BASES DE DATOS I

Zoé Agustina Tira

MÓDULO 1 -INTRODUCCIÓN A BASE DE DATOS

C1A - ¡HOLA, MUNDO!

¿Por qué Bases de datos?

Las bases de datos surgieron por seguridad, control y escalabilidad. Permite modelar, guardar y consultar datos.

Se crea una base de datos a partir del requerimiento de una aplicación.

C2A - INTRODUCCIÓN A BASE DE DATOS

¿Qué es una base de datos?

Conjunto de datos almacenados pertenecientes a un mismo contexto organizados para un propósito específico. Una base de datos nos permite:

- Almacenar (agregar, modificar y eliminar) datos.
- Acceder a los datos.
- Manipularlos y combinarlos.
- Analizar datos.
- Entre otras cosas más.

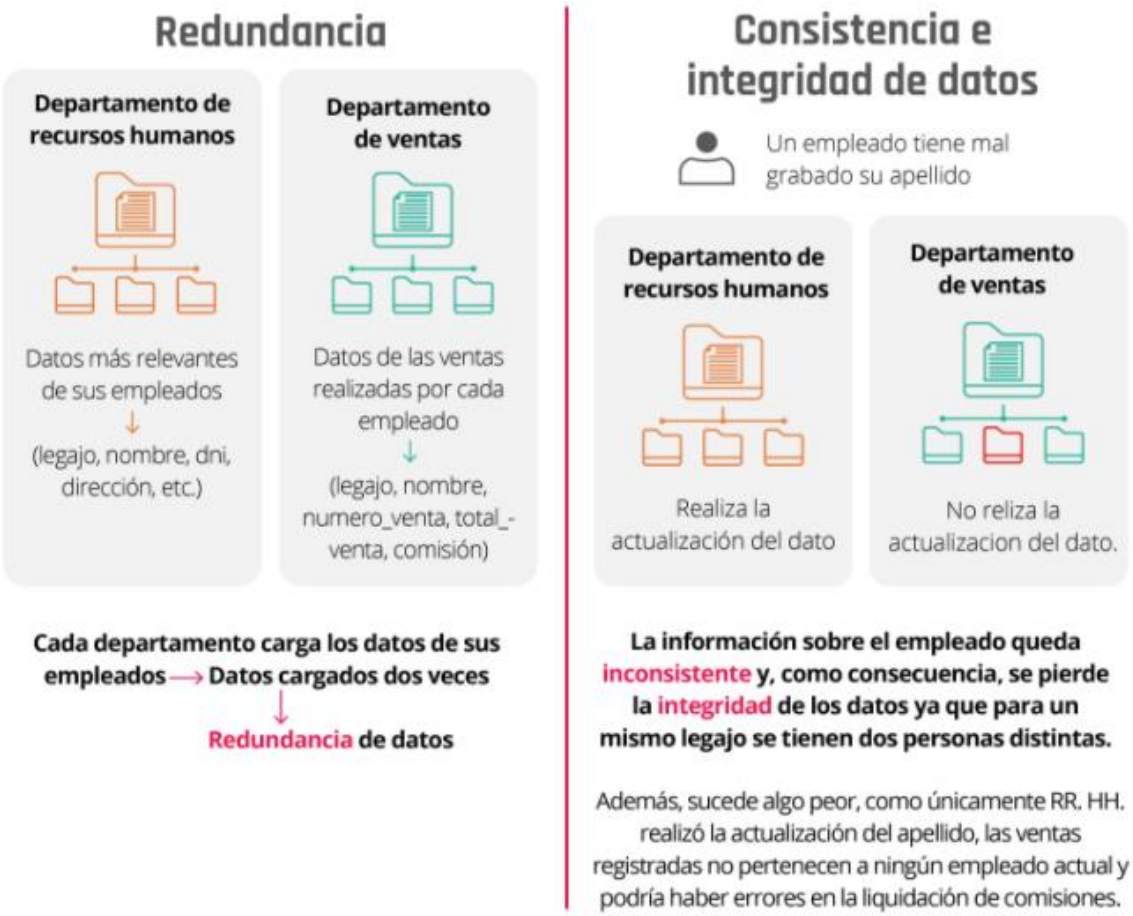
Las bases de datos evitan redundancia de datos y conserva la integridad de los mismos.

Bases de datos las hay de dos tipos: **relacionales y no relacionales**.

Bases de datos vs. Archivos planos

Un archivo está compuesto por registros y cada registro está compuesto por un conjunto de campos.

Inconvenientes con el uso de archivos planos



Motores de Bases de datos

También conocido como sistema de gestión de base de datos, es una capa de software cuyo principal objetivo es almacenar, procesar y proteger los datos. Posee distintas capacidades como la consistencia de datos, seguridad, integridad, respaldo, acceso concurrente y tiempo de respuesta.
Las bases de datos realizan chequeos de:

- Integridad.
- Coherencia de los datos.
- Redundancia.

Todas las bases de datos tienen un estándar en común, SQL. Con este lenguaje podemos crear bases de datos, modificarlas, escribir datos y consultarlos.

Modelo de base de datos

¿Cómo se comienza a desarrollar una base de datos?

Se inicia con un requerimiento o necesidad de una aplicación o sistema, lo importante es tener en claro qué es lo que se tiene que almacenar, y es por ello que este requerimiento tiene que estar lo más detallado posible.
Luego se inicia la etapa de modelado. Este documento nos permite validar que entendimos el requerimiento y empezar a pensar cómo lo vamos a implementar.
Una vez implementada la base de datos, este documento ¡no se elimina! Sino que sirve de documentación para volver a consultar y entender el porqué se implementó la base de datos de una determinada forma.

Modelado de datos

Un modelo es un conjunto de herramientas conceptuales para describir datos, sus relaciones, su significado y sus restricciones de consistencia.
El **modelado de datos es una manera de estructurar y organizar los datos** para que se puedan utilizar fácilmente por las bases de datos.

Beneficios

- Registrar los requerimientos de datos de un proceso de negocio.
- Se puede descomponer un proceso complejo en partes.
- Permite observar patrones.
- Sirve de plano para construir la base de datos física.
- El modelo de datos ayuda a las empresas a comunicarse dentro y entre las organizaciones.
- Proporciona soporte ante los cambios de requerimientos del negocio o aplicaciones.

Tipos de modelos

Existen 3 tipos de modelos que podrían implementarse:

Conceptual	Lógico	Físico
Es un modelo con un diseño muy general y abstracto, cuyo objetivo es explicar la visión general del negocio o sistema.	El modelo lógico es una versión completa que incluye todos los detalles acerca de los datos. Explica qué datos son importantes, su semántica, relaciones y restricciones. Explica el “qué”.	Es un modelo que implementa el modelo lógico. Es un esquema que se va a implementar dentro de un sistema de gestión de bases de datos. Explica el “cómo”.

Modelo lógico vs Modelo físico

Modelo lógico

Describe los datos con el mayor detalle posible, independientemente de cómo se implementarán físicamente en la base de datos.
Describe los elementos importantes del negocio, qué significa cada objeto, su nivel de detalle, cómo se relacionan entre sí y sus restricciones.

Características

- Se definen cuáles son los conceptos importantes sobre los que hay que almacenar información. Estos elementos se denominan **entidades**.
- Se especifican todos los **atributos** para cada una de las entidades.
- Se conectan las entidades mediante **relaciones**.
- Se especifica la clave principal para cada entidad.
- Se especifican las claves externas (claves que identifican la relación entre diferentes entidades).
- La normalización ocurre en este nivel.

Modelo físico

Representa **cómo** se construirá el modelo en la base de datos.
Un modelo de base de datos física muestra todas las estructuras de tablas, incluidos el nombre de columna, el tipo de datos de columna, las restricciones de columna, la clave principal, la clave externa y las relaciones entre las tablas.

Características

- Especificación de todas las tablas y columnas.
- Las claves externas se usan para identificar relaciones entre tablas.
- La desnormalización puede ocurrir según los requisitos del usuario.

- Las consideraciones físicas pueden hacer que el modelo de datos físicos sea bastante diferente del modelo de datos lógicos.
- El modelo físico puede diferir de un motor de bases de datos a otro.

Pasos para el diseño del modelo de datos físicos

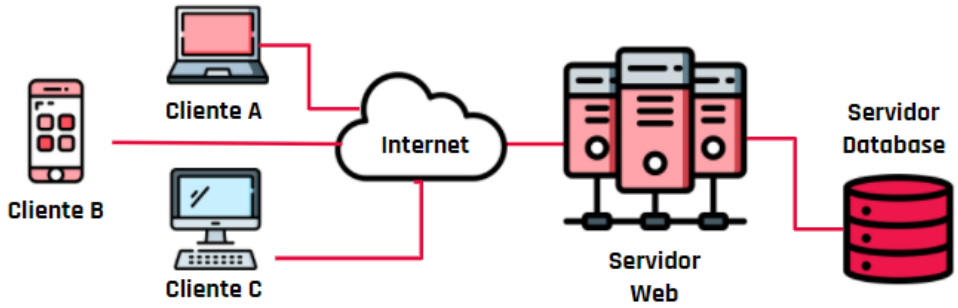
- Convertir entidades en tablas.
- Convertir relaciones en claves externas.
- Convertir atributos en columnas.
- Modificar el modelo de datos físicos en función de las restricciones / requisitos físicos.

Resumen

MODELO LÓGICO	MODELO FÍSICO
• Describe el "qué". • Explica el negocio. • Es independiente de la implementación. • Responsable: el analista.	• Describe el "cómo". • Explica el técnicamente cómo se van a almacenar los datos. • Explica la implementación en el sistema de gestión de bases de datos. • Responsable: administrador de bases de datos o simil.

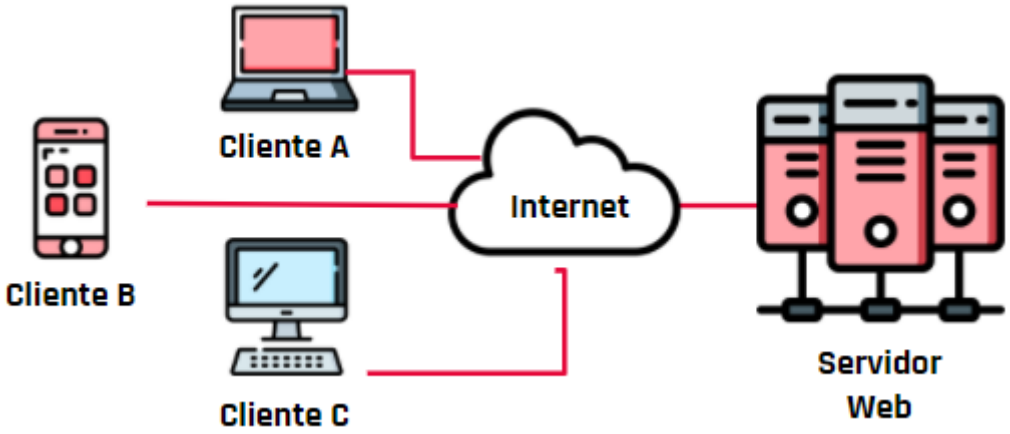
Cliente/Servidor

Las bases de datos se implementan bajo una **arquitectura Cliente - Servidor**.
Esta arquitectura tiene dos partes claramente diferenciadas, por un lado, la parte del **servidor** y, por otro, la parte de **cliente o grupo de clientes**.



Esto se realiza así porque:

- El **servidor**, normalmente, es una "máquina" bastante potente con un hardware y software específico que actúa de depósito de datos y funciona como un sistema gestor de base de datos o aplicaciones.
- Los **clientes** suelen ser estaciones de trabajo que solicitan varios servicios al servidor. Son los que necesitan los servicios del servidor. Físicamente se pueden ver como distintas computadoras, celulares y dispositivos que se conectan con el servidor.



¿Por qué es importante entender la arquitectura?

- Porque el servidor, va a ser el motor de bases de datos. En nuestro caso, **MySQL Server**.
- Nuestro cliente van a ser las aplicaciones que consulten o almacenen los datos. Por ejemplo, **MySQL Workbench** o, en el futuro, la aplicación que desarrollemos que va a interactuar con la base de datos.

MÓDULO 2 - MODELADO DE BASES DE DATOS

C4A – ENTIDADES

Modelos de bases de datos

Es una colección de herramientas conceptuales para describir datos, relaciones entre ellos, semántica asociada a los datos y restricciones de consistencia.

Objetivos

Representar mediante abstracciones del mundo real toda la información necesaria para el cumplimiento de los fines.

Modelos

1. Modelo conceptual basado en objetos: Se utiliza para la representación de la realidad No comprometida con ningún entorno informático. Sería el **Modelo Entidad-Relación** propiamente dicho.
2. Modelo lógico basado en objetos: determinan algunos criterios de almacenamiento y de operaciones de manipulación de los datos dentro de un entorno informático.

Modelo entidad-relación

Se basa en una percepción del mundo real, que consiste en un conjunto de objetos básicos llamados entidades y de relaciones entre ellos. Se emplea para interpretar, especificar y documentar los requerimientos para los sistemas de bases de datos.
Es un método de representación abstracta del mundo real centrado en las restricciones o propiedades lógicas de una base de datos.

Entidades

Es un objeto, real o abstracto, acerca del cual se recoge información de interés para la base de datos.

Tipos

- Entidades fuertes: tienen existencia por sí mismas. (alumnos, empleados, departamento.)
- Entidades débiles: dependen de otra entidad para su existencia (hijos de empleados)

Se define como **ocurrencia** de entidad al conjunto de datos para una entidad en particular.
Por ejemplo:

125, Juan Pérez, casado, 23 años.

Cada entidad tiene propiedades particulares llamadas atributos.

Atributos

Describen las características de una entidad.
Por ejemplo:

Entidad: clientes
Atributos: legajo, nombre, domicilio, etc.

Tipos

1. Atributo con simple valor: Cuando un atributo tiene un solo valor para una identidad.
2. Atributo multivalor: Cuando un atributo tiene una serie de valores para identificarse.
3. Atributos derivados: Cuando los valores de un atributo son afines y el valor para este tipo de atributo se puede derivar de los valores de otros atributos.
4. Atributo clave: Las entidades pueden contener un atributo que identifica cada una de las ocurrencias de la entidad. Es decir, usualmente contienen un atributo que diferencia los ítems entre sí.
5. Atributos nulos: Se usa cuando una entidad no tiene valor para un atributo o que el valor es desconocido.

Claves

Tipos

- Clave candidata
- Clave primaria
- Superclave

Clave candidata

Se compone por uno o más atributos cuyos valores **identifican unívocamente** a cada **ocurrencia** de la entidad, sin que ningún subconjunto de ellos pueda realizar esta misma función. Una clave candidata es una **posible clave primaria**. Pueden definirse varias claves candidatas para luego seleccionar la más adecuada.

Clave primaria

Está compuesta por uno o más atributos cuyos valores **identifican unívocamente** a cada ocurrencia de la entidad. No pueden contener valores **nulos** ni **repetidos**. Esta clave es una de aquellas que anteriormente se seleccionaron como candidata.

Superclave

Es el conjunto de uno o más **atributos** que, tomados **colectivamente**, permiten identificar de forma **unívoca** a la **ocurrencia** de una entidad. Se utiliza generalmente en las **tablas de relación**, este concepto se desarrollará en las próximas clases.

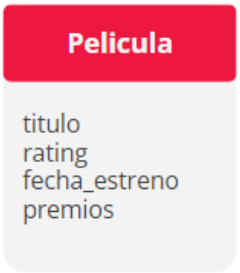
Entidades y Atributos

Entidad

Dentro de nuestro sistema tendremos **entidades** y para integrarlos en nuestro diagrama los representaremos usando un rectángulo.
Estas entidades son todos los objetivos sobre los cuales tenemos un interés de almacenar información.

Atributos

Son las características que van a definir a cada entidad.



Convención de nombres

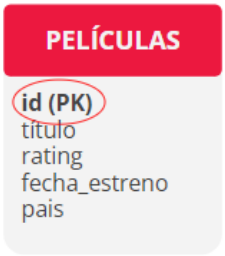
En los nombres de entidades y atributos siempre se debe utilizar **sustantivos** en **singular** o plural. No se puede utilizar **eñes**, **espacios** ni **acentos**. Si el nombre se compone por más de una palabra, se deben reemplazar los espacios con guiones bajos "_" o eliminar dicho espacio y colocar una mayúscula en la inicial de cada palabra (CamelCase).
Los nombres se distinguen entre mayúsculas y minúsculas solo si el sistema operativo posee un sistema de archivo sensible al tipo (Windows no, algunas versiones de Linux y Unix sí)

Clave primaria

La Primary Key es un campo que identifica a cada fila de una tabla de **forma única**. No puede haber dos filas en una tabla que tengan la misma PK.
Para identificar la clave primaria en una entidad, podemos escribir el atributo en **negrita** seguido de las iniciales **PK** entre paréntesis.

id	título	rating	fecha_estreno	premios
1001	Pulp Fiction	9.8	1995-02-16	10
1002	Kill Bill Vol. 1	9.5	2003-11-27	25

En este caso los id de las películas no se pueden repetir.



¿Entidades o atributos?

Con algunos atributos vamos a tener la duda si es un atributo o una entidad. Por ejemplo, el teléfono de un cliente. De qué depende:

- Del escenario que se modela.
- La semántica asociada con el atributo en cuestión.

Por ejemplo, para un *call center* es importante registrar todos los posibles teléfonos del cliente. ''Teléfonos'' es una entidad relevante dentro de su modelado.

Datos

Son los posibles valores que pueden tener los atributos. Nos permiten entender si son datos numéricos, textos, fechas, si tienen un formato en particular, si son obligatorios u opcionales.
No se modelan, pero nos permite entender mejor las entidades.

C5A – DATOS

Tipos de datos

Los datos o atributos de cada registro de una tabla tienen que ser de un tipo de dato concreto. Cuando diseñamos una base de datos tenemos que pensar qué tipo de datos requerimos para nuestro modelo.
Cada tipo de dato tiene un tamaño determinado y cuanta más precisión apliquemos en su definición, más **rápido** y **performante** va a funcionar MySQL.

Datos de tipo texto

Almacenan datos **alfanuméricos** y **símbolos**.

Char(n)

n → 1 a 255 caracteres.

Se recomienda utilizar en cadenas de texto de **longitud poco variable**.

Varchar(n)

n → 1 a 21.845 caracteres.

Se recomienda utilizar en cadenas de texto de **longitud muy variable**.

Nota: La letra ***n*** indica la **longitud máxima de caracteres** a utilizar.

Tinytext

0 a 255 caracteres.

Mediumtext

0 a 16.777.215 caracteres.

Text

0 a 4.294.967.295 caracteres.

Longtext

0 a 18.446.744.073.709.551.615 caracteres.
La longitud máxima permitida depende de la configuración del protocolo cliente-servidor y la memoria disponible.

Datos de tipo numérico

Tinyint

-128 a 127 (sin signo de 0 a 255)

Smallint

-32.768 a 32.767 (sin signo de 0 a 65.535)

Mediumint

-8.388.608 a 8.388.607 (sin signo de 0 a 16.777.215)

Int

-2.147.483.648 a 2.147.483.647 (sin signo de 0 a 4.294.967.295)

Bigint

-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (sin signo de 0 a 18.446.744.073.709.551.615)

Datos de tipo numérico decimal

Float(n,d)

Almacenan números de coma flotante pequeño.
Tienen precisión simple para la parte decimal (máx. 7 dígitos).

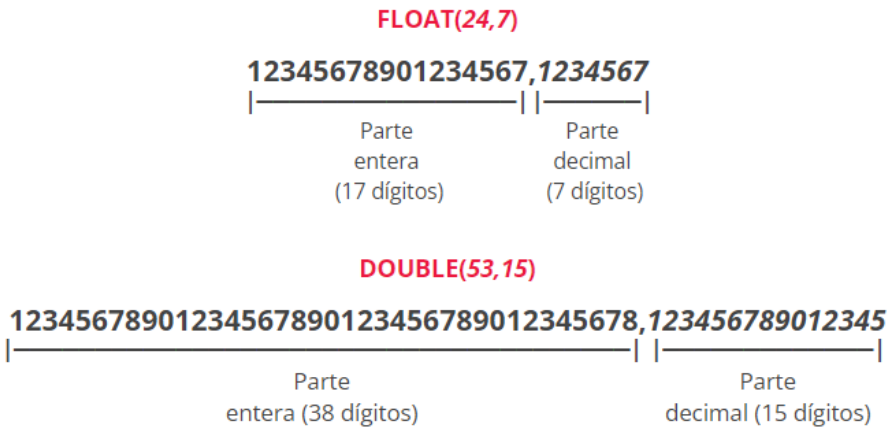
n → 1 a 24 dígitos (incluyendo la parte decimal).
d → 0 a 7 dígitos dependiendo de cuánto se asigne en n.

Double(n,d)

Almacenan números de coma flotante grande.
Tienen precisión doble para la parte decimal (máx. 15 dígitos).

n → 25 a 53 dígitos (incluyendo la parte decimal).
d → 0 a 15 dígitos dependiendo de cuánto se asigne en n.

Nota: Las letras **n** y **d** indican la longitud máxima de dígitos a utilizar.



Datos de tipo fecha

MySQL no comprueba de una manera estricta si una fecha es válida o no.

Date

Almacena solamente la fecha en formato **YYYY-MM-DD**.
Valores permitidos: '0001-01-01' a '9999-12-31'.
La fecha se debe colocar entre **comillas** simples o dobles y se separa por **guiones**. Ejemplo: '2021-05-15'.

Time

Almacena solamente la hora en formato **HH:MM:SS**.
Valores permitidos: '00:00:00' a '23:59:59'.
La hora se debe colocar entre **comillas** simples o dobles y se separa por **dos puntos**. Ejemplo: '11:50:55'.

Datetime

Almacena la fecha y hora en formato **YYYY-MM-DD HH:MM:SS**.
Valores permitidos: '0001-01-01 00:00:00' a '23:59:59 9999-12-31'.
La fecha se debe colocar entre **comillas** simples o dobles, un espacio y la hora que se debe separar por **dos puntos**. Ejemplo: '2021-05-15 11:50:55'.

Datos de tipo boolean

Boolean

MySQL guarda los booleanos como un **cero** o como un **uno**. Por cuestiones de performance, este tipo de dato viene desactivado y no se recomienda su uso.

En el caso de querer guardar valores "verdaderos" y "falsos", se recomienda utilizar el tipo de dato **TINYINT**, donde:

- **0** para representar el **false**.
- **1** para representar el **true**.

C7A – RELACIONES

Relaciones

Las relaciones indican cómo se van a relacionar dos tablas. Siempre que haya una relación entre dos tablas, se debe utilizar la clave primaria para hacer referencia. Esta PK dentro de una tabla que hace referencia a otra tabla se denomina clave foránea, ya que apunta a la clave primaria de otra tabla.

Dentro de una base de datos existen 3 tipos de relaciones:

- **Uno** a **uno**.
- **Uno** a **muchos**.
- **Muchos** a **muchos**.

Cardinalidad

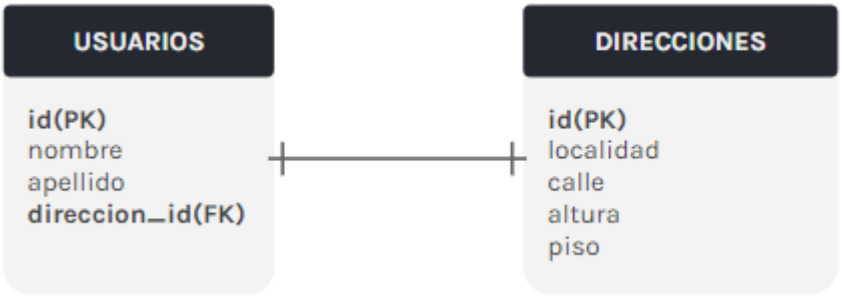
Es la forma en que se relacionan las entidades.

Cardinalidad	Se lee	Representación
1:1	Uno a uno	
1:M	Uno a muchos	
N:M	Muchos a muchos	

Tipos de relaciones

Uno a uno (1:1)

Un usuario **tiene** solo una dirección. Una dirección **pertenece** solo a un **usuario**.
Para establecer la relación colocamos la **clave primaria** de la dirección en la tabla de **usuarios**, indicando que **esa** dirección está asociada a **ese** usuario (Clave foránea).



Uno a muchos (1:N)

Un **cliente** puede tener muchas **tarjetas**. Una **tarjeta** pertenece solo a **un cliente**.
Para establecer la relación colocamos la **clave primaria** del **cliente** en la tabla de **tarjetas**, indicando que **esas** tarjetas están asociadas a un usuario en particular.

Muchos a muchos (N:M)

Un **cliente** puede comprar **muchos productos**. Un **producto** puede ser comprado por **muchos clientes**.
En las relaciones **N:M** la relación en sí pasa a ser una **tabla**. Esta tabla intermedia —también conocida como tabla pivot— puede tener 3 datos: una clave primaria **(PK)** y dos claves foráneas **(FK)**, cada una haciendo referencia a cada tabla de la relación.

MÓDULO 3 - SQL

C8A - INTRODUCCIÓN A DDL Y DML - QUERIES SM

Create, Drop, Alter

Comando CREATE DATABASE

Crea una base de datos desde cero. Se debe aclarar e nombre de la tabla, sus columnas, sus tipos y sus constraints.

```sql
CREATE DATABASE miprimerabasededatos;
USE miprimerabasededatos;
```

```sql
CREATE TABLE nombre_de_la_tabla (
    nombre_de_la_columna_1 TIPO_DE_DATO CONSTRAINT,
    nombre_de_la_columna_2 TIPO_DE_DATO CONSTRAINT
)
```

```sql
CREATE TABLE post (
    id INT PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200)
)
```

Ejemplo CREATE TABLE

```sql
CREATE TABLE peliculas (
    id  INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) NOT NULL,
    awards INT DEFAULT 0,
    release_date DATE NOT NULL,
    length INT NOT NULL
);
```

FOREIGN KEY

Cuando creemos una columna que contenga una id foránea, será necesario usar la sentencia **FOREIGN KEY** para aclarar a qué tabla y a qué columna hace referencia aquel dato.
Es importante remarcar que la tabla "**clientes**" deberá existir antes de correr esta sentencia para crear la tabla "ordenes".

```sql
CREATE TABLE ordenes (
    orden_id INT NOT NULL,
    orden_numero INT NOT NULL,
    cliente_id INT,
    PRIMARY KEY (orden_id),
    FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

Comando DROP TABLE

Borra la tabla que le especifiquemos en la sentencia.

```sql
DROP TABLE IF EXIST peliculas;
```

Comando ALTER TABLE

ALTER TABLE permite alterar una tabla ya existente y va a operar con tres comandos:

- **ADD**: para agregar una columna.
- **MODIFY**: para modificar una columna.
- **DROP**: para borrar una columna.

Ejemplos ALTER TABLE

Agrega la columna **rating**, aclarando tipo de dato y constraint.

```sql
ALTER TABLE peliculas
ADD rating DECIMAL(3,1) NOT NULL;
```

Modifica el decimal de la columna **rating**. Aunque el resto de las configuraciones de la tabla no se modifiquen, es necesario escribirlas en la sentencia.

```sql
ALTER TABLE peliculas
MODIFY rating DECIMAL(4,1) NOT NULL;
```

Borra la columna **rating**.

```sql
ALTER TABLE peliculas
DROP rating;
```

Insert, Update, Delete

INSERT

Existen dos formas de agregar datos en una tabla:

- Insertando datos en **todas** las **columnas**.
- Insertando datos en las **columnas** que **especifiquemos**.

Todas las columnas

Si estamos insertando datos en todas las columnas, no hace falta aclarar los nombres de cada columna. Sin embargo, el orden en el que insertemos los valores, deberá ser el mismo orden que tengan asignadas las columnas en la tabla.

```sql
INSERT INTO table_name (columna_1, columna_2, columna_3, ...)
VALUES (valor_1, valor_2, valor_3, ...);
```

```sql
INSERT INTO artistas (id, nombre, rating)
VALUES (DEFAULT, 'Shakira', 1.0);
```

Columnas específicas

Para insertar datos en una columna en específico, aclaramos la tabla y luego escribimos el nombre de la o las columnas entre los paréntesis.

```SQL
INSERT INTO artistas (nombre)
VALUES ('Calle 13');
```

```SQL
INSERT INTO artistas (nombre, rating)
VALUES ('Maluma', 1.0);
```

UPDATE

UPDATE modificará los registros existentes de una tabla. Al igual que con **DELETE**, es importante no olvidar el **WHERE** cuando escribimos la sentencia, aclarando la condición.

```SQL
UPDATE nombre_tabla
SET columna_1 = valor_1, columna_2 = valor_2, ...
WHERE condición;
```

```SQL
UPDATE artistas
SET nombre = 'Charly Garcia', rating = 1.0
WHERE id = 1;
```

DELETE

Con **DELETE** podemos borrar información de una tabla. Es importante recordar utilizar **siempre** el **WHERE** en la sentencia para agregar la condición de cuáles son las filas que queremos eliminar. Si no escribimos el **WHERE**, estaríamos borrando **toda** la **tabla** y no un registro en particular. Si da error, entrar a preferencias, SQL editor y destildar el "Safe Updates".

```SQL
DELETE FROM nombre_tabla WHERE condición;
```

```SQL
DELETE FROM artistas WHERE id = 4;
```

Select

Cómo usarlo

Toda consulta a la base de datos va a empezar con la palabra **SELECT**.
Su funcionalidad es la de realizar consultas sobre **una** o **varias columnas** de una tabla.
Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

```SQL
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

Ejemplo - Tabla Películas

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

De esta tabla completa, para conocer solamente los títulos y ratings de las películas guardadas en la tabla **películas,** podríamos hacerlo ejecutando la siguiente consulta:

```sql
SELECT id, titulo, rating
FROM peliculas;
```

Where y Order By

WHERE

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT**.

```sql
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;
```

Teniendo una tabla **clientes**, podría consultar primer nombre y apellido, filtrando con un **WHERE** solamente los usuarios **que su país es igual a Argentina** de la siguiente manera:

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE pais = 'Argentina';
```

Operadores

- `=` → Igual a
- `>` → Mayor que
- `>=` → Mayor o igual que
- `<` → Menor que
- `<=` → Menor o igual que
- `<>` → Diferente a
- `!=` → Diferente a

- `IS NULL` → Es nulo
- `BETWEEN` → Entre dos valores
- `IN` → Lista de valores
- `LIKE` → Se ajusta a...

Queries de ejemplo

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE pais <> 'Argentina';
```

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE id < 15;
```

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE id > 5;
```

ORDER BY

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente *(ASC)* según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

```sql
SELECT nombre_columna1, nombre_columna2
FROM tabla
WHERE condicion
ORDER BY nombre_columna1;
```

Query de ejemplo

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un **WHERE** solamente los usuarios **con rating mayor a 1** y ordenarlos de forma descendente tomando como referencia la columna nombre.

```sql
SELECT nombre, rating
FROM artistas
WHERE rating > 1.0
ORDER BY nombre DESC;
```

C11A - USO DE DML - QUERIES ML

Between y Like

BETWEEN

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- BETWEEN **incluye** los **extremos**.
- BETWEEN funciona con **números**, **textos** y **fechas**.
- Se usa como un filtro de un WHERE.

Query de ejemplo

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** solo cuando las edades estén **entre** 6 y 12.

```sql
SELECT nombre, edad
FROM alumnos
WHERE edad BETWEEN 6 AND 12;
```

LIKE

Cuando hacemos un filtro con un **WHERE**, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** *(wildcards)*.

Por ejemplo, podríamos querer buscar:

- Los nombres que tengan la letra 'a' como segundo carácter.
- Las direcciones postales que incluyan la calle 'Monroe'.
- Los clientes que empiecen con 'Los' y terminen con 's'.

COMODÍN %

Es un sustituto que representa **cero**, **uno**, o **varios** caracteres.

COMODÍN _

Es un sustituto para **un solo** carácter.

Queries de ejemplo

Devuelve aquellos nombres que tengan la letra 'a' como segundo carácter.

```sql
SELECT nombre
FROM usuarios
WHERE nombre LIKE '_a%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

```sql
SELECT nombre
FROM usuarios
WHERE direccion LIKE '%Monroe%';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

```sql
SELECT nombre
FROM clientes
WHERE nombre LIKE 'Los%s';
```

Filtros

- Select
- Where
- Order by
- Like
- Between
- AND/OR

Limit y Offset

Limit

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

```sql
SELECT nombre_columna1, nombre_columna2
FROM nombre_tabla
LIMIT cantidad_de_registros;
```

Query de ejemplo

Teniendo una tabla **peliculas**, podríamos armar un top 10 con las películas que tengan más de 4 premios usando un **LIMIT** en la siguiente consulta:

```sql
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

Offset

- En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20.
- *¿Pero cómo haríamos si quisiéramos recuperar sólo 20 películas pero salteando las primeras 10 de la tabla?*
- **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

{código}

Alias

Los **alias** se usan para darle un nombre temporal y más amigable a las **tablas**, **columnas** y **funciones**. Los **alias** se definen durante una consulta y persisten **solo** durante esa consulta. Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias.

```sql
SELECT nombre_columna1 AS alias_nombre_columna1
FROM nombre_tabla;
```

Alias para una columna

```sql
SELECT razon_social_cliente AS nombre
```

Seleccionamos la **columna** *razon_social_cliente* y le asignamos el **alias** nombre.

```sql
FROM cliente
WHERE nombre LIKE 'a%';
```

En el **FROM** elegimos tabla cliente.
Con el **WHERE** filtramos los registros de la columna nombre que empiecen con la letra a.

Alias para una tabla

```sql
SELECT nombre, apellido, edad
```

Seleccionamos las columnas nombre, apellido y edad.

```sql
FROM alumnos_comision_inicial AS alumnos;
```

Hacemos la consulta sobre la tabla *alumnos_comision_inicial* y le **asignamos** el **alias** alumnos.

No es recomendable asignar más de una palabra dentro de un alias. En el caso de necesitarlo, utilizar "_".

De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.

C13A - INFORMES (CHECKPOINT 2)

Funciones de agregación

Las funciones de agregación **realizan cálculos** sobre un conjunto de datos y **devuelven** un **único resultado**. Excepto **COUNT**, las funciones de agregación **ignorarán** los valores **NULL**..

- COUNT
- AVG
- SUM
- MIN
- MAX

COUNT

Devuelve un **único** resultado indicando la cantidad de **filas/registros** que cumplen con el criterio.
Devuelve la cantidad de registros de la tabla movies:

```sql
SELECT COUNT(*) FROM movies;
```

Devuelve la cantidad de películas de la tabla movies con el genero_id 3 y lo muestra en una columna denominada total:

```sql
SELECT COUNT(id) AS total FROM movies WHERE genre_id=3;
```

AVG, SUM

AVG (*average*) devuelve un **único** resultado indicando el **promedio** de una columna cuyo tipo de datos debe ser numérico.
Devuelve el promedio del rating de las películas de la tabla movies:

```sql
SELECT AVG(rating) FROM movies;
```

SUM (suma) devuelve un **único** resultado indicando la **suma** de una columna cuyo tipo de datos debe ser numérico.
Devuelve la suma de las duraciones de las películas de la tabla movies:

```sql
SELECT SUM(length) FROM movies;
```

MIN, MAX

MIN devuelve un **único** resultado indicando el valor **mínimo** de una columna cuyo tipo de datos debe ser numérico.
Devolverá el rating de la película menos ranqueada:

```sql
SELECT MIN(rating) FROM movies;
```

MAX devuelve un **único** resultado indicando el valor **máximo** de una columna cuyo tipo de datos debe ser numérico.
Devolverá el rating de la película mejor ranqueada:

```sql
SELECT MAX(length) FROM movies;
```

GROUP BY

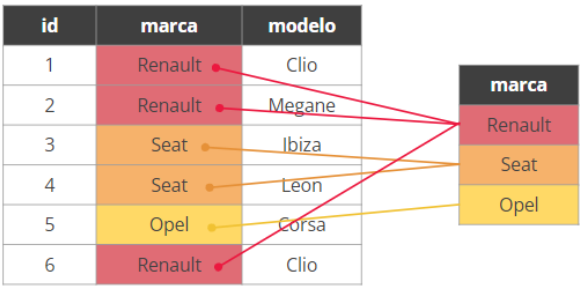
La directriz **GROUP BY** nos va a permitir agrupar los registros de la tabla resultante de una consulta por una o más columnas, según nos sea necesario.

```sql
SELECT columna_1
FROM nombre_tabla
WHERE condition
GROUP BY columna_1;
```

Ejemplo

En el siguiente ejemplo, se utiliza **GROUP BY** para agrupar los **coches** por **marca** mostrando aquellos que tienen el año de fabricación igual o superior al año **2010**.

```sql
SELECT marca
FROM coche
WHERE anio_fabricacion>= 2010
GROUP BY marca;
```



Agrupación de datos

Dado que **GROUP BY** agrupa la información, perdemos el detalle de cada una de las filas. Es decir, ya no nos interesa el valor de cada fila, sino un resultado consolidado entre todas las filas. Veamos la siguiente consulta:

```sql
SELECT id, marca
FROM coche
GROUP BY marca;
```

Si agrupamos los coches por **marca**, ya no podremos visualizar el ID de cada fila. Posiblemente, en la fila nos muestre —para el campo **ID**— el primero de cada grupo de registros.
Veamos algunos ejemplos más...

Ejemplos

Devuelve la marca y el precio más alto de cada grupo de marcas:

```sql
SELECT marca, MAX(precio) AS precio_maximo
FROM coche
GROUP BY marca;
```

Devuelve el género y la duración promedio de cada grupo de géneros:

```sql
SELECT genero, AVG(duracion) AS duracion_promedio
FROM pelicula
GROUP BY genero;
```

Conclusión

En resumen, la cláusula **GROUP BY**:

- Se usa para **agrupar filas** que contienen los **mismos valores**.
- Opcionalmente, se utiliza junto con las **funciones de agregación** (SUM, AVG, COUNT, MIN, MAX) con el objetivo de producir reportes resumidos.

Las consultas que contienen la cláusula GROUP BY se denominan **consultas agrupadas** y solo devuelven una **sola fila** para cada elemento agrupado.

```sql
SELECT marca, MAX(precio)
FROM coche
GROUP BY marca;
```

Having

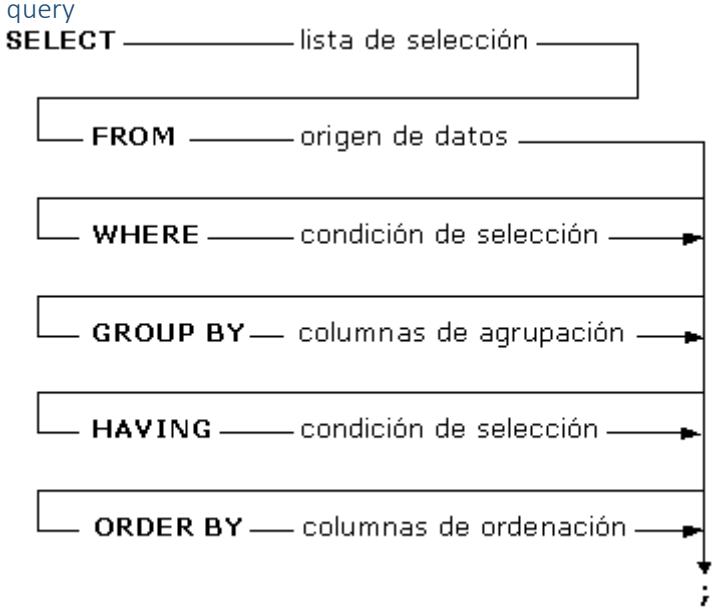
Cumple la misma función que **WHERE**, a diferencia de que **HAVING** permite la implementación de **alias** y **funciones de agregación** en las condiciones de la selección de **datos**.

```sql
SELECT columna_1
FROM nombre_tabla
WHERE condition
GROUP BY columna_1
HAVING condition_Group
ORDER BY columna_1;
```

Esta consulta devolverá la cantidad de clientes por país (agrupados por país). Solamente se incluirán en el resultado aquellos países que tengan al menos 3 clientes.

```sql
SELECT pais, COUNT(clienteId)
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId)>=3;
```

Estructura de una query



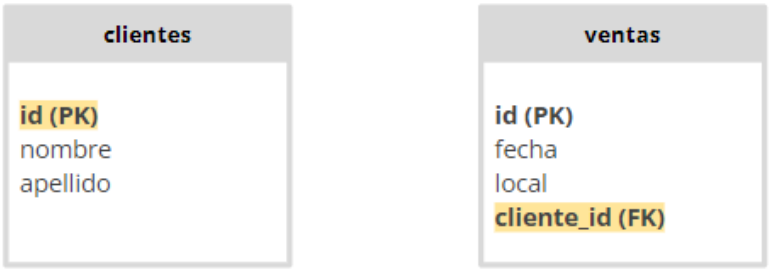
C14A - DML - QUERIES AGREGADAS

Table Reference

Consultas a más de una tabla

Es posible y necesario hacer consultas a distintas tablas y unir los resultados. Por ejemplo, un posible escenario sería querer consultar una tabla en donde están los **datos** de los **clientes** y otra tabla en donde están los **datos** de las **ventas a esos clientes.**

Seguramente, en la tabla de **ventas**, existirá un campo con el ID del cliente **(cliente_id)**.
Si quisiéramos mostrar **todas** las ventas de un cliente concreto, necesitaremos usar datos de **ambas tablas** y **vincularlas** con algún **campo** que **compartan**. En este caso, el **cliente_id.**



Consulta SQL

```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
```

Seleccionamos:

- La columna **id** de la tabla **clientes** y le asignamos el alias **ID.**
- La columna **nombre** de la tabla **clientes.**
- La columna **fecha** de la tabla **ventas.**

El **select** lo hacemos sobre las tablas **clientes** y **ventas.**

Hasta acá la consulta traería **todos** los **clientes y todas** las **ventas.** Por eso nos falta todavía agregar un **filtro** que muestre **solo** las ventas de **cada usuario** en particular.

En el WHERE creamos una condición para traer aquellos registros en donde el ID del cliente sea igual en ambas tablas.

Join

¿Por qué usar JOIN?

JOIN nos permite hacer consultas a distintas tablas y unir los resultados.

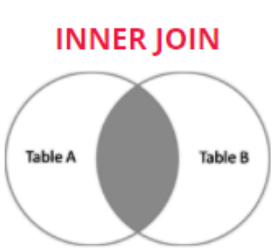
Ventajas

- Su sintaxis es mucho más comprensible.
- Presentan una mejor performance.
- Proveen de ciertas flexibilidades.

INNER JOIN

El **INNER JOIN** es la opción predeterminada y nos devuelve **todos los registros** donde se **cruzan dos o más tablas**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García



factura		
id	cliente_id	fecha
1	2	12/03/2019
2	2	22/08/2019
3	1	04/09/2019

Consulta a múltiples tablas

Antes con **table reference** escribíamos:

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente, factura;
```

Ahora con **INNER JOIN** escribimos:

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente
INNER JOIN factura;
```

Si bien ya dimos el primer paso, **cruzar** ambas tablas, aún nos falta aclarar **dónde** está ese cruce. Es decir, qué **clave primaria (PK)** se cruzará con qué **clave foránea (FK)**.

Definiendo el INNER JOIN

Para definir el **INNER JOIN** tenemos que indicar el filtro por el cual se **evaluará** el **cruce**. Para esto, debemos utilizar la palabra reservada **ON**. Es decir, que lo que antes escribíamos en el **WHERE** de table reference, ahora lo escribiremos en el **ON** de INNER JOIN.

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente
INNER JOIN factura
ON cliente.id = factura.cliente_id;
```

Distinct

Al realizar una consulta en una tabla, puede ocurrir que en los resultados existan dos o más **filas idénticas**. En algunas situaciones, nos pueden solicitar un listado con registros **no duplicados**, para esto, utilizamos la cláusula **DISTINCT** que devuelve un listado en donde cada fila es distinta.

```sql
SELECT DISTINCT columna_1, columna_2
FROM nombre_tabla;
```

Ejemplo

Partiendo de una tabla de **usuarios**, si ejecutamos la consulta:

```sql
SELECT pais FROM usuarios;
```

Obtendremos cinco filas:

usuarios
Perú
Perú
Argentina
Colombia
Argentina

Si agregamos la cláusula **DISTINCT** en la consulta:

```sql
SELECT DISTINCT pais FROM usuarios;
```

Obtendremos tres filas:

usuarios
Perú
Argentina
Colombia

{código}

```sql
SELECT DISTINCT actor.nombre, actor.apellido
FROM actor
INNER JOIN actor_pelicula
ON actor_pelicula.actor_id = actor.id
INNER JOIN pelicula
ON pelicula.id = actor_pelicula.pelicula_id
WHERE pelicula.titulo LIKE '%Harry Potter%';
```

En este ejemplo vemos una query que **pide** los **actores** que hayan actuado en **cualquier película** de **Harry Potter**.

Si no escribiéramos el **DISTINCT**, los actores que hayan participado en más de una película, aparecerán repetidos en el resultado.

Funciones de alteración

Las funciones de alteración más comunes son:

- CONCAT
- COALESCE
- DATEDIFF
- TIMEDIFF
- EXTRACT
- REPLACE
- DATE_FORMAT
- DATE_ADD
- DATE_SUB
- CASE

Más funciones de alteración: https://dev.mysql.com/doc/refman/8.0/en/

CONCAT

Usamos **CONCAT** para **concatenar** dos o más expresiones:

```
SQL  SELECT CONCAT('Hola', ' a ', 'todos.');
     > 'Hola a todos.'

SQL  SELECT CONCAT('La respuesta es: ', 24, '.');
     > 'La respuesta es 24.'

SQL  SELECT CONCAT('Nombre: ', apellido, ', ', nombre, '.')
     FROM actor;
     > 'Nombre: Clarke, Emilia.'
```

COALESCE

Usamos **COALESCE** para sustituir el valor **NULL** en una sucesión de expresiones o campos. Es decir, si la primera expresión es Null, se sustituye con el valor de una segunda expresión, pero si este valor también es Null, se puede sustituir con el valor de una tercera expresión y así sucesivamente.

```
SQL  SELECT COALESCE(NULL, 'Sin datos');
     > 'Sin datos'

SQL  SELECT COALESCE(NULL, NULL, 'Digital House');
     > 'Digital House'
```

Los tres clientes de la siguiente tabla poseen uno o más datos nulos:

```
SQL  SELECT id, apellido, nombre, telefono_movil, telefono_fijo
     FROM cliente;
```

cliente

id	apellido	nombre	telefono_movil	telefono_fijo
1	Pérez	Juan	1156685441	43552215
2	Medina	Rocío	Null	43411722
3	López	Matías	Null	Null

Usando **COALESCE** podremos sustituir los **datos nulos** en cada registro, indicando la columna a evaluar y el valor de sustitución.

```sql
SELECT id, apellido, nombre, COALESCE(telefono_movil, telefono_fijo, 'Sin datos')
AS telefono FROM cliente;
```

cliente			
id	apellido	nombre	telefono
1	Pérez	Juan	1156685441
2	Medina	Rocío	**43411722**
3	López	Matías	**Sin datos**

DATEDIFF

Usamos **DATEDIFF** para devolver la **diferencia** entre dos fechas, tomando como granularidad el intervalo especificado.

```sql
SELECT DATEDIFF('2021-02-03 12:45:00', '2021-01-01 07:00:00');
> 33
```

Devuelve 33 porque es la cantidad de días de la diferencia entre las fechas indicadas.

```sql
SELECT DATEDIFF('2021-01-15', '2021-01-05');
> 10
```

Devuelve 10 porque es la cantidad de días de la diferencia entre las fechas indicadas.

TIMEDIFF

Usamos **TIMEDIFF** para devolver la **diferencia** entre dos horarios, tomando como granularidad el intervalo especificado.

```sql
SELECT TIMEDIFF('2021-01-01 12:45:00', '2021-01-01 07:00:00');
> 05:45:00
```

```sql
SELECT TIMEDIFF('18:45:00', '12:30:00');
> 06:15:00
```

EXTRACT

Usamos EXTRACT para extraer partes de una fecha:

```sql
SELECT EXTRACT(SECOND FROM '2014-02-13 08:44:21');
> 21
```

```sql
SELECT EXTRACT(WEEK FROM '2014-02-13 08:44:21');
> 6
```

```sql
SELECT EXTRACT(MINUTE FROM '2014-02-13 08:44:21');
> 44
```

```sql
SELECT EXTRACT(MONTH FROM '2014-02-13 08:44:21');
> 2
```

```sql
SELECT EXTRACT(HOUR FROM '2014-02-13 08:44:21');
> 8
```

```sql
SELECT EXTRACT(QUARTER FROM '2014-02-13 08:44:21');
> 1
```

```sql
SELECT EXTRACT(DAY FROM '2014-02-13 08:44:21');
> 13
```

```sql
SELECT EXTRACT(YEAR FROM '2014-02-13 08:44:21');
> 2014
```

REPLACE

Usamos **REPLACE** para reemplazar una cadena de caracteres por otro valor. Cabe aclarar que esta función hace distinción entre minúsculas y mayúsculas.

```
SQL  SELECT REPLACE('Buenas tardes', 'tardes', 'Noches');
> Buenas Noches
```

```
SQL  SELECT REPLACE('Buenas tardes', 'a', 'A');
> BuenAs tArdes
```

```
SQL  SELECT REPLACE('1520', '2', '5');
> 1550
```

DATE_FORMAT

Usamos **DATE_FORMAT** para cambiar el formato de salida de una fecha según una condición dada.

```
SQL  SELECT DATE_FORMAT('2017-06-15', '%Y');
> 2017
```

```
SQL  SELECT DATE_FORMAT('2017-06-15', '%W %M %e %Y');
> Thursday June 15 2017
```

Para mostrar la fecha escrita en español se debe configurar el idioma con la siguiente instrucción:

```
SQL  SET lc_time_names = 'es_ES';
     SELECT DATE_FORMAT('2017-06-15', '%W, %e de %M de %Y');
> jueves, 15 de junio de 2017
```

DATE_ADD

Usamos **DATE_ADD** para sumar o agregar un período de tiempo a un valor de tipo DATE o DATETIME.

```
SQL  SELECT DATE_ADD('2021-06-30', INTERVAL '3' DAY);
> 2021-07-03
```

```
SQL  SELECT DATE_ADD('2021-06-30', INTERVAL '9' MONTH);
> 2022-03-30
```

```
SQL  SELECT DATE_ADD('2021-06-30 09:30:00', INTERVAL '4' HOUR);
> 2021-06-30 13:30:00
```

DATE_SUB

Usamos **DATE_SUB** para restar o quitar un período de tiempo a un valor de tipo DATE o DATETIME.

```
SQL  SELECT DATE_SUB('2021-06-30', INTERVAL '3' DAY);
> 2021-06-27
```

```
SQL  SELECT DATE_SUB('2021-06-30', INTERVAL '9' MONTH);
> 2020-09-30
```

```
SQL  SELECT DATE_SUB('2021-06-30 09:30:00', INTERVAL '4' HOUR);
> 2021-06-30 05:30:00
```

CASE

Usamos **CASE** para **evaluar condiciones** y devolver la primera condición que se cumpla. En este ejemplo, la tabla resultante tendrá 4 columnas: id, titulo, rating, calificacion. Esta última columna mostrará los valores: Mala, Regular, Buena y Excelente; **según** el **rating** de la película.

```sql
SELECT id, titulo, rating,
  CASE
    WHEN rating < 4 THEN 'Mala'
    WHEN rating BETWEEN 4 AND 6 THEN 'Regular'
    WHEN rating BETWEEN 7 AND 9 THEN 'Buena'
    ELSE 'Excelente'
  END AS calificacion
FROM pelicula;
```

Tabla resultante:

pelicula			
id	**titulo**	**rating**	**calificacion**
1	El Padrino	6	**Regular**
2	Tiburón	4	**Regular**
3	Jurassic Park	9	**Buena**
4	Titanic	10	**Excelente**
5	Matrix	3	**Mala**

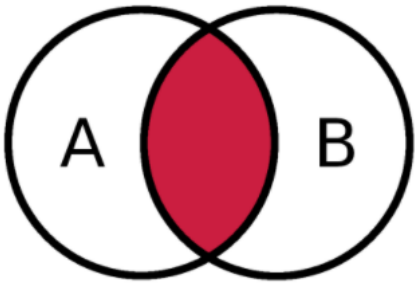
C16A - DML - QUERIES XXL

Tipos de Joins: Inner, Left y Right

INNER JOIN

El **INNER JOIN** entre dos tablas devuelve únicamente los registros que cumplen la condición indicada en la cláusula **ON**. Es la opción predeterminada y nos devuelve **todos los registros** donde **se cruzan dos o más tablas**.

```sql
SELECT columna1, columna2, …
FROM tabla A
INNER JOIN tabla B
ON condicion
```

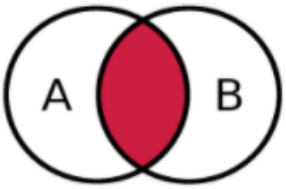


Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente

id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

INNER JOIN



factura

id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

El ejemplo anterior, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
INNER JOIN factura
ON cliente.id = factura.cliente_id;
```

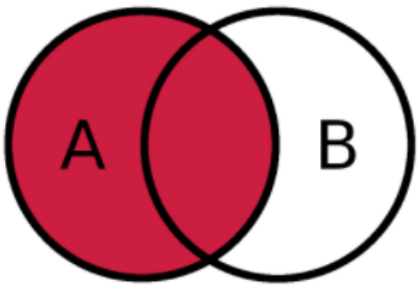
Dando como resultado:

nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019

LEFT JOIN

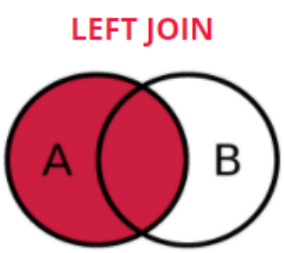
El **LEFT JOIN** entre dos tablas devuelve todos los registros de la primera tabla (en este caso sería la tabla A), **incluso** cuando los **registros no cumplan** la condición indicada en la cláusula **ON**.

```sql
SELECT columna1, columna2, …
FROM tabla A
LEFT JOIN tabla B
ON condicion
```



Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellos clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

El ejemplo anterior, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
LEFT JOIN factura
ON cliente.id = factura.cliente_id;
```

Dando como resultado:

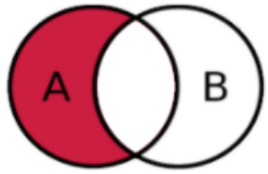
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019
null	García	Marta	null

LEFT Excluding JOIN

Este tipo de **LEFT JOIN** nos devuelve únicamente los **registros** de una primera tabla (A), excluyendo los registros que cumplan con la condición indicada en la cláusula **ON**.

Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

Continuando con el ejemplo, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
LEFT JOIN factura
ON cliente.id = factura.cliente_id
WHERE factura.id IS NULL;
```

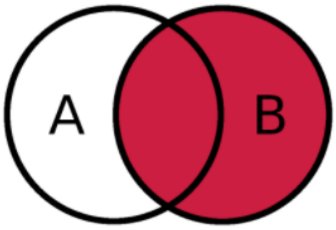
Dando como resultado:

nro_factura	apellido	nombre	fecha
null	García	Marta	null

RIGHT JOIN

El **RIGHT JOIN** entre dos tablas devuelve todos los registros de la segunda tabla (B), **incluso** cuando los **registros no cumplan** la condición indicada en la cláusula **ON**.

```sql
SELECT columna1, columna2, …
FROM tabla A
RIGHT JOIN tabla B
ON condicion
```

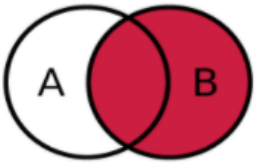


Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellas facturas que no tengan un cliente asignado.

cliente

id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT JOIN



factura

id	cliente_id	fecha
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

El ejemplo anterior, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
RIGHT JOIN factura
ON cliente.id = factura.cliente_id;
```

Dando como resultado:

nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
12	**null**	**null**	20/09/2019
13	Perez	Juan	24/09/2019

RIGHT Excluding JOIN

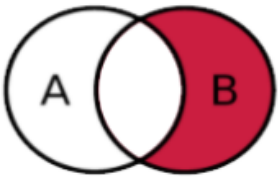
Este tipo de **RIGTH JOIN** nos devuelve únicamente los registros de una segunda tabla (B), **excluyendo** los registros que cumplan con la condición indicada en la cláusula **ON**.

Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de facturas que no tengan asignado un cliente.

cliente

id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT Excluding JOIN



factura

id	cliente_id	fecha
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

Continuando con el ejemplo, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
RIGHT JOIN factura
ON cliente.id = factura.cliente_id
WHERE cliente.id IS NULL;
```

Dando como resultado:

nro_factura	apellido	nombre	fecha
12	**null**	**null**	20/09/2019

Experimentando con LEFT y RIGHT

¿Qué pasa si intercambiamos de lugar las tablas o si cambiamos LEFT por RIGHT?

LEFT JOIN -> RIGHT JOIN

Experimentemos con el último ejemplo que vimos.

```sql
SELECT factura.id AS factura, apellido
FROM cliente
LEFT JOIN factura
ON factura.cliente_id = cliente.id;
```

```sql
SELECT factura.id AS factura, apellido
FROM cliente
RIGHT JOIN factura
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
13	Perez
null	García

factura	apellido
11	Sanchez
12	**null**
13	Perez

Intercambiando tablas

Ahora, intercambiamos el lugar de las tablas implicadas.

```sql
SELECT factura.id AS factura, apellido
FROM factura
LEFT JOIN cliente
ON factura.cliente_id = cliente.id;
```

```sql
SELECT factura.id AS factura, apellido
FROM factura
RIGHT JOIN cliente
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
12	**null**
13	Perez

factura	apellido
11	Sanchez
13	Perez
null	García

C17A - DML - QUERIES XXL - PARTE II

Vistas

¿Qué es una vista?

Es un elemento de la base de datos que facilita el acceso a los datos de las tablas. Básicamente, su función es **guardar** un **SELECT**. Este es un recurso que nos permite visualizar los resultados abstrayéndonos de cómo esté definida una consulta.

Las vistas tienen la misma **estructura** que una **tabla** (filas y columnas). La diferencia es que solo se almacena la **definición de la consulta**, no así los datos. En otras palabras, una vista es una tabla virtual basada en el conjunto de resultados de una **consulta SQL**.

¿Para qué sirven las vistas?

- Para simplificar el acceso a los datos cuando se requiere la implementación de consultas complejas.
- Para impedir la modificación de datos por terceros.
- Para facilitar la consulta de datos a aquellas personas que no conozcan el modelo de datos o que no sean expertos en SQL.

¿Qué hay que saber sobre las vistas?

- Una vista se **ejecuta** en el momento en que se invoca.
- Los **nombres** de las vistas deben ser **únicos** (no se pueden usar nombres de tablas existentes).
- Solamente se pueden incluir **sentencias SQL** de tipo **SELECT**.
- Los **campos** de las vistas heredan los **tipos de datos** de la tabla.
- El conjunto de resultados que devuelve una vista es **inmodificable**, a diferencia de lo que sucede con el conjunto de resultados de una tabla.

Creación de vistas

Una vista se crea con la cláusula **CREATE VIEW**:

```SQL
CREATE VIEW nombre_de_la_vista AS consulta SQL;
```

```SQL
CREATE VIEW canciones_de_rock AS
SELECT canciones.id, canciones.nombre, generos.nombre AS genero
FROM canciones
INNER JOIN generos
ON canciones.id_genero = generos.id
WHERE generos.nombre IN ('Rock', 'Rock And Roll');
```

Modificación de vistas

Usamos la cláusula **ALTER VIEW** para modificar o reemplazar una vista.

```SQL
ALTER VIEW nombre_de_la_vista AS consulta SQL;
```

```SQL
ALTER VIEW vista_coche AS SELECT * FROM coche WHERE marca = 'Fiat';
```

Eliminación de vistas

Utilizamos la cláusula **DROP VIEW** para eliminar una vista.

```sql
DROP VIEW nombre_de_la_vista;
```

```sql
DROP VIEW vista_coche;
```

Invocación de vistas

Si bien las vistas son elementos diferentes a las tablas, la **invocación** es la misma que la de una tabla. Es decir, se utiliza la cláusula **SELECT**.

Invocando una tabla:

```sql
SELECT * FROM coche;
```

Invocando una vista:

```sql
SELECT * FROM vista_coche;
```

Invocando una vista junto con la cláusula **WHERE**:

```sql
SELECT * FROM vista_coche WHERE id > 10;
```

MÓDULO 4 - BUENAS PRÁCTICAS Y OPTIMIZACIÓN

C20A - BUENAS PRÁCTICAS

Buenas prácticas en SQL

Las siguientes recomendaciones tienen la finalidad de **optimizar los tiempos de ejecución de las consultas y escribir consultas SQL sólidas**.
Cuando no implementamos estas recomendaciones, SQL realiza el análisis sobre toda la/s tabla/s que se están consultando. Si la tabla tiene numerosos registros, el tiempo en devolver el resultado puede perjudicar a la aplicación.

CREATE

- *Intentemos utilizar VARCHAR en vez de TEXT.*

Si requiere almacenar volúmenes de texto muy grandes, pero son menores a 8000 caracteres.

- *Evalúemos cuidadosamente el uso de CHAR y VARCHAR.*

El uso de CHAR y VARCHAR depende de si el campo en el que se va a usar varía mucho o no de tamaño. El motor de SQL procesa más rápido las columnas de longitud fija. Usemos CHAR para columnas de poca variación en longitud y VARCHAR para aquellas que no tienen una longitud estable o promedio.

- *No usemos columnas con tipos de datos FLOAT, REAL o DATETIME como FOREIGN KEY.*

- *Usemos CONSTRAINT para mantener la integridad de los datos.*

- *Evitemos claves primarias COMPUESTAS.*

Si esperamos que nuestra tabla con una clave primaria compuesta tenga millones de filas, el rendimiento de la operación CRUD está muy degradado.
En ese caso, es mucho mejor usar una clave primaria ID simple que tenga un índice lo suficientemente compacto y establezca las restricciones de motor de bases de datos necesarias para mantener la singularidad.

SELECT

- *Evitemos usar SELECT * FROM tabla.*

El comodín (*) debe omitirse y en su lugar especificarse los campos que sean necesario traerse. El uso del comodín impide, además, un uso efectivo de forma eficiente de los índices.

- *Anteponer el ALIAS de la tabla a cada columna.*

Especificar el alias de la tabla delante de cada campo definido en el SELECT ahorra tiempo al motor de tener que buscar a qué tabla pertenece el campo especificado.

- *Evitemos en la medida de lo posible el uso de GROUP BY, DISTINCT y ORDER BY.*

Evadamos, siempre que sea posible, el uso de GROUP BY, DISTINCT y ORDER BY, dado que consume una elevada cantidad de recursos.
Consideremos si es realmente necesario usarlo o si, por otro lado, se puede dejar el ordenamiento de los resultados a la aplicación que recibirá los datos.

WHERE

- *Evitemos usar de Wilcards en LIKE como "%valor%"*

En el caso de que se use la instrucción LIKE, no usemos el comodín "%" al inicio de la cadena a buscar. Esto debido a que si se aplica, la búsqueda tendría que leer todos los datos de la tabla o tablas involucradas para responder a la consulta. Se recomienda que existan al menos tres caracteres antes del comodín.

- *Evitar usar IN en subconsultas, es mejor EXISTS*

Promover el uso de EXISTS y NOT EXISTS, en lugar de IN y NOT IN.

- *Intente no utilizar funciones dentro de las condiciones del WHERE.*

SQL no puede buscar eficientemente los registros cuando utiliza funciones, por ejemplo, de conversión, dentro de una columna. En las condiciones intente utilizar el formato de la columna original.

UNION

- *Utilice UNION ALL para evitar un distinct implícito*

En caso de usar la instrucción UNION y existiera la seguridad de que en los SELECT involucrados no se obtendrán registros duplicados, entonces, lo recomendable en este escenario es sustituir UNION por UNION ALL para evitar que se haga uso implícito de la instrucción DISTINCT ya que esta aumenta el consumo de recursos.

CRUD

- *Usar SET NOCOUNT ON con operaciones CRUD.*

Usar SET NOCOUNT ON con operaciones CRUD para no contar el número de filas afectadas y ganar rendimiento sobre todo en tablas con muchos registros.

Orden de procesamiento de una query

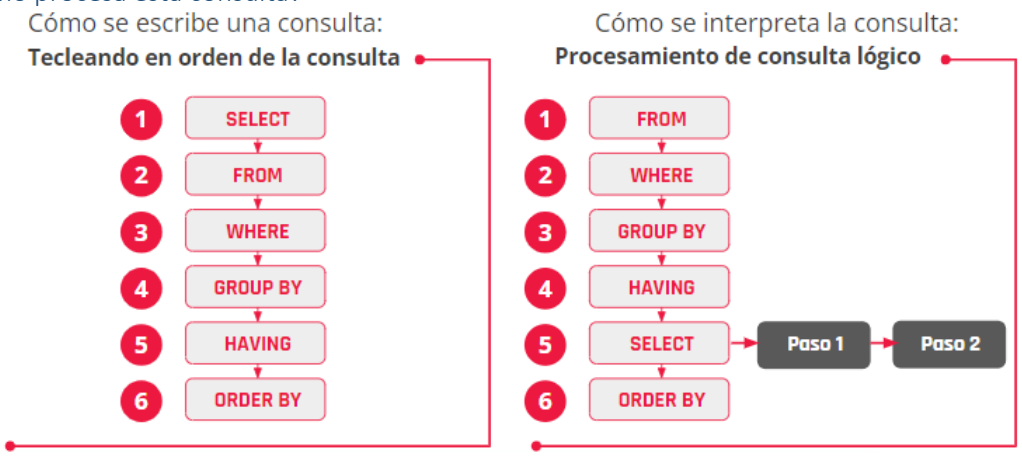
¿Cómo lee la consulta SQL el servidor de bases de datos?

El servidor no lee la consulta exactamente igual al orden de cómo lo escribimos. Una vez que hacemos clic en **EJECUTAR**, el servidor recibe la consulta y la ordena de otra forma.

¿Para qué?

Para interpretar cada una de las partes de nuestro código y responder con el menor tiempo de respuesta.

¿Cómo procesa esta consulta?



¿Cómo interpreta la consulta un motor de base de datos?

En el siguiente orden:

1. FROM
2. JOIN
3. ON
4. WHERE
5. GROUP BY
6. HAVING
7. SELECT
8. DISTINCT
9. ORDER BY
10. LIMIT

1, 2 y 3 FROM, ON y JOIN

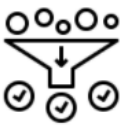
Se obtiene **la tabla o las tablas** de donde necesitamos la información. Es por ello que es el primer bloque en ejecutarse. Tenemos que tener bien en claro de dónde tenemos que sacar los datos y sus condiciones de junta.

4 WHERE

Es aquí donde aplicamos los **filtros** necesarios, uno de los errores es cuando intentamos utilizar el alias de una columna que está en el SELECT. Genera el error debido a que esa columna aún no es creada, se genera en la fase del SELECT y todavía no se ejecuta.

5 GROUP BY

Una vez filtradas las filas que necesitamos, podemos utilizar la sentencia GROUP BY para agrupar resultados, recordemos que aquellas columnas que no utilicen una sentencia de agregado como MAX, MIN, AVG, COUNT, entre otras, deberán ser listadas en esta fase. De la misma forma que en la fase anterior no debemos utilizar los alias.



6 HAVING

Es similar a la fase WHERE, pero aquí se filtran los resultados del GROUP BY, es por ello que se ejecuta después.

7 SELECT

Es en esta fase es donde las columnas son creadas a partir de las funciones que hemos indicado, por ejemplo, YEAR, COUNT, AVG, entre otras. También podemos utilizar sentencias como CASE WHEN, IIF.

8 DISTINCT

Una vez que está calculado el resultado, si necesitamos quitar los duplicados, se realiza en esta etapa. Es opcional.

9 ORDER BY

Es la última fase y es en la única en la que se permiten utilizar los ALIAS que fueron declarados en la fase SELECT dado que esta última se ejecuta previamente.

10 LIMIT

¿Necesitamos mostrar solo una X cantidad de filas? Bueno, se realiza en el último paso. Es una opción de presentación de la información, no de cálculo.

¿Qué es un índice?

Es una **estructura de datos** que mejora la velocidad de las consultas, por medio de un **identificador único** de cada fila de una tabla, permitiendo un rápido acceso a los registros de una tabla en una base de datos.

Veamos con un ejemplo sobre una tabla de alumnos:

Id_Alumno (PK)	Nombre	Apellido	Email	Fecha_Nacimiento
1	Jose	Mentos	jmentos@gmail.com	29/5/2002
2	Laura	Gomez	LauG93@gmail.com	02/3/1993
3	Lucas	Estevanez	EsteLu@gmail.com	31/3/2000

Si buscamos un alumno por su ID, sería fácil porque este es un elemento único, ordenado y estructurado.

¿Qué sucede si una aplicación necesita buscar los alumnos por email?

Cada vez que tenga que encontrar un alumno, deberá buscar celda por celda hasta encontrar el email solicitado. Con 3 filas es fácil, pero ¿si tenemos una base de 1 millón de alumnos? ¿Y si la búsqueda se realiza repetidas veces durante el día?
Vamos a tener problemas de performance. Donde cada consulta que realicemos demorará tiempo en responder.

¡Para solucionar esto se utilizan índices!
Un índice es una estructura adicional, donde elegimos 1 o más columnas que formarán parte del índice. Esto permitirá localizar de forma rápida las filas de la tabla en base a su contenido en la/las columna/s indexada/s.

Ventajas

- La utilización de índices puede mejorar el rendimiento de las consultas ya que los datos necesarios para satisfacer las necesidades de la consulta existen en el propio índice.
- Pueden mejorar de forma significativa el rendimiento si la consulta contiene agregaciones (GROUP BY), combinaciones (JOINS) de tabla o una mezcla de agregaciones y combinaciones.

Desventajas

Aunque parece muy bueno el funcionamiento de un índice también tiene sus desventajas.

- Las tablas utilizadas para almacenar los índices ocupan espacio.
- Los índices **consumen recursos** ya que cada vez que se realiza una **operación de actualización, inserción o borrado** en la tabla indexada, se tienen que actualizar todas las tablas de índice definidas sobre ella —en la actualización solo es necesaria la actualización de los índices definidos sobre las columnas que se actualizan—. Por estos motivos no es buena idea definir índices indiscriminadamente.

Consideraciones

- Hay que **evitar crear demasiados índices** en tablas que se actualizan con mucha frecuencia y procurar definirlos con el menor número de columnas posible.
- Es conveniente utilizar un número mayor de índices para mejorar el rendimiento de consultas en tablas con pocas necesidades de actualización, pero con grandes volúmenes de datos.
- Se recomienda utilizar una longitud corta en la clave de los índices agrupados. Los índices agrupados también mejoran si se crean en columnas únicas o que no admiten valores NULL.

Tipos de índices

Tenemos los tipos de índices: simple, compuesto, agrupado y no agrupado.

- **Simple:** está definido sobre una sola columna:

```sql
CREATE INDEX "I_LIBROS_AUTOR"
ON "LIBROS" (AUTOR);
```

- **Compuesto:** está formado por varias columnas de la misma tabla.

```sql
CREATE INDEX "I_LIBROS_AUTOREDITORIAL"
ON "LIBROS" (AUTOR,EDITORIAL);
```

- **Único:** los valores deben ser únicos y diferentes. Si intentamos agregar un registro con un valor ya existente, aparece un mensaje de error. A su vez, puede ser simple o compuesto.

```sql
CREATE UNIQUE INDEX "I_LIBROS_AUTOR"
ON "LIBROS" (AUTOR);
```

```sql
CREATE UNIQUE INDEX "I_LIBROS_AUTOREDITORIAL"
ON "LIBROS" (AUTOR,EDITORIAL);
```

En algunos otros motores SQL

- **Índice agrupado (CLUSTERED):** almacena los datos de las filas en orden. Solo se puede crear un único índice agrupado en una tabla de base de datos. Esto funciona de manera eficiente únicamente si los datos se ordenan en orden creciente o decreciente.

```SQL
CREATE CLUSTERED INDEX "I_LIBROS_AUTOR"
ON "LIBROS" (AUTOR);
```

- **Índice no agrupado:** organiza los datos de forma aleatoria, pero el índice especifica internamente un orden lógico. El orden del índice no es el mismo que el ordenamiento físico de los datos. Los índices no agrupados funcionan bien con tablas donde los datos se modifican con frecuencia y el índice se crea en las columnas utilizadas en orden por las declaraciones WHERE y JOIN.

```SQL
CREATE NONCLUSTERED INDEX "I_LIBROS_AUTOR"
ON "LIBROS" (AUTOR);
```

Sintaxis

Sintaxis CREATE INDEX

Con **CREATE INDEX** podemos crear un índice indicando las columnas involucradas:

```SQL
CREATE INDEX "NOMBRE_ÍNDICE"
ON "NOMBRE_TABLA" (NOMBRE_COLUMNA);
```

```SQL
CREATE INDEX "I_LIBROS_AUTOREDITORIAL"
ON "LIBROS" (AUTOR,EDITORIAL);
```

Sintaxis DROP INDEX

Con **DROP INDEX** podemos eliminar un índice de una determinada tabla:

```SQL
ALTER TABLE "NOMBRE_TABLA"
DROP INDEX "NOMBRE_ÍNDICE";
```

Sintaxis ANALYZE TABLE

Con **ANALYZE TABLE** analizamos y almacenamos la distribución de claves para una tabla:

```SQL
ANALYZE TABLE "NOMBRE_TABLA";
```

Expliquemos la consulta SQL

Nos ponemos en el lugar de un **motor de base de datos**.

```
SELECT Legajo, Nombre_Apellido
FROM Alumnos
ORDER BY Legajo
```

"Lo primero que realizamos es ubicar la tabla Alumnos dentro de la base de datos de la Universidad.
Analizamos las columnas que tiene esa tabla y las que nos solicitan en el Select.
Nos quedamos solamente con aquellas columnas que necesitamos: Legajo y Nombre_Apellido y el resto las eliminamos del resultado provisorio.
Una vez que tengo el listado, lo ordenamos por la columna Legajo de manera ascendente.
Este resultado lo enviamos al cliente (Workbench) para que el usuario visualice el resultado."

C23A - STORED PROCEDURES

Stored Procedures

¿Qué es un stored procedure?

Son un conjunto de instrucciones en formato SQL que se almacenan, compilan y ejecutan dentro del servidor de bases de datos.
Pueden incluir parámetros de entrada y salida, devolver resultados tabulares o escalares, mensajes para el cliente e invocar instrucciones DDL y DML.
Se los utiliza para definir la lógica del negocio dentro de la base de datos y reducir la necesidad de codificar dicha lógica en programas clientes.

Estructura de un stored procedure

1. **DELIMITER:** Se escribe esta cláusula seguida de una combinación de símbolos que no serán utilizados en el interior del SP.
2. **CREATE PROCEDURE:** Se escribe este comando seguido del nombre que identificará al SP.
3. **BEGIN:** Esta cláusula se utiliza para indicar el inicio del código SQL.
4. **Bloque de instrucciones SQL**.
5. **END:** Se escribe esta cláusula seguida de la combinación de símbolos definidos en DELIMITER y se utiliza para indicar el final del código SQL.

```SQL
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento()
BEGIN
    -- Bloque de instrucciones SQL;
END $$
```

Definición de un stored procedure

- **CREATE PROCEDURE**: Crea un procedimiento almacenado.

```SQL
CREATE PROCEDURE sp_nombre_procedimiento()
```

- **DROP PROCEDURE**: Elimina un procedimiento almacenado. Se requiere del privilegio de ALTER ROUTINE.

```SQL
DROP PROCEDURE [IF EXISTS] sp_nombre_procedimiento();
```

Variables

Declaración de variables

- Dentro de un **SP** se permite declarar variables que son elementos que almacenan datos que pueden ir cambiando a lo largo de la ejecución.
- La declaración de variables se coloca después de la cláusula BEGIN y antes del bloque de instrucciones SQL.
- Opcionalmente, se puede definir un valor inicial mediante la cláusula DEFAULT.

Sintaxis:

```SQL
DECLARE nombre_variable TIPO_DE_DATO [DEFAULT valor];
```

Ejemplo:

```SQL
DECLARE salario FLOAT DEFAULT 1000.00;
```

Asignación de valores a variables

- Para asignar un valor a una variable se utiliza la cláusula SET.
- Las variables solo pueden contener valores escalares. Es decir, un solo valor.

Sintaxis:

```sql
SET nombre_variable = expresión;
```

Ejemplo:

```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento()
BEGIN
    DECLARE salario FLOAT DEFAULT 1000.00;
    SET salario = 25700.50;
END $$
```

Parámetros

¿Qué es un parámetro?

- Los parámetros son variables por donde se envían y reciben datos de programas clientes.
- Se definen dentro de la cláusula CREATE.
- Los **SP** pueden tener uno, varios o ningún parámetro de entrada y asimismo, pueden tener uno, varios o ningún parámetro de salida.
- Existen 3 tipos de parámetros:

Parámetro	Tipo	Función
IN	Entrada	Recibe datos
OUT	Salida	Devuelve datos
INOUT	Entrada-Salida	Recibe y devuelve datos

Declaración del parámetro IN

Es un parámetro de entrada de datos y se utiliza para recibir valores. Este parámetro viene definido por defecto cuando no se especifica su tipo.

Sintaxis

```sql
CREATE PROCEDURE sp_nombre_procedimiento(IN param1 TIPO_DE_DATO, IN param2 TIPO_DE_DATO);
```

Ejemplo:

```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(IN id_usuario INT)
BEGIN
    -- Bloque de instrucciones SQL;
END $$
```

Ejecución:

```sql
CALL sp_nombre_procedimiento(11);
```

Declaración del parámetro OUT

Es un parámetro de salida de datos y se utiliza para devolver valores.

Sintaxis

```sql
CREATE PROCEDURE sp_nombre_procedimiento(OUT param1 TIPO_DE_DATO, OUT param2 TIPO_DE_DATO);
```

Ejemplo:

```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(OUT salario FLOAT)
BEGIN
    SET salario = 25700.50;
END $$
```

Ejecución:

```sql
CALL sp_nombre_procedimiento(@salario);
SELECT @salario; -- Bloque de instrucciones SQL
```

Declaración del parámetro INOUT

Es un mismo parámetro que utiliza para la entrada y salida de datos. Puede recibir valores y devolverlos los resultados en la misma variable.

Sintaxis

```sql
CREATE PROCEDURE sp_nombre_procedimiento(INOUT param1 TIPO_DE_DATO, INOUT param2 TIPO_DE_DATO);
```

Ejemplo:

```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(INOUT aumento FLOAT)
BEGIN
    SET aumento = aumento + 25700.50;
END $$
```

Ejecución:

```sql
SET @salario = 2000.00; -- Declaración y asignación de variable (dato)
CALL sp_nombre_procedimiento(@salario); -- Ejecución y envío de dato (2000.00)
SELECT @salario; -- Muestra el resultado
```

Ventajas y desventajas

Ventajas

- **Gran velocidad de respuesta**: Todo se procesa dentro del servidor.
- **Mayor seguridad**: Se limita e impide el acceso directo a las tablas donde están almacenados los datos, evitando la manipulación directa por parte de las aplicaciones clientes.
- **Independencia**: Todo el código está dentro de la base de datos y no depende de archivos externos.
- **Reutilización del código**: se elimina la necesidad de escribir nuevamente un conjunto de instrucciones.
- **Mantenimiento más sencillo**: Disminuye el costo de modificación cuando cambian las reglas de negocio.

Desventajas

- **Difícil modificación:** Si se requiere modificarlo, su definición tiene que ser reemplazada totalmente. En bases de datos muy complejas, la modificación puede afectar a las demás piezas de software que directa o indirectamente se refieran a este.
- **Aumentan el uso de la memoria:** Si usamos muchos procedimientos almacenados, el uso de la memoria de cada conexión que utiliza esos procedimientos se incrementará sustancialmente.
- **Restringidos para una lógica de negocios compleja:** En realidad, las construcciones de procedimientos almacenados no están diseñadas para desarrollar una lógica de negocios compleja y flexible.

Integración de Instrucciones DDL y DML

Instrucciones DDL

Instrucción CREATE TABLE

Dentro de un **SP** podemos utilizar diferentes instrucciones DDL. Si queremos crear una tabla para almacenar datos temporales, debemos introducir la instrucción CREATE TABLE dentro para crear dicha tabla.

```sql
DELIMITER $$
CREATE PROCEDURE sp_crear_tabla()
BEGIN
    CREATE TABLE nombre_tabla (
        id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
        descripcion VARCHAR(200));
END $$
```

```sql
CALL sp_crear_tabla();
```

Instrucción ALTER TABLE

Si requerimos modificar una tabla porque su estructura cambia de forma frecuente, introducimos la instrucción ALTER TABLE dentro de un **SP**.

```sql
DELIMITER $$
CREATE PROCEDURE sp_modificar_tabla()
BEGIN
    ALTER TABLE nombre_tabla ADD COLUMN campo VARCHAR(50) NOT NULL;
END $$
```

```sql
CALL sp_modificar_tabla();
```

Instrucción DROP TABLE

Si necesitamos eliminar una tabla temporal, introducimos la instrucción DROP TABLE dentro de un **SP**.

```sql
DELIMITER $$
CREATE PROCEDURE sp_eliminar_tabla()
BEGIN
    DROP TABLE IF EXISTS nombre_tabla;
END $$
```

```sql
CALL sp_eliminar_tabla();
```

Instrucciones DML con parámetros

Instrucción INSERT

Dentro de un **SP** podemos utilizar diferentes instrucciones DML. Si queremos agregar un nuevo usuario llamado “DIEGO PEREZ”, debemos utilizar parámetros de entrada para que el **SP** reciba dichos datos. Estos datos, serán utilizados como valores en la instrucción INSERT.

```sql
DELIMITER $$
CREATE PROCEDURE sp_agregar_usuario(
    IN nombre VARCHAR(30), IN apellido VARCHAR(30))
BEGIN
    INSERT INTO usuario (nombre, apellido) VALUES (nombre, apellido);
END $$
```

```sql
CALL sp_agregar_usuario('DIEGO', 'PEREZ');
```

Instrucción UPDATE

También, podemos modificar los datos de una tabla. Por ejemplo, se necesita cambiar el nombre de un usuario llamado “DIEGO” por “PABLO”. Para esto, se utilizan parámetros de entrada para que el **SP** reciba dichos datos. Estos datos, serán utilizados como valores en la instrucción UPDATE.

```sql
DELIMITER $$
CREATE PROCEDURE sp_modificar_nombre_usuario(
    IN id INT, IN nombre VARCHAR(30))
BEGIN
    UPDATE usuario SET nombre = nombre WHERE id_usuario = id;
END $$
```

```sql
CALL sp_modificar_nombre_usuario(1,'PABLO');
```

Instrucción DELETE

Asimismo, podemos eliminar los datos de una tabla. Por ejemplo, se requiere borrar los datos de un usuario cuyo ID es 1. Entonces, se utilizan parámetros de entrada para que el **SP** reciba el valor del ID y tal valor será introducido en el WHERE de la instrucción DELETE.

```sql
DELIMITER $$
CREATE PROCEDURE sp_eliminar_usuario(IN id INT)
BEGIN
    DELETE FROM usuario WHERE id_usuario = id;
END $$
```

```sql
CALL sp_eliminar_usuario(1);
```

Instrucción SELECT Con IN – OUT

La instrucción SELECT nos permite listar los datos de una tabla. Por ejemplo, se quiere saber cuál es el nombre del usuario con ID de valor 1. Para esto, se recibe el ID en un parámetro de entrada y el resultado se almacena en un parámetro de salida con la cláusula INTO.

```sql
DELIMITER $$
CREATE PROCEDURE sp_dame_nombre_usuario(
     INOUT id INT, OUT nom VARCHAR(30))
BEGIN
     SELECT nombre INTO nom FROM usuario WHERE id_usuario = id;
END $$
```

```sql
CALL sp_dame_nombre_usuario(1,@nombre); -- Ejecución del SP y envía "1" como dato
SELECT @nombre; -- Muestra el resultado
```

Instrucción SELECT Con INOUT

Una variante del uso de esta instrucción podría ser utilizar un mismo parámetro para la entrada y la devolución del resultado. Por ejemplo, se quiere saber cuántos usuarios tienen en su nombre la letra "a".

```sql
DELIMITER $$
CREATE PROCEDURE sp_dame_nombre_usuario(INOUT valor VARCHAR(30))
BEGIN
      SELECT COUNT(*) INTO valor FROM usuario
      WHERE nombre LIKE CONCAT('%', valor, '%');
END $$
```

```sql
SET @letra = 'a'; -- Declaración y asignación de una variable (dato)
CALL sp_dame_nombre_usuario(@letra); -- Ejecución del SP y envía "a" como dato
SELECT @letra; -- Muestra el resultado
```

MÓDULO 5 - ORM

C25A - BASE DE DATOS DESDE BACK END

ORM

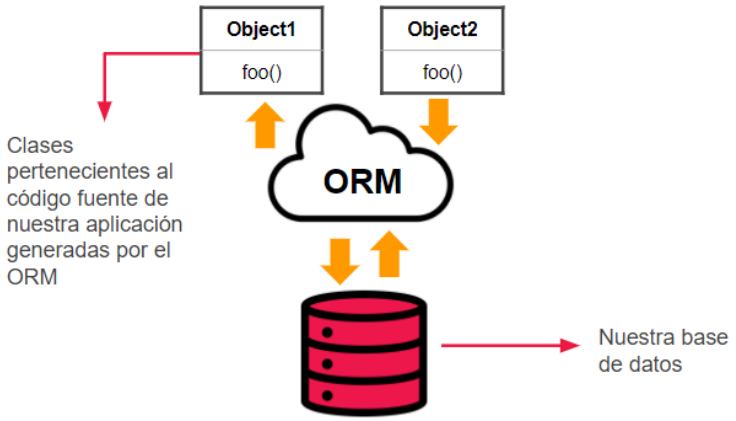
¿Qué es ORM?

El Object Relational Mapping, es un modelo de programación que permite mapear las estructuras de una base de datos relacional sobre una estructura lógica de entidades para simplificar y acelerar el desarrollo de nuestras aplicaciones.
Estas estructuras de bases de datos relacionales quedan vinculadas con esas entidades lógicas o base de datos virtual de tal modo que las acciones CRUD (Create, Read, Update, Delete) a ejecutar sobre la base de datos física, se realicen de forma indirecta por medio del ORM.
Cada modelo hace referencia a una tabla en nuestra base de datos. Estos modelos van a tener atributos, que son nuestras columnas a las cuales le vamos a tener que enseñar que tipos de datos pueden contener. De esta manera, le estamos brindando mayor seguridad al sistema al momento de llevar y traer datos.
Es una técnica que permite la relación de los objetos del código fuente de nuestra aplicación con los datos que ellos mismos representan en la base de datos.

¿Por qué usar ORM?

Debemos tener en cuenta que al usar ORMs existe una sobrecarga de código en nuestras aplicaciones para manejar los datos.
Por esta razón, ORMs está especialmente recomendado en:

- Aplicaciones con modelos de datos complejos.
- Aplicaciones donde el rendimiento no sea crítico.



Los ORMs son útiles cuando hay que gestionar transacciones —operaciones insert/update/delete—, pero cuando se trata de recolectar datos con resultados complejos, resulta muy importante evaluar la performance y la eficiencia de una herramienta ORM.

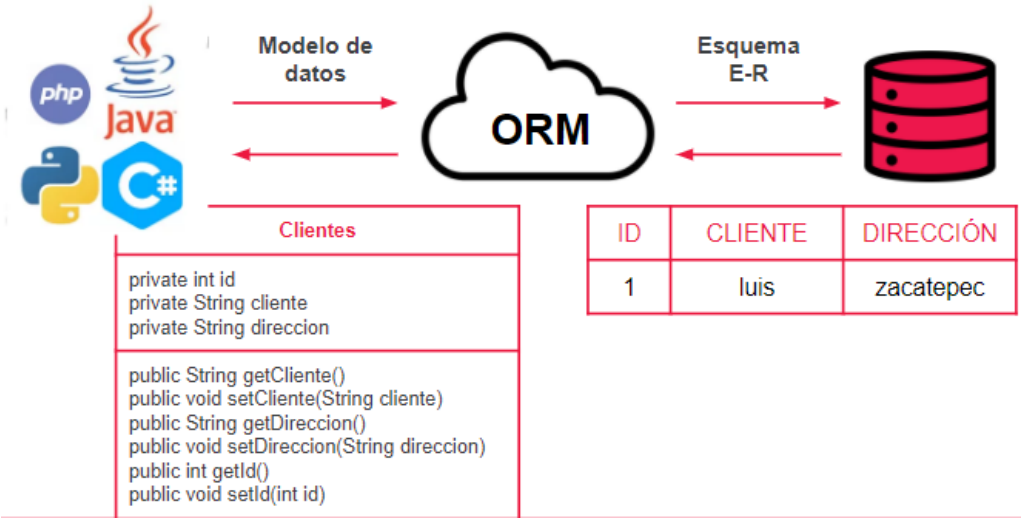
ORMs más conocidos

La mayoría de los ORM se fundan sobre las mismas bases e inclusive suelen trabajar de forma muy similar. Cada lenguaje de programación, puede usar su propio ORM.

Algunos ejemplos de ORMs son:

- Hibernate (Java)
- Entity Framework (.NET)
- Doctrine (PHP)
- SQLAlchemy (Python)
- ActiveRecord (Ruby)

Ejemplo



Ventajas y desventajas