tablas dónde el nombre fuera clave ajena (aunque si funciona la restricción de clave ajena lo haría solo)

Restricción not null

Declarar que el *nombre_sucursal* de una *sucursal* es **not null**

nombre_sucursal **char**(15) **not null**

Declarar que el dominio *Euros* sea **not null**

create domain *Euros* **numeric**(12,2) **not null**

Restricción unique

- La especificación unique indica que los atributos $A_1, A_2, \ldots A_m$ constituyan una clave candidata.
- Las claves candidatas pueden tener atributos nulos (al contrario que las claves primarias)

Cláusula check

- **check** (*P*), donde *P* es un predicado

Ejemplos de restricciones

```
CREATE TABLE persona (
    idCliente  char(3),
    nombreC varchar(40) not null,
    fechaNac date,
    fechaBoda date,
    telefono  numeric(9,0),
    num int,
   PRIMARY KEY(idCliente));
alter table persona add constraint check(num>0);
alter table persona add constraint
check(fechaBoda>fechaNac);
```

Añadir restricciones una vez creada la tabla. Intentaremos definir las restricciones junto a la definición de la tabla correspondiente. Considero más conveniente añadirlas en la creación de la tabla.

```
CREATE TABLE sucursal(
    nombre_sucursal   CHAR(15),
    ciudad_sucursal    CHAR(30),
    activos        INTEGER,
    PRIMARY KEY (nombre_sucursal),
    CHECK (activos >= 0));
```

Ejemplo: Declarar *nombre_sucursal* como clave primaria para *sucursal* y asegurar que el valor de *activos* no sea negativo.

Integridad referencial

- Asegura que un valor que aparece en una relación para un conjunto de atributos determinado aparezca también en otra relación para un cierto conjunto de atributos.

Ejemplo: *Si "As Pontes" es un nombre de sucursal que aparece en una de las tuplas de la relación cuenta, entonces existirá una tupla en la relación sucursal para la sucursal "As Pontes".*

Si el código del artículo 'Ord. Sobremesa' aparece en la relación compra, ese código de artículo debe aparecer en la relación artículo.

Las claves primarias, candidatas y las claves externas o ajenas se pueden especificar como parte de la instrucción **create table** de SQL:

- La cláusula **primary key** incluye una lista de los atributos que comprende la clave primaria.
- La cláusula **unique key** incluye una lista de los atributos que comprende una clave candidata.
- La cláusula **foreign key** incluye una lista de los atributos que comprende la clave externa y el nombre de la relación a la que hace referencia mediante la clave externa. Por defecto, una clave externa hace referencia a los atributos de la clave primaria de la tabla referenciada.

La diferencia entre unique y primary key, una clave unique permite nulos, en cuanto una primary key no permite nulos es decir ya incluye la constraint de not null para cada atributo.

Restricción clave ajena

```
FOREIGN KEY(idCliente) REFERENCES cliente(idCliente)
        ON DELETE cascade,
```

- Sirve para relacionar dos o más tablas, se necesita un campo en común, por ejemplo, idCliente y idCliente, existe en cliente y en compra.
- Si queremos eliminar algún cliente, las filas que se correspondan en compras con ese cliente serán eliminadas automáticamente.

Índices

- Los índices permiten localizar y devolver registros de una manera sencilla y rápida, son útiles cuando queremos buscar elementos si tenemos miles de registros.
- Cuando no usamos índices, podemos percibir que la consulta tarda en ejecutarse.
- Sin un índice, la búsqueda se inicia con el primer registro y busca por toda la tabla para encontrar los registros importantes. Búsqueda secuencial.
- Un índice de base de datos se parece mucho a un índice de un libro: ocupa su propio espacio, es redundante y hace referencia a la información actual almacenada en otro lugar.
- Se generan índices de forma automática para los atributos definidos como clave primaria y clave candidata(unique)

Índices, sintaxis para su creación

Create index nombreIndice **on** nombreTabla(nombreDelAtributo);

Show index from nombreTabla;

En la creación de una tabla se puede poner

Index nombreDelIndice(nombreAtributo)

Se suelen crear índices en los campos por los que vamos a buscar, ordenar, agrupar en muchas ocasiones

Ejemplo creación de tablas, base de datos tienda de informática

```
CREATE TABLE cliente (
  idCliente  varchar(3),
  nombreC varchar(40) not null,
  direccion varchar(40) ,
  telefono  numeric(9,0) not null,
  PRIMARY KEY(idCliente)
);
```

```
CREATE TABLE articulo (
  idArticulo  varchar(4),
  nomArticulo    varchar(40) not null,
  precio    numeric(6,2) not null,
  unidades    integer not null,
   descuento numeric(3,0),
  PRIMARY KEY(idArticulo)
);
```

```
CREATE TABLE compra (
  idCliente     varchar(3),
  idArticulo     varchar(4),
  fecCompra  date     not null,
  numUnidades   integer not null,
 CHECK(numUnidades)>0,
  PRIMARY KEY(idCliente, idArticulo,fecCompra),
  FOREIGN KEY(idCliente) REFERENCES cliente(idCliente)
        ON DELETE cascade,
  FOREIGN KEY(idArticulo) REFERENCES articulo(idArticulo)
        ON DELETE cascade
);
```

Definición básica de esquemas SQL

Añadir datos a una tabla o relación

INSERT INTO r **VALUES** (A_1,…,A_n)

Borrar todas las tuplas de una table o relación

DELETE FROM r ;

Borrar algunas tuplas de una relación

DELETE FROM r **WHERE** condicion;

Eliminar una relación en una base de datos.

DROP TABLE r

Añadir atributos a una relación existente

ALTER TABLE *r* **ADD** *A D*

Cambiar el tipo de datos de un atributo

ALTER TABLE r **MODIFY** A tipoDeDatos;

Borrar atributos

ALTER TABLE *r* **DROP** *A*

Actualizar datos de las tablas de una BD.

UPDATE tabla
SET operación para modificar
WHERE condición;

UPDATE *empleados*
SET *Direccion='Gran Vía 241', telefono='686567687'*
WHERE *Nombre='Ana García';*

Si se omite WHERE, se actualizan todas las filas de la tabla destino.

Ejemplos

```
INSERT INTO cliente VALUES ('015', 'Pedro Glez.', 'Gerona 14',      917845308);
INSERT INTO articulo VALUES ('0001', 'Ord. Sobremesa',     600, 12);
INSERT INTO compra VALUES('015', '0007', '2015/11/06', 2);
 INSERT INTO cuenta VALUES ('A-9732', 'Navacerrada',1200);
INSERT INTO cuenta VALUES ('A-777','Navacerrada', null);
```

```
DROP DATABASE IF EXISTS tiendaInformatica;
CREATE DATABASE tiendaInformatica;
USE tiendaInformatica;
```

Cargar datos de un archivo, mirar documento cargar datos en workbench

```
DROP TABLE IF EXISTS compra;
DROP TABLE IF EXISTS cliente;
DROP TABLE IF EXISTS articulo;
```

```
set sql_safe_updates=0;
```

Para poder realizar modificaciones y borrados

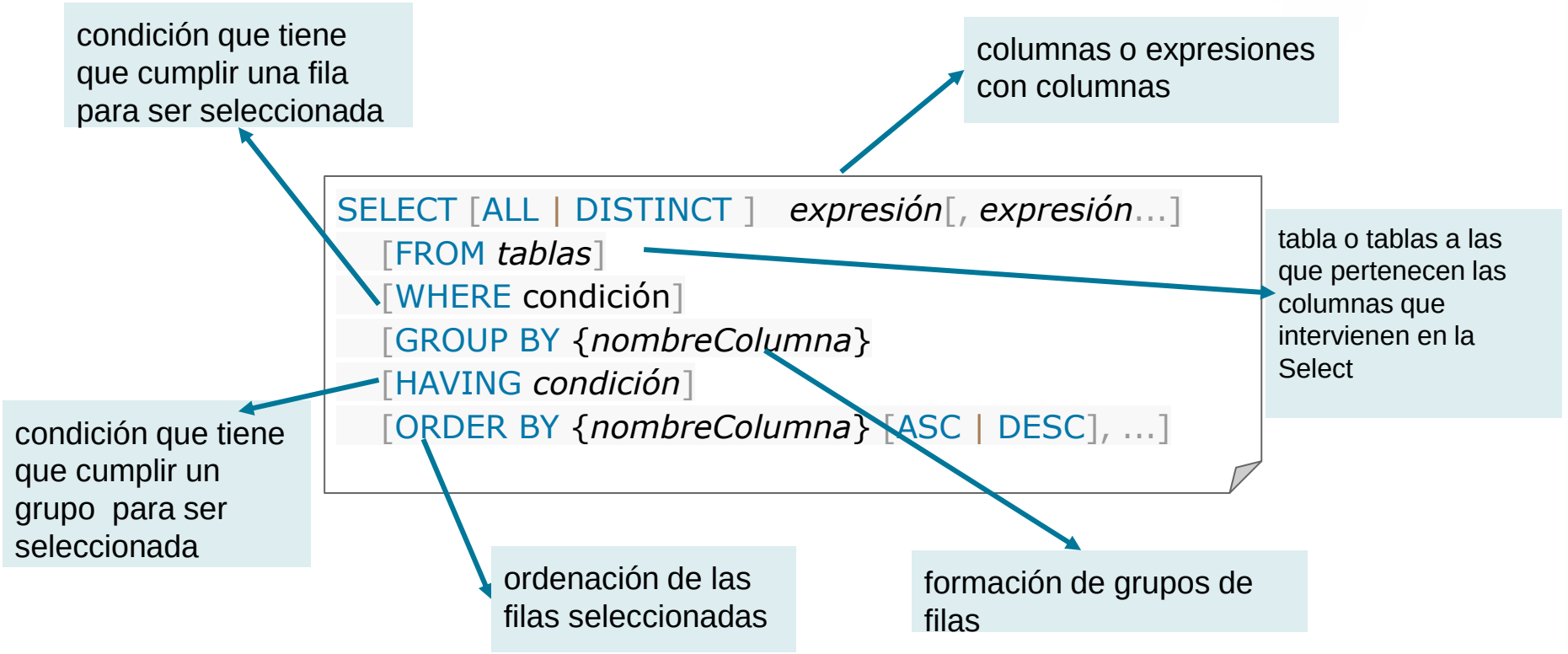
Los ficheros a cargar es necesario guardarlos en el directorio que nos dé la siguiente select:

SELECT @@GLOBAL.secure_file_priv;

La carpeta dependiendo de la versión de workbench, es muy parecida a: 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads'

**Ojo / y no **

La claúsula SELECT



La claúsula SELECT

Una consulta característica de SQL tiene la forma:

SELECT $A_1, A_2, ..., A_n$
FROM $r_1, r_2, ..., r_m$
WHERE P

A_i representan los atributos
r_i representan las relaciones
P es un predicado

El resultado de una consulta de SQL es una relación.
La cláusula **SELECT** se utiliza para dar la relación de los atributos deseados en el resultado de una consulta

Ejemplo: *obtener los nombres de todas las sucursales en la relación prestamo:*

SELECT *nombre_sucursal*
FROM *prestamo*

Ejemplo: *obtener el listado de todos los datos de los artículos de la tienda de productos informáticos*

SELECT *
FROM *articulo*

Todos los atributos

La cláusula SELECT

Para forzar la eliminación de duplicados, insertar la clave **distinct** después de SELECT.

Obtener los nombres de todas las sucursales en las relaciones prestamos, y anular los duplicados

SELECT DISTINCT *nombre_sucursal*
FROM *préstamo*

La clave **all** especifica que los duplicados no se han anulado.

SELECT ALL *nombre_sucursal*
FROM *prestamo*

La cláusula **SELECT** puede contener expresiones aritméticas que involucran la operación, +, –, * y /, y que funcionan en las constantes o en los atributos de las tuplas.

La siguiente relación es la misma que préstamo, excepto que el atributo importe se multiplica por 100. pero no modifica la Base de Datos.

SELECT *número_prestamo, nombre_sucursal, importe * 100*
FROM *prestamo*

La cláusula WHERE

La cláusula **WHERE** especifica las condiciones que debe satisfacer el resultado

La búsqueda de todos los números de crédito de los prestamos ha dado como resultado la sucursal Navacerrada con las cantidades de prestamos mayores a 1200 €.

SELECT *número_prestamo*
FROM *prestamo*
WHERE *nombre_sucursal* = ‘ Navacerrada’ **AND** *importe* > 1200

Escribir los artículos que se han acabado

SELECT *idArticulo, nomArticulo*
FROM *articulo*
WHERE *unidades =0*

Los resultados de la comparación se pueden combinar utilizando las conectivas lógicas **and, or** y **not.** Las comparaciones se pueden aplicar a los resultados de las expresiones aritméticas.

SQL incluye un operador de comparación **BETWEEN**

Ejemplo: *Obtener el número de préstamo de aquellos con cantidades de crédito entre 90,000€ y 100,000€*

SELECT *número_prestamo*
FROM *prestamo*
WHERE *importe* **between** 90000 **AND** 100000

Clausula from: consultas con varias tablas

- Las tablas se relacionan mediante claves ajenas, identificador que permite relacionar una tabla con otra.
- Si queremos combinar tablas para mostrar información de diferentes tablas como si fuese sólo una tabla, tenemos que hacer coincidir los valores de las columnas relacionadas.
- Si no hacemos coincidir las columnas relacionadas además de obtener gran duplicidad de filas, se realizará el producto cartesiano de las tablas.

  Por ejemplo:
  Select * from articulo; -- obtenernos 8 artículos, 8 filas
  Select * from compra; -- obtenemos 15 compras, 15 filas

  Y si hacemos la consulta
  Select * from articulo,compra; -- obtenemos más de 50 filas

- Si queremos obtener sólo los artículos que se han comprado tenemos que poner ***where compra.idArticulo=articulo.idArticulo***

La clausula FROM

En la clausula **FROM** se especifica una lista de las relaciones que se van a explorar en la evaluación de la expresión. Corresponde a la operación del producto cartesiano del álgebra relacional.

Buscar el producto cartesiano prestatario X prestamo

```
SELECT *
FROM prestatario, prestamo
```

- *Buscar el nombre, el número de préstamo y la cantidad del préstamo de todos los clientes que tengan un crédito en la sucursal Navacerrada.*

```
SELECT nombre_cliente, prestatario.numero_prestamo, importe
FROM prestatario, prestamo
WHERE prestatario.número_prestamo= prestamo.numero_prestamo
      AND   nombre_sucursal = 'Navacerrada'
```

nombreTabla.nombreAtributo

- *Nombre de los clientes que han comprado más de 3 unidades*

```
SELECT  distinct nombre
FROM cliente, compra
WHERE cliente.idCliente=compra.IdCliente AND compra.numUnidades>3
```

La operación de renombramiento as

SQL permite renombrar las relaciones y atributos utilizando la cláusula **as**:

nombre_antiguo **as** *nombre_Nuevo*

Obtener el nombre, el número de préstamo y la cantidad del préstamo de todos los clientes; renombrar el nombre de la columna número_préstamo como identificador_prestamo.

```
SELECT nombre_cliente, prestatario.numero_préestamo AS identificador_prestamo, importe
FROM prestatario, prestamo
WHERE prestatario.numero_prestamo= prestamo.numero_prestamo
```

```
SELECT nombre_cliente, prestatario.numero_prestamo AS identificador_prestamo, importe
FROM prestatario AS p, prestamo AS pr
WHERE p.numero_prestamo= pr.numero_prestamo
```

```
SELECT cliente.nombreC, cliente.telefono
FROM cliente as cli, compra as co
WHERE cli.idCliente = co.idClienteAND compra.idArticulo = '0006';
```

La operación de renombramiento

Las variables tupla se definen en la cláusula **FROM** mediante el uso de la cláusula **as**.

- *Obtener los nombres , números de préstamo e importe de todos los clientes que tengan un préstamo en alguna sucursal.*

```
SELECT nombre_cliente, T.número_prestamo, S.importe
FROM prestatario as T, préstamoas S
WHERE  T.número_préstamo= S.número_prestamo
```

- *Obtener los nombres de todas las sucursales que tengan activos mayores que las sucursales situadas en Barcelona.*

```
SELECT distinct T.nombre_sucursal
FROM sucursal as T, sucursal as S
 WHERE T.activos> S.activos AND S.ciudad_sucural = ‘ Barcelona’
```

- *Obtener el nombre del cliente y el teléfono de todos los clientes que han comprado el articulo ‘0006’.*

```
SELECT nombreC, telefono
FROM cliente as cli, compra as co
WHERE cli.idCliente = co.idClienteAND co.idArticulo = '0006';
```

Operaciones con cadenas

SQL incluye un operador de coincidencia de cadenas para comparaciones de cadenas de caracteres. *el operador “LIKE” utiliza patrones que son descritos por los caracteres especiales:

tanto por ciento**(%).** El carácter **%** encaja con cualquier **subcadena**.
guión bajo **(_)**. El carácter _ encaja con cualquier **carácter**.

Obtener los nombres de todos los clientes cuyas calles incluyan la subcadena “Mayor”.
SELECT *nombre_cliente*
FROM *cliente*
WHERE *calle_cliente* **LIKE** ‘%Mayor%’
Coincide el nombre “Mayor%” (para que puedan contener los caracteres especiales, se pone la palabra clave escape.
LIKE ‘Mayor\%’ **escape** ‘\’
SQL soporta una variable de operaciones con cadenas como concatenación (que utiliza “||”) conversión de mayúscula a minúsculas(y viceversa) **upper() lower()**

Búsqueda de la longitud de la cadena, extracción de subcadena, etc.

Reunión de relaciones

Las **operaciones de reunión** toman dos relaciones y las devuelven como resultado otra relación.

Estas operaciones adicionales se utilizan generalmente como expresiones de subconsulta de la cláusula **FROM**

Condición de reunión – define qué tuplas de las dos relaciones coinciden, y qué atributos están presentes en el resultado de la reunión.

Tipo de reunión – define cómo se tratan las tuplas de cada relación que no coincide con ninguna tupla de la otra relación (basada en la condición de reunión).

Tipos de reunión
inner join **left outer join** **right outer join**

Join

Cliente as A

CodCliente	ombreC	direccion	telefono
008	orcuato Montero	Rio Duero 14	937846308
009	suncion Rodríguez	Pez 14	914565308
010	ustquia Alonso	Rio Lozoya 35	917845208
011	ngela Callejo	Pedro Villar 330	914849303
012	aribel Riocal	Luna 11	914394943
013	uan Antonio Sanz	Clavel 21	915656501
014	lara Garcia	Cercona 57	913389307
015	sabel Sanrio	Travesia del rio 14	917845308
016	ugenio Arribas	Tinajas 14	917845308

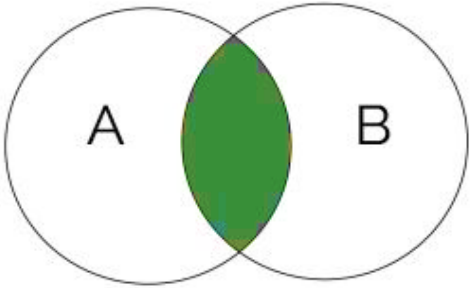
Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45

Inner join, left join y rigth join

tablaA **inner Join** tablaB

INNER JOIN: Devuelve **todas las filas** cuando hay al menos **una coincidencia** en **ambas** tablas.



SELECT nombreColumna(s)
FROM tablaA **INNER JOIN** tablaB **ON** tablaA.nombreColumna=tablaB.nombreColumna;

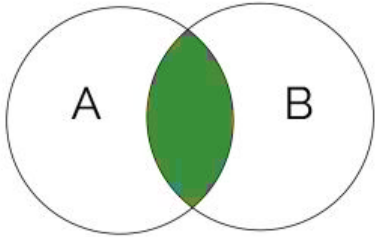
Cliente **inner Join** compra

Cliente as A

CodCliente	nombreC	direccion	telefono
008	Torcuato Montero	Rio Duero 14	937846308
009	Asuncion Rodríguez	Pez 14	914565308
010	Eustquia Alonso	Rio Lozoya 35	917845208
011	Angela Callejo	Pedro Villar 330	914849303
012	Maribel Riocal	Luna 11	914394943
013	Juan Antonio Sanz	Clavel 21	915656501
014	Clara Garcia	Cercona 57	913389307
015	Isabel Sanrio	Travesia del rio 14	917845308
016	Eugenio Arribas	Tinajas 14	917845308

Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45



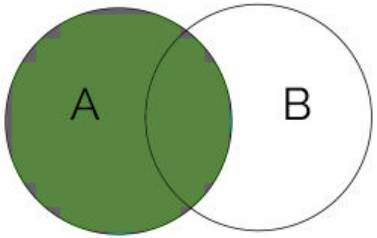
Nombres de los clientes y artículos que han comprado

select idCliente, nombreC, idCliente, idArticulo
from cliente **inner join** compra **on** cliente.idCliente=compra.IdCliente;

codCliente	nombreC	idCliente	idArticulo
011	Angela Callejo	011	0001
011	Angela Callejo	011	0005
012	Maribel Riocal	012	0002
012	Maribel Riocal	012	0003
013	Juan Antonio Sanz	013	0003
013	Juan Antonio Sanz	013	0006
015	Isabel Sanrio	015	0002
015	Isabel Sanrio	015	0004
015	Isabel Sanrio	015	0007

tablaA **left Join** tablaB

LEFT JOIN: Devuelve todas las filas de la tabla de la **izquierda**, y las filas coincidentes de la tabla de la **derecha**. El resultado es NULL en la parte de la derecha cuando no hay registros que correspondan con la condición(igualdad de las claves).




```
SELECT nombreColumna(s)
FROM tablaA LEFT JOIN tablaB ON
                    tablaA.nombreColumna=tablaB.nombreColumna;
```

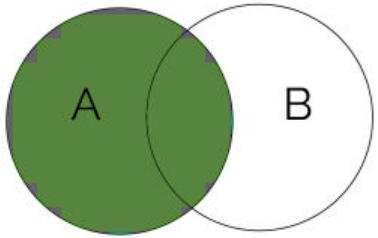
Cliente as A

CodCliente	nombreC	direccion	telefono
008	Torcuato Montero	Rio Duero 14	937846308
009	Asuncion Rodríguez	Pez 14	914565308
010	Eustquia Alonso	Rio Lozoya 35	917845208
011	Angela Callejo	Pedro Villar 330	914849303
012	Maribel Riocal	Luna 11	914394943
013	Juan Antonio Sanz	Clavel 21	915656501
014	Clara Garcia	Cercona 57	913389307
015	Isabel Sanrio	Travesia del rio 14	917845308
016	Eugenio Arribas	Tinajas 14	917845308

Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45

Cliente **left Join** compra




```
select idCliente, nombreC
from cliente left join compra on cliente.idCliente=compra.IdCliente;
```

codCliente	nombreC	idCliente	idArticulo
008	Torcuato Montero	NULL	NULL
009	Asuncion Rodríguez	NULL	NULL
010	Eustquia Alonso	NULL	NULL
011	Angela Callejo	011	0001
011	Angela Callejo	011	0005
012	Maribel Riocal	012	0002
012	Maribel Riocal	012	0003
013	Juan Antonio Sanz	013	0003
013	Juan Antonio Sanz	013	0006
014	Clara Garcia	NULL	NULL
015	Isabel Sanrio	015	0002
015	Isabel Sanrio	015	0004
015	Isabel Sanrio	015	0007
016	Eugenio Arribas	NULL	NULL

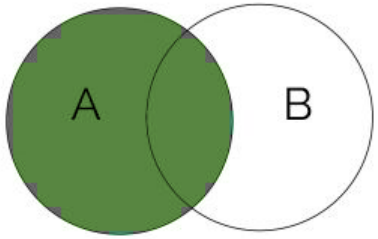
Cliente **left Join** compra

Cliente as A

CodCliente	nombreC	direccion	telefono
008	Torcuato Montero	Rio Duero 14	937846308
009	Asuncion Rodríguez	Pez 14	914565308
010	Eustquia Alonso	Rio Lozoya 35	917845208
011	Angela Callejo	Pedro Villar 330	914849303
012	Maribel Riocal	Luna 11	914394943
013	Juan Antonio Sanz	Clavel 21	915656501
014	Clara Garcia	Cercona 57	913389307
015	Isabel Sanrio	Travesia del rio 14	917845308
016	Eugenio Arribas	Tinajas 14	917845308

Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45



Nombres de los clientes que no han comprado

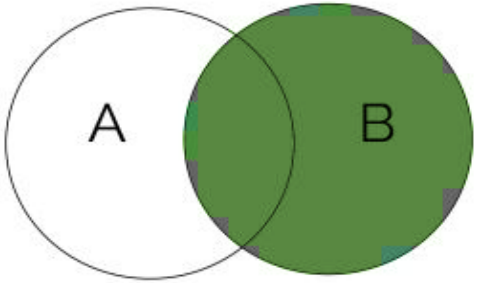
```
select idCliente, nombreC
from cliente left join compra on cliente.idCliente=compra.IdCliente
where idCliente is NULL;
```

codCliente	nombreC
008	Torcuato Montero
009	Asuncion Rodríguez
010	Eustquia Alonso
014	Clara Garcia
016	Eugenio Arribas

Los clientes que están en la tabla clientes, pero que no han registrado una compra, que no aparecen en compra, que no existe su idCliente en compra

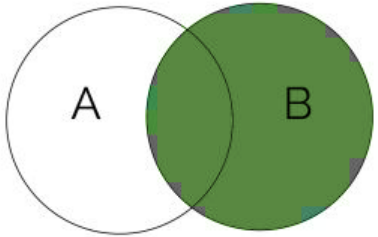
tablaA **rigth Join** tablaB

RIGHT JOIN: Devuelve todas las filas de la tabla de la **derecha**, y las filas coincidentes de la tabla de la **izquierda**.




```
SELECT nombreColumna(s)
FROM tablaA RIGTH JOIN tablaB ON
                    tablaA.nombreColumna=tablaB.nombreColumna;
```

Cliente **right Join** compra



Nombres de los clientes que han comprado alguna vez

Cliente as A

CodCliente	nombreC	direccion	telefono
008	Torcuato Montero	Rio Duero 14	937846308
009	Asuncion Rodríguez	Pez 14	914565308
010	Eustquia Alonso	Rio Lozoya 35	917845208
011	Angela Callejo	Pedro Villar 330	914849303
012	Maribel Riocal	Luna 11	914394943
013	Juan Antonio Sanz	Clavel 21	915656501
014	Clara Garcia	Cercona 57	913389307
015	Isabel Sanrio	Travesia del rio 14	917845308
016	Eugenio Arribas	Tinajas 14	917845308

Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45

```
select distinct(idCliente), nombreC
from cliente right join compra on cliente.idCliente=compra.IdCliente;
```

codCliente	nombreC
011	Angela Callejo
012	Maribel Riocal
015	Isabel Sanrio
013	Juan Antonio Sanz

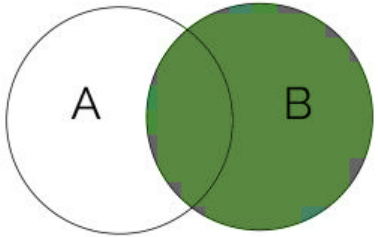
Cliente as A

CodCliente	nombreC	direccion	telefono
008	Torcuato Montero	Rio Duero 14	937846308
009	Asuncion Rodríguez	Pez 14	914565308
010	Eustquia Alonso	Rio Lozoya 35	917845208
011	Angela Callejo	Pedro Villar 330	914849303
012	Maribel Riocal	Luna 11	914394943
013	Juan Antonio Sanz	Clavel 21	915656501
014	Clara Garcia	Cercona 57	913389307
015	Isabel Sanrio	Travesia del rio 14	917845308
016	Eugenio Arribas	Tinajas 14	917845308

Compra as B

idCliente	idArticulo	fecCompra	numUnidades
011	0001	2025-10-06	1
011	0005	2026-10-06	2
012	0002	2001-11-06	1
012	0003	2001-11-06	3
013	0003	2027-10-06	2
013	0006	2027-10-06	2
015	0002	2024-11-06	1
015	0004	2003-11-06	1
015	0007	2015-11-06	45

compra **right Join** cliente



Nombres de los clientes que **no** han comprado

```
select idCliente, nombreC, idCliente, idArticulo
from compra right join cliente on cliente.idCliente=compra.IdCliente
where idCliente is NULL ;
```

codCliente	nombreC	idCliente	idArticulo
008	Torcuato Montero	NULL	NULL
009	Asuncion Rodríguez	NULL	NULL
010	Eustquia Alonso	NULL	NULL
011	Angela Callejo	011	0001
011	Angela Callejo	011	0005
012	Maribel Riocal	012	0002
012	Maribel Riocal	012	0003
013	Juan Antonio Sanz	013	0003
013	Juan Antonio Sanz	013	0006
014	Clara Garcia	NULL	NULL
015	Isabel Sanrio	015	0002
015	Isabel Sanrio	015	0004
015	Isabel Sanrio	015	0007
016	Eugenio Arribas	NULL	NULL

Sale la misma tabla que en la consulta de left join, en este caso es rigth join y hemos permutado las tablas

Relaciones prestamo, prestatario, cuenta

prestamo

número-prestamo	*nombre-sucursal*	*importe*
L-200	Madrid	3000
L-230	Rascafria	4000
L-260	Navacerrada	1700

prestatario

nombre-cliente	*número-prestamo*
Gonzalez	L-200
Pérez	L-230
López	L-155

cuenta

nombre-sucursal	*número-cuenta*	*saldo*
Navacerrada	A-102	400
Navacerrada	A-201	900
Lozoya	A-217	750
Lozoya	A-215	750
Rascafria	A-222	700

Cuenta agrupada por nombre de sucursal

nombre-sucursal	*saldo*
Navacerrada	1300
Barcelona	1500
Reus	700

Reunión de relaciones – Ejemplos

Préstamo **inner join** *prestatario* **on** *prestamo.número_préstamo= prestatario.número_prestamo*

número-préstamo	*nombre-sucursal*	*importe*	*nombre-cliente*	*número-prestamo*
P-170 P-230	Centro Moralzarzal	3000 4000	Santos Gómez	P-170 P-230

Préstamo **left inner join** *prestatario* **on** *prestamo.número_préstamo= prestatario.número_prestamo*

número-prestamo	*nombre-sucursal*	*importe*	*nombre-cliente*	*número-prestamo*
P-170 P-230 P-260	Centro Moralzarzal Navacerrada	3000 4000 1700	Santos Gómez *null*	P-170 P-230 *null*

Orden en la presentación de las tuplas (order by)

Lista en orden alfabético los nombres de todos los clientes que tengan un crédito en la sucursal Navacerrada

SELECT distinct *nombre_cliente*
FROM *prestatario, prestamo*
WHERE *prestatario.número_préstamo=prestamo.número_préstamo* **AND**
sucural_nombre = 'Navacerrada'
order by *nombre_cliente;*

Listar los clientes en orden descendente

SELECT * FROM cliente ORDER BY nombreC DESC;

Se puede especificar la cláusula **desc** para orden descendente o **asc** para orden ascendente, de cada atributo; ***el orden ascendente es el orden por defecto.***

Ejemplo: **order by** *nombre_cliente* **desc**

Operaciones con conjuntos

Las operaciones de conjuntos **union, intersect,** y **except** operan sobre relaciones y corresponden a las operaciones de álgebra relacional ∪, ∩, –. Cada una de las operaciones antes citadas elimina duplicados automáticamente; para retener todos los duplicados se utilizan las versiones de multiconjunto correspondientes **union all, intersect all** y **except all.**

Obtener todos los clientes que tengan un préstamo, una cuenta o ambos:

(SELECT *nombre_cliente* **FROM** *impositor*)
union
(SELECT *nombre_cliente* **FROM** *prestatario)*

Obtener todos los clientes que tengan un préstamo y una cuenta.

(**SELECT** *nombre_cliente* **FROM** *impositor*)
intersect
(SELECT *nombre_cliente* **FROM** *prestatario)*

Obtener todos los clientes que tengan una cuenta pero no un préstamo.

(**SELECT** *nombre_cliente* **FROM** *impositor*)
except
(SELECT *nombre_cliente* **FROM** *prestatario)*

Funciones de agregación

Estas funciones operan en el multiconjunto de valores de una columna de una relación, y devuelven un valor

avg: valor medio
min: valor mínimo
max: valor máximo
sum: suma de valores
count: número de valores

Obtener el saldo medio de las cuentas de la sucursal Navacerrada.

SELECT avg *(saldo)*
FROM *cuenta*
WHERE *nombre_sucursal* = 'Navacerrada'

Obtener el número de tuplas de la relación cliente

SELECT count (*)
FROM *cliente*

Obtener el número de impositores en el banco

SELECT count (distinct *nombre_clientes)*
FROM *impositor*

Funciones de agregación – Group By

Obtener el número de impositores de cada sucursal.

SELECT *nombre_sucursal,* **count (distinct** *nombre_cliente)*
FROM *impositor, cuenta*
WHERE *impositor.número_cuenta = cuenta.número_cuenta*
group by *nombre_sucursal*

Los atributos de la cláusula SELECT fuera de las funciones de agregación deben aparecer en la lista group by

```
SELECT  nombre_sucursal, sum(saldo), count(*),  avg(saldo), min(saldo), max(saldo)
FROM cuenta
group by nombre_sucursal;
```

nombreSucur...	sum(saldo)	count(*)	avg(saldo)	min(saldo)	max(saldo)
Becerril	1000	3	333.3333	100	700
Centro	900	1	900.0000	900	900
Collado Mediano	42350	3	14116.6667	350	30000
Galapagar	235650	3	78550.0000	750	234000

Funciones de agregación –Cláusula Having

Clausula **Having** búsqueda por grupos, una vez agrupado se pregunta en having la condición que tienen que cumplir los grupos.

Obtener los nombres de todas las sucursales en las que el saldo medio de las cuentas es mayor de 1.200€.

```
SELECT nombre_sucursal, avg (saldo)
FROM cuenta
group by nombre_sucursal
having avg (saldo) > 1200
```

Los predicados de la cláusula having se aplican después de la formación de grupos mientras que los permitidos en la cláusula WHERE se aplican antes de la formación de grupos

Clausula WHERE y Having

La cláusula WHERE se aplica primero a las filas individuales de las tablas. Solo se agrupan las filas que cumplen las condiciones de la cláusula WHERE.

La cláusula HAVING se aplica a continuación a las filas del conjunto de resultados. Solo aparecen en el resultado de la consulta los grupos que cumplen las condiciones HAVING. Solo puede aplicar una cláusula HAVING a las columnas que también aparecen en la cláusula GROUP BY o en una función de agregado.

```
SELECT editorial, count(*)
FROM libros
 WHERE editorial<>'Planeta'
GROUP BY editorial;

SELECT editorial, count(*)
FROM libros
GROUP BY editorial
HAVING editorial<>'Planeta';
```

Ambas devuelven el mismo resultado, pero son diferentes.
La primera, selecciona todos los registros rechazando los de editorial "Planeta" y luego los agrupa para contarlos.

La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza fila con la cuenta correspondiente a la editorial "Planeta“, mas ineficiente.

Clausula WHERE y Having

```
SELECT Cod_Depto, COUNT(*) numE
FROM empleados
GROUP BY Cod_Depto
HAVING COUNT(*) >= 2
ORDER BY numE DESC;
```

Obtiene el número de empleados por departamento siempre que haya al menos 2 empleados en el departamento. Además, se ordena la salida por el número de empleados por departamento en orden descendente.

- **WHERE**: Selecciona las filas
- **GROUP BY**: Agrupa estas filas
- **HAVING**: Filtra los grupos. Selecciona y elimina los grupos
- **ORDER BY**: Clasifica la salida. Ordena los grupos.

Valores nulos

Es posible que las tuplas tengan un valor nulo, indicado por medio de ***null*****,** en alguno de sus atributos

***null* significa un valor desconocido o un valor que no existe**.

El predicado **is null** se puede utilizar para comprobar los valores nulos.

Ejemplo: obtener todos los números de préstamos que aparecen en la relación préstamo con valores nulos para importe

SELECT *número_prestamo*
FROM prestamo
WHERE importe **is null**

El resultado de la expresión aritmética que involucra a *null* es nulo

Ejemplo: 5 + *null* devuelve nulo

Todas las operaciones agregadas excepto count(*) ignoran las tuplas con valores nulos de los atributos agregados

Valores nulos y lógica de tres valores

Cualquier comparación con *null* se convierte en *desconocido*

Ejemplo*: 5 < null o null <> null o null = null*

Lógica de tres valores que utiliza el valor real desconocido:

OR: (*desconocido* **or** *cierto*) = *true*, (*desconocido* **or** *falso*) = *desconocido,* (*desconocido* **or** *desconocido) = desconocido*

AND: *(cierto* **AND** *desconocido) = desconocido, (falso* **AND** *desconocido) = falso,*
(desconocido **AND** *desconocido) = desconocido*

NOT*: (***not** *desconocido) = desconocido*

"*P* **is desconocido"** se evalúa a *cierto* si el predicado *P* se evalúa a *desconocido*

El resultado del predicado de la cláusula **WHERE** se toma como *falso* si se evalúa en *desconocido*

El total de todas las cantidades de prestamos

SELECT sum (*importe*)
FROM *prestamo*

La instrucción anterior ignora las cantidades nulas. El resultado es *null* si todas las cantidades son nulas

Subconsultas anidadas

SQL proporciona un mecanismo para las subconsultas anidadas.

Una subconsulta es una expresión **SELECT-FROM-WHERE** que se anida dentro de otra consulta.

Obtener todos los clientes que tengan una cuenta y un préstamo en el banco (intersect).

SELECT distinct *nombre_cliente*
FROM *prestatario*
WHERE *nombre_cliente* **in** (**SELECT** *nombre_cliente*
FROM *impositor*)

Obtener todos los clientes que tengan un préstamo en el banco pero que no tengan una cuenta en dicho banco (except-minus)

SELECT distinct *nombre_cliente*
FROM *prestatario*
WHERE *nombre_cliente* **not in** (**SELECT** *nombre_cliente*
FROM *impositor*)

Subconsultas anidadas

Nombre de los artículos adquiridos por el cliente 015

```
select nomArticulo as nombreArticulo, idArticulo
from articulo
where idArticulo in (select idArticulo
                     from compra
                     where idCliente='015');
```

Artículos que aún no se han comprado

```
select nomArticulo as nombre
from articulo
where idArticulo not in (select idArticulo
                         from compra);
```

Las subconsultas son más ineficientes que los join

Ejemplo de consulta

Obtener todos los clientes que tengan tanto una cuenta como un préstamo en la sucursal Navacerrada

SELECT distinct *nombre_cliente*
FROM prestatario, prestamo
WHERE *prestatario.número_préstamo= prestamo.número_préstamo* **AND**
nombre_sucursal = 'Navacerrada' **AND**
(*nombre_sucursal, nombre_cliente*) **in**
(SELECT *nombre_sucursal, nombre_cliente*
FROM *impositor, cuenta*
WHERE *impositor.número_cuenta = cuenta.número_cuenta*)

- **Se puede escribir la consulta anterior de forma mucho más simple. Se ha escrito así para ilustrar las características de SQL**

Comparación de conjuntos

Obtener los nombres de todas las sucursales que tengan activos mayores que al menos una sucursal situada en Barcelona.

```
SELECT distinct  T.nombre_sucursal
FROM sucursal as T, sucursal as S
WHERE  T.activo > S.activo AND
                 S.ciudad_sucursal = ‘ Barcelona ’
```

La misma consulta utilizando la clausula > some

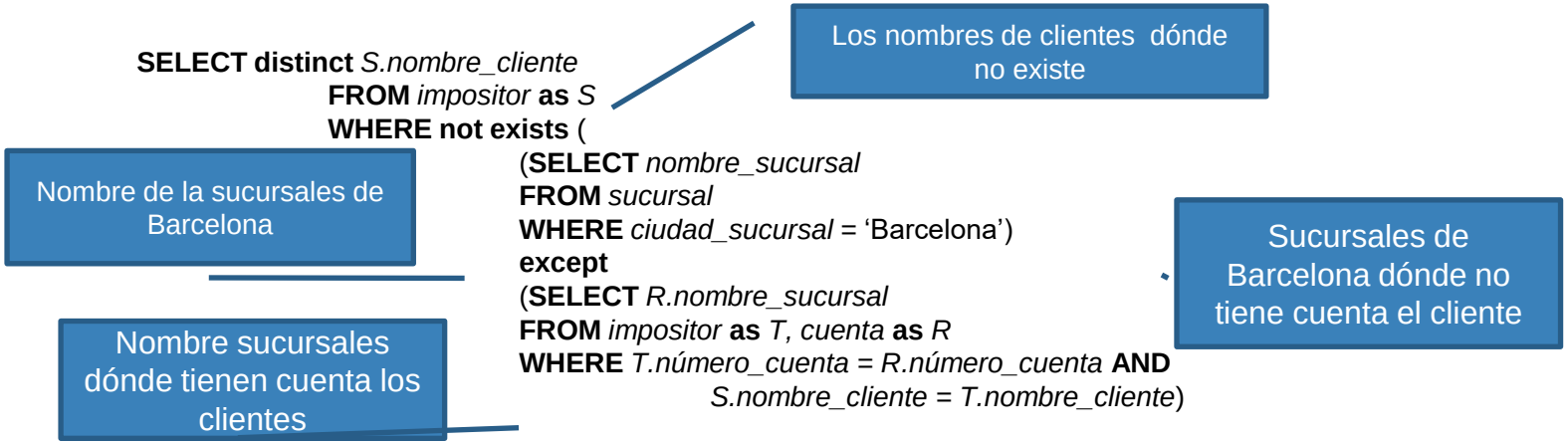
```
SELECT nombre_sucursal
FROM sucursal
WHERE activo > some(SELECT activo
                   FROM sucursal
                   WHERE ciudad_sucursal = ‘Barcelona’)
```

Obtener los nombres de todas las sucursales que tienen activos mayores que todas las sucursales situadas en Barcelona.

```
SELECT nombre_sucusal
FROM sucursal
WHERE activo > all   (SELECT activo
                      FROM sucursal
                      WHERE ciudad_sucursal = ‘Barcelona’)
```

Consulta ejemplo

Obtener todos los clientes que tengan una cuenta en todas las sucursales situadas en Barcelona.




```
SELECT distinct S.nombre_cliente
        FROM impositor as S
        WHERE not exists (
                (SELECT nombre_sucursal
                FROM sucursal
                WHERE ciudad_sucursal = 'Barcelona')
                except
                (SELECT R.nombre_sucursal
                FROM impositor as T, cuenta as R
                WHERE T.número_cuenta = R.número_cuenta AND
                        S.nombre_cliente = T.nombre_cliente)
```

Datos del cliente que ha comprado al menos una unidad de todos los artículos distintos que se han vendido

```
select idCliente,nombreC,count(distinct idArticulo) totalArticulos
from compra as c inner join cliente as cli on cli.idCliente=c.idCliente
group by idCliente
having totalArticulos = (select count(distinct idArticulo)
                         from compra);
```

Comprobación de ausencia de tuplas duplicadas

La construcción **unique** comprueba si una subconsulta tiene alguna tupla duplicada en sus resultados.

Obtener todos los clientes que sólo tengan una cuenta en la sucursal Navacerrada.

```
SELECT T.nombre_cliente
FROM impositor as T
WHERE unique (
        SELECT R.nombre_cliente
        FROM cuenta, impositor as R
        WHERE T.nombre_cliente = R.nombre_cliente AND
              R.número_cuenta= cuenta.número_cuenta AND
              cuenta.nombre_sucursal = ‘ Navacerrada’ )
```

Obtener todos los clientes que tengan al menos dos cuentas en la sucursal Navacerrada.

```
SELECT distinct T.nombre_cliente
FROM impositor as T
WHERE not unique (
    SELECT R.nombre_cliente
    FROM cuenta, impositor as R
    WHERE T.nombre_cliente = R.nombre_cliente AND
          R.número_cuenta = cuenta.número_cuenta  AND
          cuenta.nombre_sucursal  = ‘Navacerrada’)
```

Relaciones derivadas

SQL permite utilizar expresiones de subconsulta en la cláusula **FROM**

Obtener el saldo promedio de las cuentas en las que dicho saldo sea mayor de 1200€.

```
SELECT nombre_sucursal, saldo_medio
 FROM (SELECT nombre_sucursal, avg (saldo)
        FROM cuenta
        group by nombre_sucursal)
          as media_sucursal(nombre_sucursal, saldo_medio)
WHERE saldo_medio > 1200
```

No es necesario utilizar la cláusula **having**, puesto que se calcula la relación temporal (vista) resultado en la cláusula **FROM**, y los atributos de media_sucursal se pueden utilizar directamente en la cláusula **WHERE.**

Vistas

En algunos casos, no es deseable para todos los usuarios ver el modelo lógico completo (es decir, todas las relaciones actuales almacenadas en la base de datos).

Se utilizan para tres fines:

1. Prohibir el acceso a datos confidenciales
2. Simplificar la formulación de consultas complejas o repetitivas
3. Aumentar la independencia de los programas respecto a los datos

Una persona que necesita conocer un número de préstamo de un cliente pero no tiene necesidad de conocer el importe del préstamo. Esta persona debería ver una relación descrita en SQL como

(**SELECT** *nombre_cliente, número_prestamo*
FROM *prestatario, prestamo*
WHERE *prestatario.número_préstamo = prestamo.número_prestamo*)

Una **vista** proporciona un mecanismo para ocultar ciertos datos de la vista de ciertos usuarios.

Cualquier relación que no es del modelo conceptual pero se hace visible para el usuario como una “relación virtual” se nomArticuloina una **view**.

Definición de vista

Una vista se define utilizando la instrucción **create view** que tiene la forma

create view *v* **as** <expresión de consulta>

donde <expresión de consulta> es cualquier expresión de consulta legal de SQL. El nombre de la vista se representa por *v.*

Una vez definida la vista, su nombre puede utilizarse para referirse a la relación virtual que la vista genera.

La definición de vista no es lo mismo que la creación de una nueva relación mediante la evaluación de la expresión de consulta.

Una definición de vista permite el ahorro de una expresión para ser sustituida por consultas que utilizan esa vista.

Ejemplo vista

Gastos por cliente

```
create view gastoPorCliente as
select nombreC, sum(precio*numUnidades) total
from articulo as a INNER JOIN compra as p  INNER JOIN cliente as cli
        ON  idCliente=IdCliente and idArticulo=idArticulo
group by idCliente
order by nombreC desc;

select * from gastoPorCliente;
```

nombreC	total
Maribel Riocal	1060.00
Juan Antonio Sanz	540.00
Isabel Sanrio	5700.00
Angela Callejo	614.00

Vistas

Cliente que ha gastado más dinero en la tienda

```
create view gastoPorCliente(idCliente,nombreC,gasto) as
select idCliente,nombreC, sum(numUnidades*precio)
from articulo as a inner join compra as c on a.idArticulo=c.idArticulo
        inner join cliente as cli on cli.idCliente=c.idCliente
group by idCliente;


select nombreC, gasto
from gastoPorCliente
where gasto=(select max(gasto)
                    from gastoPorCliente);
```

Consultas de ejemplo

Una vista de las sucursales y sus clientes.

create view *todos_los_clientes* **as**
(**SELECT** *nombre_sucursal, nombre_cliente*
FROM *impositor, cuenta*
WHERE *impositor.número_cuenta = cuenta. número_cuenta*)
union
(**SELECT** *nombre_sucursal, nombre_cliente*
FROM *prestatario, prestamo*
WHERE *prestatario.número_cuenta = prestamo.número_cuenta*)

Averiguar todos los clientes de la sucursal de Navacerrada

SELECT *nombre_cliente*
FROM *todos_los_clientes*
WHERE *nombre_sucursal* = ‘Navacerrada’

Modificación de la base de datos– Borrado

Borrar todos los registros de cuentas de la sucursal Navacerrada

```
DELETE FROM cuenta
WHERE nombre_sucursal = 'Navacerrada'
```

```
set sql_safe_updates=0;
```

Borrar todas las cuentas de cada sucursal situada en la ciudad de Navacerrada.

```
DELETE FROM cuenta
WHERE nombre_sucursal in (SELECT nombre_sucursal
                          FROM sucursal
                          WHERE ciudad_sucursal = 'Navacerrada')
```

Borrar el registro de todas las cuentas con saldos inferiores a la media del banco.

```
DELETE FROM cuenta
 WHERE saldo < (SELECT avg (saldo)
                FROM cuenta )
```

- **Problema**: al borrar tuplas, el saldo medio cambia
- ***Solución utilizada en SQL:***

  1. Primero, calcular el saldo medio **avg** (saldo) de todas las tuplas que se van a borrar
  2. Después, borrar todas las tuplas encontradas antes (sin recalcular avg (saldo) o recomprobando las tuplas)

Modificación de la base de datos– Borrado

Borrar los clientes que cuando compran un artículo solo compran una unidad

```
delete from cliente
where idCliente in (select idCliente
                    from compra
                    where numUnidades =1 and
                          idCliente not in(select idCliente
                                           from compra
                                           where numUnidades>1));
```

Modificación de la base de datos– Inserción

Se proporciona como regalo a todos los clientes que tengan un préstamo en la sucursal Navacerrada, una cuenta de ahorro de 200€. Hacer que el número de préstamo sirva como número de cuenta de la nueva cuenta de ahorro

INSERT INTO *cuenta*
SELECT *número_prestamo, nombre_sucursal,* 200
FROM *prestamo*
WHERE *nombre_sucursal* = 'Navacerrada'

INSERT INTO *impositor*
SELECT *nombre_cliente, número_prestamo*
FROM *prestamo, prestatario*
WHERE nombre_sucursal = 'Navacerrada' **AND** *prestamo.número_cuenta= prestatario.número_cuenta*

La sentencia **SELECT FROM WHERE** se evalúa completamente antes de que ninguno de sus resultados se inserte en la relación (de otra forma las consultas como

insert into *tabla*1 **SELECT** * **FROM** *tabla*1

generarían problemas)

Modificación de la base de datos– Actualizaciones

Aumentar todas las cuentas con saldos por encima de 10.000€ con el 6%, todas las demás cuentas reciben un 5%.

Escribir dos instrucciones **update**:

update *cuenta*
set *saldo* = *saldo* ∗ 1,06
WHERE *saldo* > 10000

El orden es importante

update *cuenta*
set *saldo* = *saldo* ∗ 1,05
WHERE *saldo* ≤ 10000

Aumentar el precio de los artículos con precios inferiores a 50 euros en un 10% y los demás artículos aumentar el precio en un 5%

update *articulo*
set *precio* =*precio*∗ 1,05
WHERE *precio* > 50

update *articulo*
set *precio*=*precio*∗ 1,10
WHERE *precio* ≤ 50

Actualizaciones

Incrementar un 10% el precio de los artículos con menos de 5 que fueron vendidos en noviembre

```
set sql_safe_updates=0;
update articulo set precio=precio*1.1, descuento=50
where unidades<5 and idArticulo in (select idArticulo
                                    from compra
                                    where month(fecCompra)=11);
```

Actualización de una vista

Crear una vista de todos los datos de prestamos en la relación préstamo, ocultando el atributo importe

create view *sucursal_préstamo***as**
SELECT *número_prestamo, nombre_sucursal,*
FROM *prestamo*

Añadir una tupla nueva a sucursal_prestamo

insert into *sucursal_préstamo* **values** (‘P-37’,‘Navacerrada’)

Esta inserción se debe representar mediante la inserción de la tupla
(‘P-37’, ‘Navacerrada’, *null*)
dentro de la relación *prestamo*

Actualización de una vista

Algunas actualizaciones de vistas son difíciles o imposibles de traducir en relaciones de la base de datos

create view *v* **as**

SELECT *nombre_sucursal* **FROM** *cuenta*

insert into *v* **values (**‘ *Navacerrada*’**)**

Otras no se pueden traducir de forma única

insert into *todos_los_clientes* **values** *(*‘Navacerrada’*,* ‘Juan’**)**

¡Hay que elegir préstamo cuenta y crear un nuevo número de prestamo/cuenta!

La mayor parte de las implementaciones de SQL permiten actualizar sólo vistas simples (sin agregados) definidas sobre una sola relación.

Asertos (en workbench no se pueden definir)

Un *aserto* es un predicado que expresa una condición que se desea que la base de datos satisfaga siempre**.**

Un aserto en SQL tiene la forma

create assertion <nombre-aserto > **check** <predicado>

Cuando se crea un aserto, el sistema comprueba su validez, y la comprueba de nuevo en cada actualización que puede violar el aserto

Esta prueba puede introducir una cantidad considerable de sobrecarga; por lo tanto **se deben utilizar los asertos con mucha cautela**.

El aserto para todo *X, P(X)* se consigue en un modo indirecto utilizando no existe *X* tal que *no P(X)*

Ejemplo de aserto

Cada préstamo tiene al menos un prestatario que mantiene una cuenta con un saldo mínimo o 1.000,00€

```
create assertion restricción-saldo check
(not exists (
    SELECT * FROM prestamo
        WHERE not exists (
         SELECT *
            FROM prestatario, impositor, cuenta
            WHERE prestamo.número-préstamo= prestatario.número-prestamo
                    AND prestatario.nombre-prestatario =
                            impositor.nombre-cliente
                    AND impositor.número-cuenta =
                            cuenta.número-cuenta
                    AND cuenta.saldo >= 1000)))
```

Ejemplo de aserto

La suma de todas las cantidades de préstamo de cada sucursal debe ser menores que la suma de todos los saldos de las cuentas de la sucursal.

create assertion *restricción-suma* **check**
(not exists (SELECT * FROM *sucursal*
WHERE (SELECT sum*(importe)* **FROM** *prestamo*
WHERE *prestamo.nombre-sucursal = sucursal.nombre-sucursal)*
>= **(SELECT sum***(importe)* **FROM** *cuenta*
WHERE *prestamo.nombre-sucursal = sucursal.nombre-sucursal)))*

Trigger

- Un trigger o disparador en una Base de Datos, es un objeto asociado a una tabla se ejecuta cuando se cumple una condición establecida en la tabla, se activa cuando ocurre un evento en particular para la tabla. Esta debe ser una tabla permanente, no puede ser una vista.

- Dependiendo de la base de datos, los triggers pueden ser de inserción (INSERT), actualización (UPDATE) o borrado (DELETE).

- Un trigger no se llama directamente, la acción definida en el disparador se ejecuta automáticamente cuando se produce una inserción, actualización o borrado.

- No se aplican a los datos almacenados en la base de datos antes de su definición. Sólo se aplican cuando una vez creados se ejecutan comandos que manipulan las tablas sobre las que están definidos.

- Cuando se crean están activos por defecto.

Ventajas de los trigger

- Restricciones para la entrada de datos.
- Los cambios de un disparador se reflejan automáticamente en todas las aplicaciones que tienen que ver con la tabla, sin necesidad de volver a compilar o de unir tablas de nuevo.
- Logs automáticos de cambios en las tablas, se puede crear un trigger que se active cada vez que se produzcan cambios en la tabla.
- Mantener la integridad de la base de datos. Forzar reglas de integridad que son difíciles de de definir mediante restricciones.
- Calcular atributos derivados.

Se recomienda el uso moderado de los triggers

```
CREATE TRIGGER nombreDisp momentoDisp eventoDisp
    ON nombreTabla
    FOR EACH ROW  sentenciaDisp
```

momentoDisp es el momento en que el disparador entra en acción. Puede ser BEFORE (antes) o AFTER

eventoDisp indica la clase de sentencia que activa al disparador. Puede ser INSERT, UPDATE, o DELETE.

Por ejemplo, un disparador BEFORE para sentencias INSERT podría utilizarse para validar los valores a insertar.

No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y sentencia.

NO se pueden tener dos disparadores BEFORE UPDATE.

SI es posible tener los disparadores BEFORE UPDATE y BEFORE INSERT o BEFORE UPDATE y AFTER UPDATE.

sentenciaDisp es la sentencia que se ejecuta cuando se activa el disparador.

Si se desean ejecutar múltiples sentencias, deben colocarse entre BEGIN ... END, el constructor de sentencias compuestas. Esto además posibilita emplear las mismas sentencias permitidas en rutinas almacenadas.

Identificadores new y old

- Para relacionar el trigger con columnas especificas de una tabla usaremos los identificadores OLD y NEW.
- OLD indica el valor antiguo de la columna y NEW el valor nuevo que pudiese tomar. Por ejemplo: OLD.idArticulo o NEW.idArticulo.
- Si usamos la sentencia **UPDATE** se refiere a un valor **OLD y NEW**, ya que modifica registros existentes por los valores.
- Si usamos **INSERT** sólo usa **NEW,** ya que lo que hacemos es insertar nuevos valores a las columnas.
- Y con **DELETE** utilizamos **OLD** debido a que borra valores existentes.
- Sí en un Trigger AFTER se ejecuta una sentencia UPDATE, no es lógico editar valores nuevos NEW, porque el evento ya ocurrió. Lo mismo pasa con la sentencia INSERT, el Trigger tampoco podría hacer referencia a valores NEW, ya que los valores que en algún momento fueron NEW, han pasado a ser OLD.

El delimitador DELIMITER

- Un delimitador es una instrucción que permite cambiar el símbolo que indica el fin de la sentencia SQL. Se escribe principalmente cuando se escriben procedures y trigger, dónde necesitamos escribir múltiples sentencias de SQL dentro de un bloque.

- MySQL usa el **punto y coma (;)** como delimitador para indicar el final de una sentencia. Sin embargo, cuando creamos procedimientos almacenados o trigger, estos contienen múltiples sentencias SQL dentro de un bloque BEGIN ... END, lo que puede generar errores si el **;** no se maneja correctamente.

- Para evitar esto, usamos DELIMITER para cambiar temporalmente el delimitador a otro símbolo, suele utilizarse // o $$.

Ejemplos

```
CREATE TRIGGER
sumarHabitacionesAIhotel
AFTER INSERT ON habitación
FOR EACH ROW
UPDATE hotel
SET numHab=numHab + 1
WHERE num = NEW.nomHotel;
```

```
delimiter //
CREATE TRIGGER
restarHabitacionesAD
AFTER DELETE ON habitacion
FOR EACH ROW UPDATE hotel
SET numHab = numHab – 1
WHERE num = OLD.nomHotel;
delimiter ;
```

```
DELIMITER $$
drop trigger if exists actualizarProductoAI ;
CREATE TRIGGER actualizarProductoAI
AFTER INSERT ON compra
FOR EACH ROW
BEGIN
 update articulo
 set unidades=unidades-NEW.numUnidades
 where articulo.idArticulo=new.idArticulo;
END$$
DELIMITER ;

-- actualiza el número de unidades cada vez que hay una compra de un artículo
select * from articulo;

INSERT INTO compra VALUES('009', '0002', '2019/11/09',6);
SELECT * FROM articulo;
```

Ejemplo, verifica si tenemos artículos disponibles antes de realizar una compra

```
use tiendainformatica;
DELIMITER //
CREATE TRIGGER VerificarInventarioBI
BEFORE INSERT ON compra
FOR EACH ROW
BEGIN
   DECLARE disponible INT;
  SELECT unidades INTO disponible
   FROM articulo
   WHERE idArticulo = NEW.IDArticulo;
   IF NEW.numUnidades > disponible
    THEN  SIGNAL SQLSTATE '45000'   SET MESSAGE_TEXT = 'No hay suficientes unidades
disponibles para la venta';
  END IF;
END;//
DELIMITER ;
```

Por qué no abusar de los trigger

Los **triggers** son útiles para automatizar tareas en la base de datos, pero su uso excesivo puede generar **problemas de rendimiento, mantenimiento y depuración**. A continuación, te explico las razones principales para no abusar de ellos.

1. Dificultan la depuración y el mantenimiento

- Los TRIGGER **se ejecutan automáticamente**, lo que puede hacer que sea difícil identificar errores.
- Si hay errores en los datos, no siempre es evidente que fueron causados por un trigger.
- No es fácil **rastrear ni depurar** el flujo de ejecución en comparación con procedimientos almacenados o código en la aplicación.

Ejemplo: Si un TRIGGER actualiza otra tabla y esa tabla tiene otro TRIGGER, puede ser complicado encontrar la causa de un problema.

Por qué no abusar de los trigger

2. Impacto en el rendimiento

- Se ejecutan en **cada operación INSERT, UPDATE o DELETE**, lo que puede ralentizar el sistema.
- Los triggers pueden afectar a **gran cantidad de registros** sin que el usuario lo note.

📌 **Ejemplo:** Un trigger que actualiza un saldo en cada transacción bancaria puede afectar el rendimiento si hay muchas operaciones simultáneas.

3. Complejidad en la lógica de negocio

- La lógica implementada en triggers **puede volverse difícil de entender** con el tiempo.
- Es más fácil y claro manejar la lógica de negocio en el código de la aplicación o en procedimientos almacenados.
- Si muchos triggers interactúan entre sí, pueden generar **cascadas de eventos** difíciles de controlar.

📌 **Ejemplo:** Un trigger que inserta datos en una tabla, que a su vez activa otro trigger, puede llevar a una **ejecución en cadena** difícil de manejar.

Por qué no abusar de los trigger

4. No se activan en operaciones LOAD DATA o ALTER TABLE

- Si usas LOAD DATA INFILE para importar datos, los triggers **no se ejecutan**, lo que puede causar inconsistencias.
- Al modificar una tabla (ALTER TABLE), algunos cambios pueden hacer que los triggers sean incompatibles o requieran ajustes manuales.

📌 **Ejemplo:** Si un trigger debe actualizar un log de auditoría pero la carga de datos masiva lo omite, la auditoría se vuelve incompleta.

5. Portabilidad y compatibilidad limitada

- No todos los motores de bases de datos manejan los triggers de la misma manera.
- Si necesitas migrar a otro sistema de base de datos, los triggers pueden requerir **reescritura o adaptación**.

📌 **Ejemplo:** Un trigger diseñado en MySQL podría no funcionar en PostgreSQL sin cambios en su sintaxis.

Usar **triggers con moderación** ayuda a mantener la base de datos eficiente, clara y fácil de depurar

Procedimientos almacenados

- Un procedimiento almacenado es una serie de instrucciones SQL que realizan una tarea determinada, que tiene un nombre, se puede guardar y reutilizar.

- Pueden tener **parámetros,** para pasar información al procedimiento o para sacar información del procedimiento. Pueden ser:

  IN: se utiliza por defecto, cuando se invoque al procedimiento tendrá que pasar un argumento de este tipo. El procedimiento trabaja con una copia de su valor

  OUT: el valor del parámetro puede ser cambiado en el cuerpo del procedimiento y su valor modificado será devuelto a quién ha invocado este procedimiento

  INOUT:

- Para crearlo es necesario tener los permisos INSERT y DELETE

Procedimientos almacenados

```
CREATE PROCEDURE nombreProcedimiento(parámetros)
BEGIN
    instrucciones;
END;
```

Cláusula 'delimiter’

Un procedimiento almacenado puede tener muchos comandos SQL entre las palabras claves begin y end, por ello debemos decir de alguna forma a MySQL que no ejecute esos comandos.

Utilizamos el comando 'delimiter' cambiando el caracter ';' como fin de instrucción.

Se utilizan los delimitadores $$, //,…

Procedimientos almacenados

```
-- con variables de usuario, este tipo de varibles no necesitan declaración, van precedidas del carácter @
 -- este tipo de variables pueden ser texto, fechas y números
drop procedure if exists conVariables;
delimiter $$
create procedure conVariables()
begin
set @v1=@v1*2;
end $$
delimiter ;

 set @v1=100;
 call conVariables();
 select @v1;
```

Ejemplo parámetro de entrada, parámetro por valor

```
-- con parámetro IN ,el procedimiento trabaja con una "copia" del parámetro que recibe
 -- no modifica nada, es como los parámetros por valor

 drop procedure if exists conParametro;
delimiter $$ create procedure conParametro(in x int)
begin
  set x=2*x;
end $$
delimiter ;

set @xVar=100;
call conParametro(@xVar);
select @xVar; -- no se ha modificado porque era como un parámetro por valor
```

Ejemplo con parámetro por referencia

```
-- con parámetro OUT ,el procedimiento modifica el parámetro que recibe
 -- es como los parámetros por referencia

drop procedure if exists conParametroRef;
delimiter $$
create procedure conParametroRef(OUT y int)
begin
 set y= 2;
end $$
delimiter ;

set @yVar=100;
call conParametroRef(@yVar);
select @yVar;
 -- se ha modificado porque era como un parámetro por referencia
```

Ejemplo con if

```
drop procedure if exists conIf;
delimiter $$
create procedure conIf()
begin
declare edad int;
   if edad <=18     then select 'menor de edad';
  else select 'mayor de edad';
end if;
end $$
delimiter ;

call conIf();
-- sale el else porque no le hemos dado un valor,
 -- será por defecto null, podría hacer dos if, para subsanar estos
errores
```

Ejemplo con if

```
drop procedure if exists conIfParametro;
delimiter $$
create procedure conIfParametro(edad int)
Begin
  if edad <=18     then select 'menor de edad';
  else select 'mayor de edad';
end if;
end $$
 delimiter ;
call conIfParametro(20);
-- sale el else porque no le hemos dado un valor, será por defecto null
```

Procedimientos almacenados

```
create procedure numeroProductos()
  select count(*) from producto ;

create procedure actualizaProducto(nuevoPrecio numeric(4,2), codigoP int)
  update producto set precio=nuevoPrecio where idProducto=1;
```

```
USE tiendainformatica;
DROP procedure IF EXISTS descuentoNavidad;
DELIMITER $$
USE tiendainformatica $$
CREATE PROCEDURE descuentoNavidad ()
BEGIN
update  articulo
set descuento= 50
where unidades >15;
END$$
DELIMITER ;
call descuentoNavidad();
select * from articulo;
```

```
DELIMITER //

CREATE PROCEDURE Ejemplo ()
-- procedimiento sin parámetros para
saludarnos
BEGIN
   SELECT ‘¡Hola!’;
END
Delimiter ;
```

```
DELIMITER //

CREATE PROCEDURE procCASE (IN param1 INT)
BEGIN
   DECLARE variable1 INT;
   SET variable1 = param1 + 1;

   CASE
      WHEN variable1 = 0 THEN
         INSERT INTO table1 VALUES (param1);
      WHEN variable1 = 1 THEN
         INSERT INTO table1 VALUES (variable1);
      ELSE
         INSERT INTO table1 VALUES (99);
   END CASE;

END //
```

Procedimientos almacenados

```
create table ninos(edad int, nombre varchar(50));
create table adultos(edad int, nombre varchar(50));


 delimiter //
 create procedure introducePersona(in edad int, in nombre varchar(50))
  begin
        if edad < 18 then insert into ninos values(edad,nombre);
         else insert into adultos values(edad,nombre);
        end if;
  end; //

   call introducePersona(25,”JoseManuel”);
```

Función para calcular el DNI

```
DROP FUNCTION IF EXISTS letraDNI;
  delimiter //
  create function letraDNI (dni varchar(9))
    returns char(1)
  deterministic
  no sql
   begin
     return substr('TRWAGMYFPDXBNJZSQVHLCKE',mod(dni,23)+1,1);
   end//
 delimiter ;

select letraDNI('00280432');-- Q
```

Trigger que utiliza la función anterior

```
/*Este trigger se activa antes de insertar en la tabla persona,
 comprobará si el dni que nos dan tiene letra y si no la tiene,
 llamará a la función para concatenar la letra al número de dni
 */
Drop trigger if exists personaBI;
 delimiter //
create trigger personaBI
 before insert on persona
 for each row
 begin
  if right(new.dni,1) not like '%[a-zA-Z]%'
       then   set new.dni=concat(new.dni,letraDNI(new.dni));
 end if;
end //
```

Hay más procedimientos almacenados en los scripts que adjunto.

Comparación entre TRIGGER y PROCEDURE

Característica	TRIGGER	PROCEDURE
Ejecución	Automática cuando ocurre `INSERT`, `UPDATE`, `DELETE`	Se ejecuta manualmente con `CALL`
Control del flujo	Se ejecuta en segundo plano, sin intervención del usuario	Se puede llamar en el momento exacto deseado
Depuración	Difícil de depurar (ocurre automáticamente)	Más fácil de depurar con mensajes y validaciones
Impacto en el rendimiento	Puede afectar el rendimiento si se ejecuta con frecuencia	Mejor rendimiento porque se ejecuta solo cuando es necesario
Escalabilidad	Puede generar problemas en sistemas grandes con muchas reglas	Más flexible y fácil de mantener
Portabilidad	Depende del motor de base de datos	Más portable y compatible con otros sistemas

ntic master
revolucionamos la comunicación

UNIVERSIDAD COMPLUTENSE MADRID