**CURSO DE SQL – CODERHOUSE**

## **Clase 1:**

### **Introducción a bases de datos**

Definición: Conjunto de información, usualmente perteneciente a un mismo contexto, que es almacenada sistemáticamente para su posterior uso

Historia:

* **1884** año del origen de las DB y a Herman Hollerith como su creador
* En **1969**, Edgar Codd diseñó el modelo relacional, el más usado hasta hoy
* En la década del **70**, nace la primer BD comercial, Oracle
* Durante los **80’s**, las DB relacionales cobran gran popularidad
* **1960s** => Se acuña el termino de DATA BASE
* Nace el Modelo Relacional y lenguaje SQL “Sequel”
* **1970s** => Nace la base de datos ORACLE
* **1980s** => Las DB conquistan los servidores y PCs, Db2, SQL Server, dBase-Paradox, MS Access, Sybase SQL
* **1990s =>** Nace la POO y se incorpora a las DB
  + Las DB incorporan objetos
  + ANSI SQL normaliza el lenguaje
  + Nace Transact SQL
* **2000s** => Nacen las DB de tipo NO-SQL

Al Sistema de Gestión de DB (SGBD) o Database Management System (DBMS) nos podemos conectar de manera local, es decir dentro del equipo donde reside o de manera remota, en otro u otros equipos. Cuando se utiliza conexión remota se dice que se tiene un servidor de DB, se utiliza internet y un puerto habilitado para establecer la conexión.  
SGBD o DBMS se le denomina al conjunto de componente de software que permite realizar todo tipo de gestión y operaciones sobre la información que la base de datos almacena, como también sobre la DB en sí.

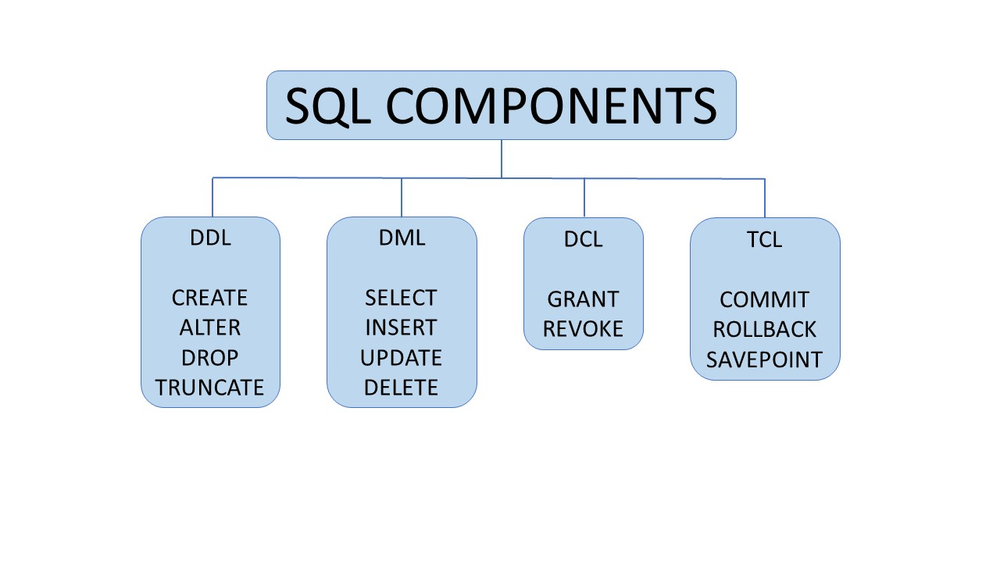
Operaciones sobre una DB: Las operaciones más frecuentes sobre una DB, se denominan bajo el término **C.R.U.D.** (*Crear, Leer, Actualizar y Eliminar*). Estas operaciones se pueden realizar sobre los datos almacenados y también sobre cualquier objeto que compone la base de datos.

**DB SQL**

* Son DB de tipo Relacionales y estructuradas.
* Su nombre SQL proviene del término (Structured Query Language), en español: Lenguaje de Consulta Estructurado.
* Se describe como un lenguaje declarativo.
* Pasó a ser el estándar del Instituto Nacional Estadounidense de Estándares (ANSI) en 1986 y de la Organización Internacional de Normalización (ISO) en 1987.

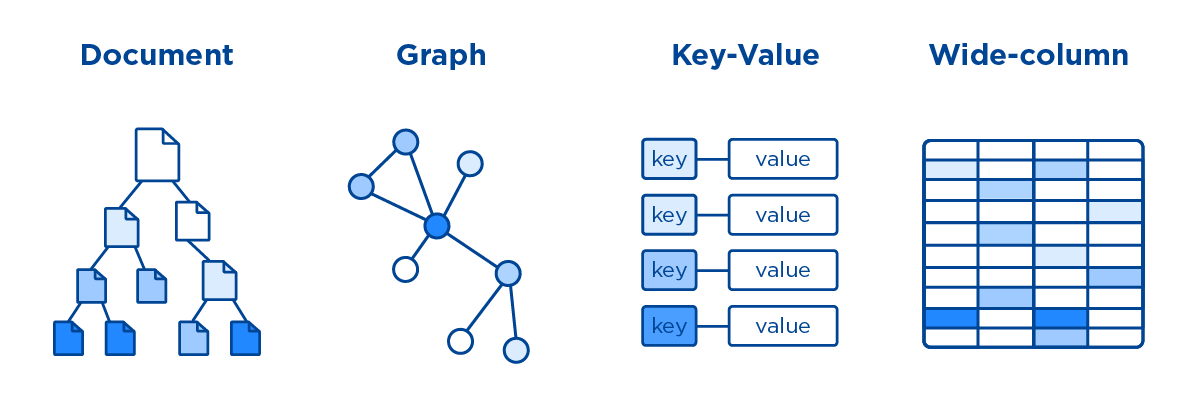
**Lenguaje y Sublenguajes**

* SQL DML: Data Manipulation Language (Lenguaje de Manipulación de datos)
* SQL DDL: Data Definition Language (Lenguaje de Definición de Datos)
* SQL DCL: Data Control Language (Lenguaje de control de datos)
* SQL TCL: Transaction Control Language (Lenguaje de Control de Transacciones)



**DB NoSQL**

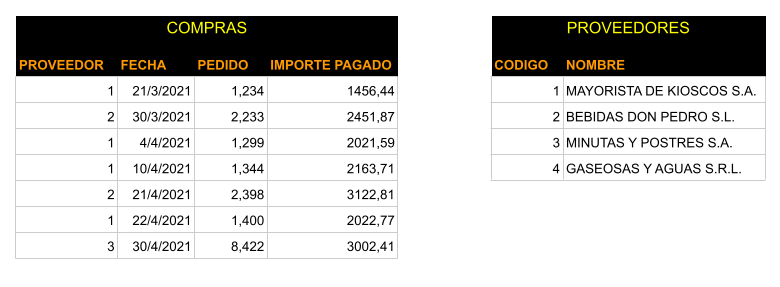
Surgen porque atacan algunas falencias/limitaciones de las DB relacionales, en particular se enfocan en mejorar, el rendimiento, la disponibilidad y la escalabilidad.



\*\* Una base de datos plana es un modelo de base de datos que almacena todos los datos en una sola tabla, compuesta por columnas y filas. Los registros se almacenan como filas de datos individuales, separadas por delimitadores como tabulaciones o comas. **EXCEL**\*\*

### **Bases de datos relacionales**

Las bases de datos relacionales se basan en el Modelo Relacional usando tablas para representar los datos y las relaciones entre ellos. Para transformar la base de datos plana en una base de datos relacional, esta sería una forma mucho más correcta de “normalizar” la información que representamos.



Ventajas:

* Evitar datos duplicados
* Optimiza el espacio de almacenamiento
* Reduce la complejidad del uso de las tablas

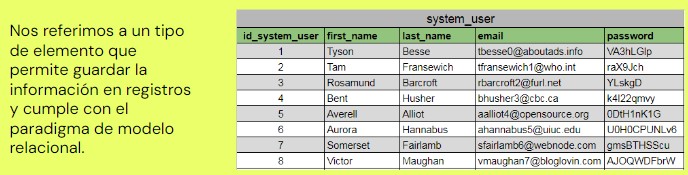
**Modelo Relacional**

El Modelo Relacional, en el cual se apoyan las Bases de Datos Relacionales, almacenan la información en un conjunto de tablas, y a su vez, las aprovechan para representar tanto los datos como también las relaciones entre cada una de ellas.

Ventajas:

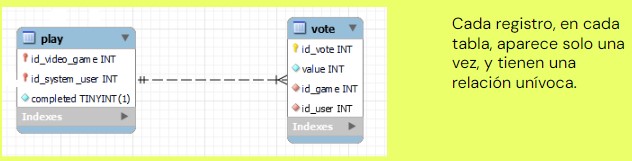
* Favorece el proceso de normalización, el cual permite eliminar la redundancia evitando la duplicidad de información en los registros guardados.
* Permite realizar consultas y obtener reportes de forma ágil y rápida por medio de SQL u otro lenguaje de base de datos estructurado.
* Se pueden crear una o varias relaciones entre las tablas.
* Garantiza la integridad referencial; si un registro se relaciona con otro registro de otra tabla, no permite que el mismo sea eliminado.
* Asimismo, si se quiere borrar, también pasará con todos los datos relacionados.

**Tablas**

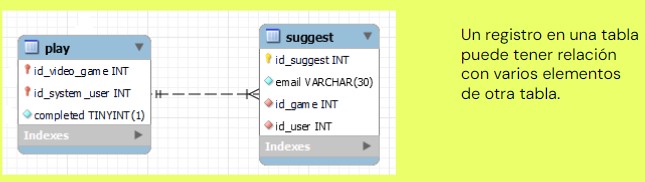


**Tipos de relaciones**

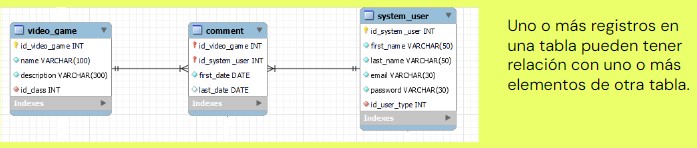
Uno a uno



Uno a Muchos



Muchos a Muchos



**Claves**

*Lógicas*

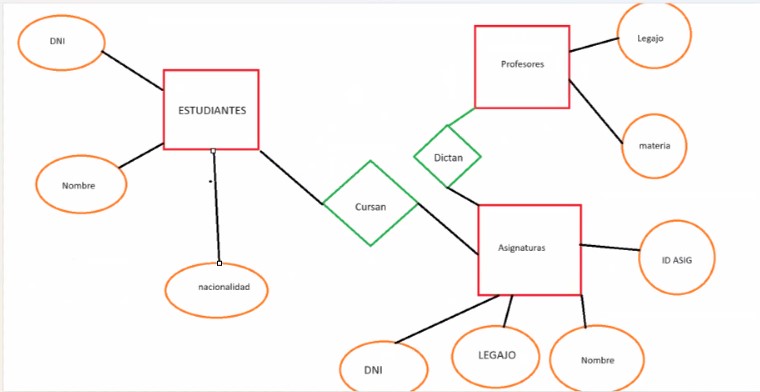
* **Clave primaria PK:** también llamada llave primaria o primary key, hace que el registro sea unívoco y obligatoriamente no nulo
* **Clave foránea FK:** también llamada foreign key, clave secundaria o clave externa, puede ser -o no- una clave primaria dentro de la tabla. Su característica es que es el punto de enlace con otra tabla donde ésta es primary key.
* **Clave índice:** es un campo que facilita la búsqueda dentro de una tabla. Generalmente son campos primary key.

*Conceptuales*

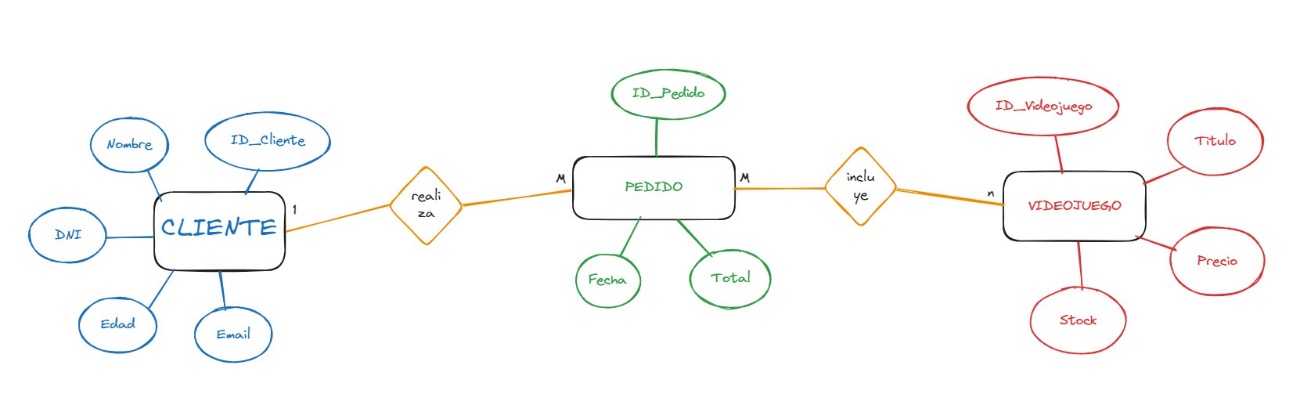
* **Clave concatenada CK:** ayuda a encontrar la singularidad en una tabla combinando dos campos, ya que no hay una llave primaria.
* **Claves candidatas:** cuando existe más de una clave primaria dentro de la tabla, por ejemplo, legajo y dni.

**Modelo ENTIDAD-RELACIÓN**

El modelo de datos entidad-relación (E-R) se basa en la percepción del mundo real que consiste en un conjunto de objetos básicos, denominados entidades, y de las relaciones entre esos objetos.



\*Mi propio Diagrama\*



## **Clase 2:**

### **Lenguaje SQL**

**Sintaxis SQL:** La sintaxis es el conjunto de reglas que deben seguirse al escribir el código SQL; para considerarse como correctas y así completar la ejecución exitosamente.

* Las sentencias SQL no son sensibles a las mayúsculas y minúsculas. No obstante, es importante respetarlas al colocar el nombre de un campo o tabla
* Cada sistema de bases de datos tiene sus particularidades sintácticas
* Sin embargo, si conocemos la base de SQL podremos adaptarnos sin dificultades
* Cada consulta finaliza con punto y coma (;)

**Uso de SELECT-FROM**  
La sentencia **SELECT**, como lo indica su nombre, permite seleccionar información a extraer y gracias a esto visualizar el resultado.

La cláusula **FROM** complementa al SELECT. Esta declara la/s tabla/s desde la/s cual/es se va a extraer la información.

* El orden de los campos en el SELECT es irrelevante.
* Podemos definir el que necesitemos en primer lugar, independientemente de la posición donde éste, haya sido definido cuando se creó la tabla.
* Al visualizar el resultado, el orden de los campos será tal como lo coloquemos en la consulta.
* El símbolo asterisco (\*) juega el mismo papel que cuando lo usamos para buscar archivos o carpetas. Representa a “todos” los que existan.

**SELECT DISTINCT**

La cláusula DISTINCT funciona en conjunción con SELECT. Permite filtrar de una consulta aquellos registros repetidos del resultado de la misma.  
Para utilizar la sentencia SELECT DISTINCT, debemos incorporar siempre el nombre de al menos un campo de la tabla.

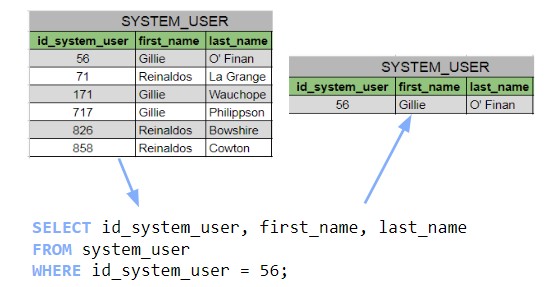
**Operadores de Comparación**

Los operadores de comparación en SQL nos permiten evaluar una condición y determinar si el resultado es verdadero, falso o desconocido (TRUE, FALSE o UNKNOWN). En SQL se utilizan estos operadores combinados con la sentencia WHERE, que veremos a continuación.



**Sentencia WHILE** (Aplicar condicionales)

La sentencia WHERE permite agregar condiciones para filtrar los resultados. Obtendremos únicamente los registros que cumplan con dichas condiciones.



Las sentencias tienen un orden para su correcto funcionamiento. Cuando tenemos una sintaxis incorrecta, el SGBD nos presenta el error y en la mayoría de los casos lo hace de forma explícita.

1. SELECT (campos)
2. FROM (tabla)
3. WHERE (condición/es)

### Sublenguajes SQL

Practica con operadores (practicaConOperadores.sql)

**Ordenamiento de datos (OrderBy)**

Así como podemos consultar, seleccionar qué mostrar y cuáles datos filtrar, SQL nos brinda también la posibilidad de ordenar la información.  
Formato:

1. SELECT (campos)
2. FROM (tabla)
3. WHERE (condición/es)
4. ORDER BY (columna)
5. ASC

Ejemplos:

* SELECT id\_level, name FROM game where id\_level < 30 Order By id\_level asc; -- por default es "asc"
* SELECT id\_level, name FROM game where id\_level < 30 Order By id\_level desc;
* SELECT id\_level, name FROM game where id\_level < 30 Order By id\_level, name desc;

**Limit**

La utilizamos al final de toda la sentencia SELECT, para restringir el número de filas en el resultado de la consulta. LIMIT espera uno o dos parámetros:

* desde qué registro comenzar a mostrar.
* el total de registros próximos a mostrar.

**Alias**

SQL Alias es una forma de acotar el nombre de una tabla o columna, simplificando así su uso en sentencias SQL. Se logra reducir las sentencias SQL cuando incluyen dos o más tablas y/o varios campos. Se debe usar la palabra reservada AS, seguida del alias que se desea dar a dicho campo o tabla.

**SELECT**

     su.id\_system\_user **AS** id,

     su.last\_name **AS** l\_n,

     su.password **AS** pass

**FROM** system\_user su

**ORDER BY** su.id\_system\_user;

**Funciones de agregación**

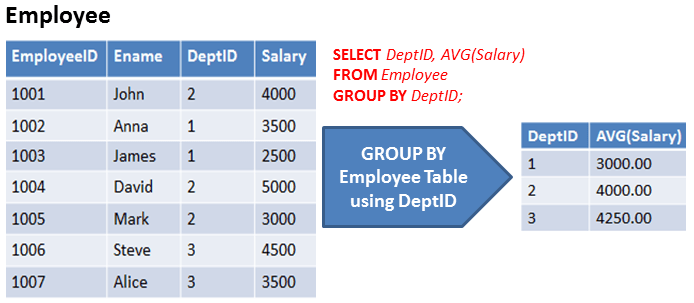
Así como SQL nos permite obtener datos de una o más tablas, también nos permite obtener valores simplificados o resumidos, sobre datos específicos que necesitemos. Esto se conoce como Funciones de Agregación o Agrupación. Podemos combinar funciones de totalización, conteo, promedios, valores mínimos y/o máximos, entre otras, al momento de realizar la consulta.

Las funciones de agregación se combinan con la cláusula GROUP BY y el uso de AS.

* count()
* max()
* min()
* sum()
* avg()

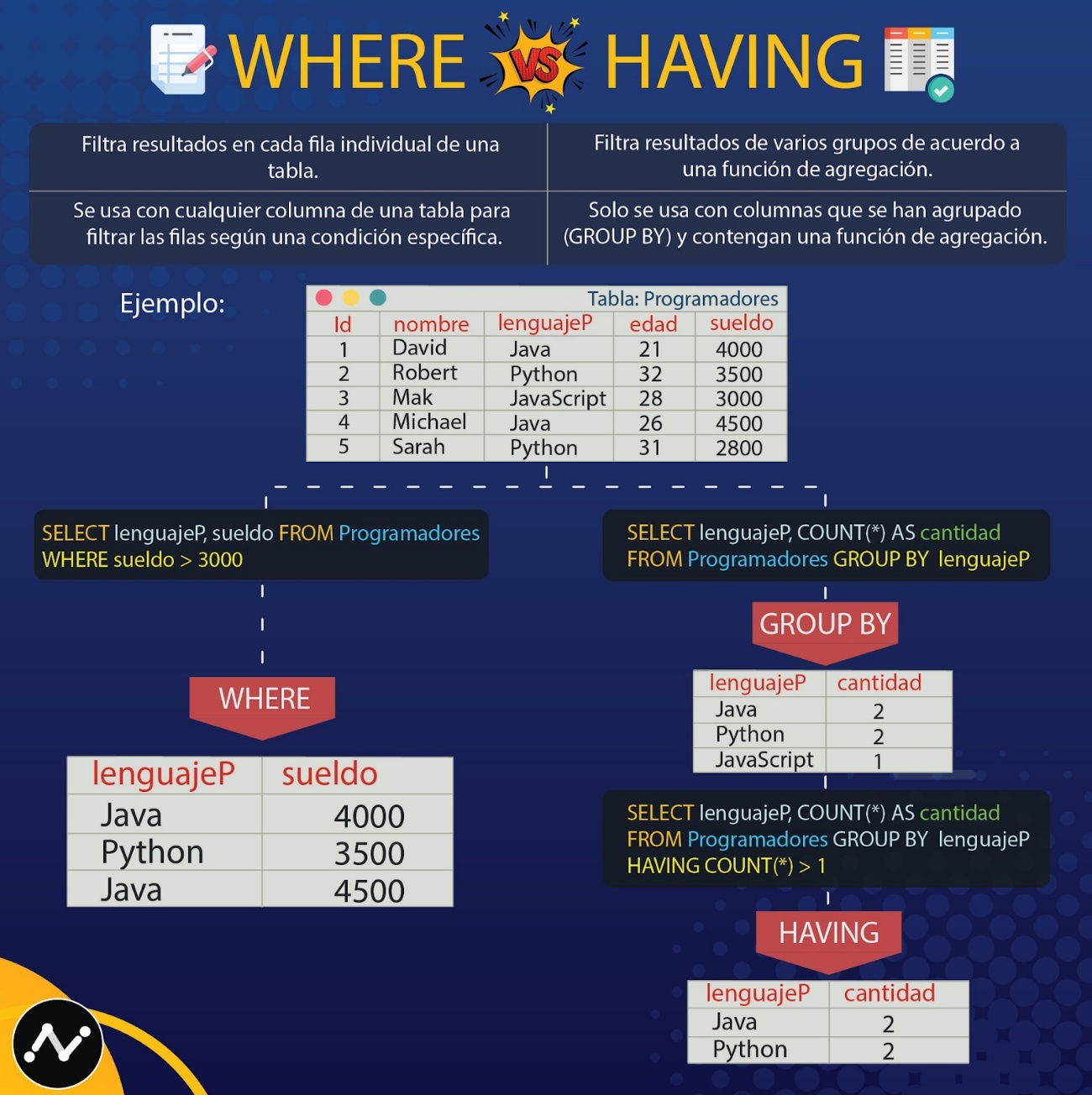
**Group By**

La cláusula GROUP BY es fundamental para usarse junto a las funciones de agregación, la debemos utilizar cuando debemos obtener información que nace de la agrupación de registros. Por lo tanto, será el aliado ideal para usarse junto a COUNT(), SUM() y AVG().



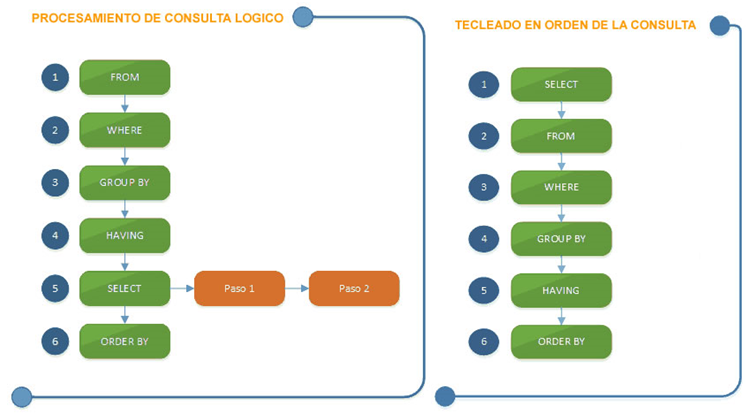
**HAVING**

HAVING, al igual WHERE, permite establecer condiciones para filtrar los resultados. Para ello, necesitamos generar campos con resultados filtrados, para luego sumar a HAVING. Por lo tanto, debemos tener presente que esta sentencia solo funciona con campos generados a partir de una función. El HAVING siempre se le aplica a un Group By.



**Orden de Sintaxis**

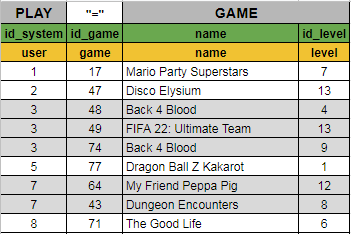
1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. LIMIT

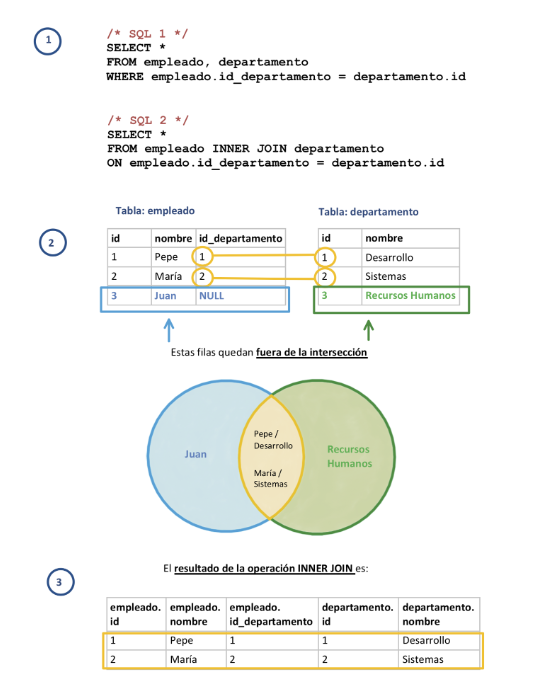
****

**JOIN**

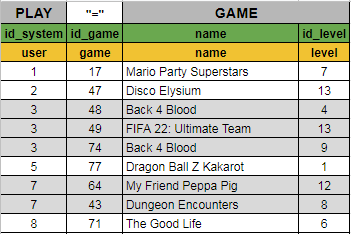
JOIN permite combinar registros de diferentes tablas, complementándose con la cláusula ON, la cual establece la condición por la cual queremos unir las tablas. Generalmente son campos comunes entre tablas. Los tipos de JOIN importantes, son cuatro: Inner Join, Left Join, Right Join, Full Join.

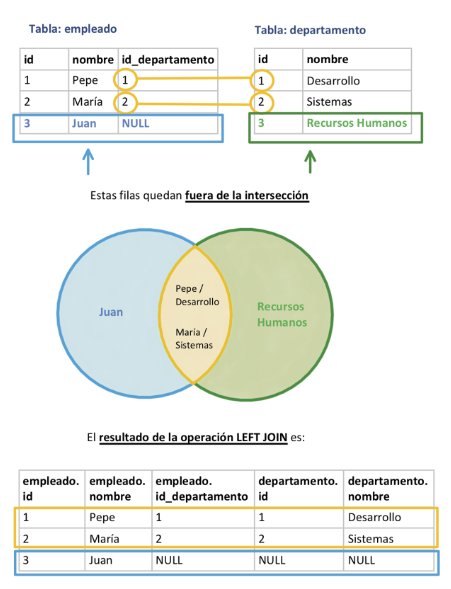
Inner Join: INNER JOIN, o JOIN, retorna todas las filas de las dos tablas siempre que haya coincidencia por el campo declarado en el ON. El resultado es NULL cuando no hay coincidencia alguna.



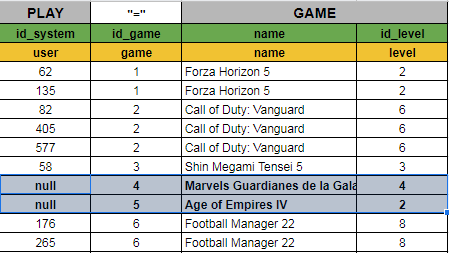


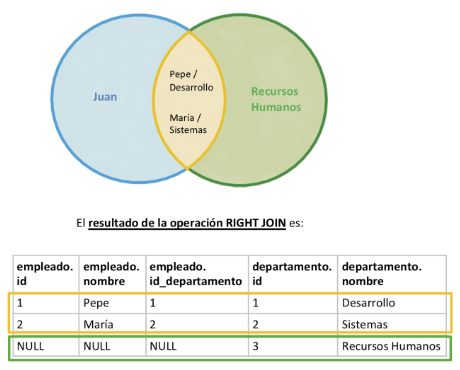
Left Join: LEFT JOIN retorna todas las filas de la tabla izquierda que coincidan con las filas de la tabla derecha. El resultado es NULL del lado derecho, cuando no hay coincidencia.



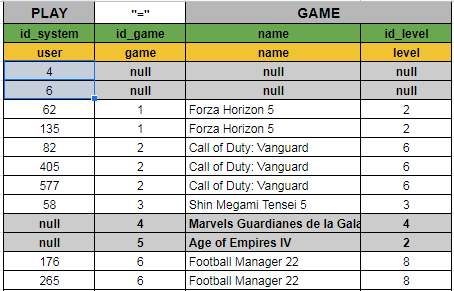


Right Join: RIGHT JOIN retorna todas las filas de la tabla derecha que coincidan con las filas de la tabla izquierda. El resultado es NULL cuando no hay coincidencia del lado izquierdo





Full Join: FULL JOIN retorna todas las filas de la tabla derecha y también las filas de la tabla izquierda. Básicamente combina los resultados de LEFT y RIGHT JOIN, pudiendo tener valores nulos de ambos lados. Nota: MySQL no soporta FULL JOIN.



## **Clase 3:**

### **Consultas y subconsultas SQL**

**Union**

El operador UNION combina los resultados de dos o más consultas en un único resultado que incluye todas las filas que pertenecen a todas las consultas que aparecen. Es decir, las consultas se ejecutan por separado, concatenando luego los resultados de cada una.

SELECT id\_game, name, description, id\_level, id\_class

FROM game

WHERE id\_level = 1

UNION

SELECT id\_game, name, description, id\_level, id\_class

FROM game

WHERE id\_level = 2;



UNION

* **Elimina Duplicados:** El operador UNION combina los resultados de dos o más consultas SQL y elimina los registros duplicados de los resultados finales. Esto significa que solo se muestran los registros únicos.
* **Uso de Recursos:** Debido a que UNION elimina duplicados, consume más recursos del sistema ya que necesita realizar una comparación y eliminación de duplicados.
* **Orden:** Aunque UNION elimina duplicados, no garantiza el orden de los resultados a menos que se utilice una cláusula ORDER BY al final del conjunto de consultas combinadas.

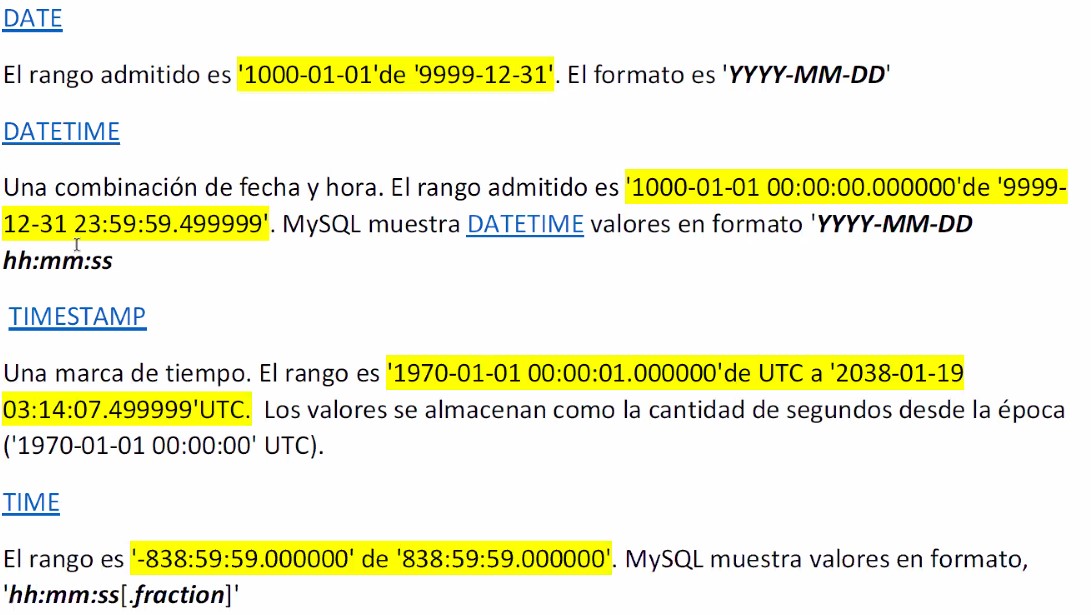
UNION ALL

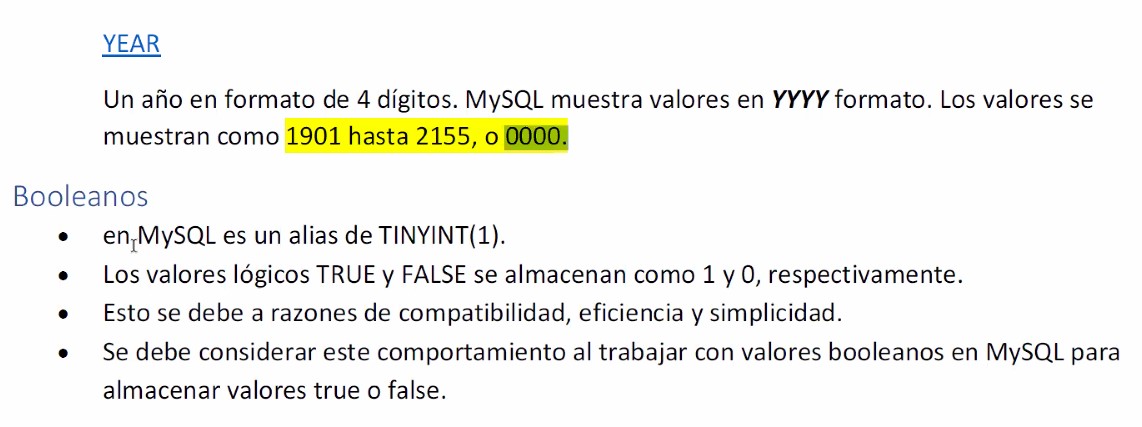
* **Incluye Duplicados:** El operador UNION ALL también combina los resultados de dos o más consultas SQL, pero a diferencia de UNION, no elimina los registros duplicados. Muestra todos los registros, incluidos los duplicados.
* **Uso de Recursos:** UNION ALL es más eficiente en términos de rendimiento y uso de recursos porque no necesita realizar la eliminación de duplicados.
* **Orden:** Al igual que UNION, UNION ALL no garantiza el orden de los resultados a menos que se utilice una cláusula ORDER BY al final del conjunto de consultas combinadas.

**Tipos de datos**

|  |  |  |
| --- | --- | --- |
| **Tipo de dato** | **valor SQL** | **Ejemplo** |
| Número entero | int | 1000 |
| Texto | text(n) | Coderhouse |
| Alfanumérico | varchar(n) | AB123CD |
| Fecha | date | 21/03/1975 |
| Fecha y Hora | datetime | 21/01/1972 15:00:00 |
| Verdadero o Falso | boolean | TRUE ó FALSE |
| Decimal | decimal(p, s) | 3008,05 |
| Numérico | numeric(p, s) | 1407,96 |

*Ejemplo de uso del DATE*





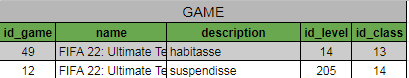
**LIKE ‘%’**

La implementación de este operador se realiza sobre campos del tipo texto o alfanuméricos, para buscar parte de un valor coincidente.

SELECT \*

FROM game

WHERE name LIKE ‘%Ultimate%’;

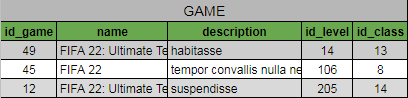


E implementando el caracter comodín “\_”, podemos también definir el desconocimiento de un solo caracter. Se puede combinar con el caracter %.

SELECT \*

FROM game

WHERE name LIKE ‘\_IFA%’;



\*\*practica en Workbench sobre like\*\*

**Subconsultas SQL**  
  
Obtener datos de una tabla con una subconsulta. Si debemos incluir en la cláusula WHERE algún criterio de selección que existe en otra tabla, las subconsultas son el elemento ideal que nos permitirá recuperar los valores acordes a dicha condición.

Para poder llevar a cabo esto de manera exitosa, debemos tener en cuenta las siguientes reglas:

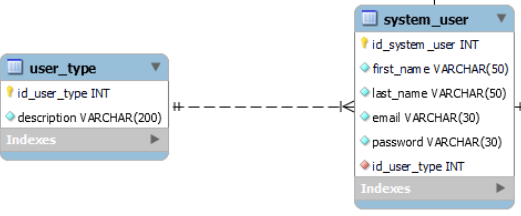
* La subconsulta debe ir entre paréntesis.
* La subconsulta debe tener una sola columna o expresión.
* No podemos utilizar BETWEEN o LIKE en la subconsulta.
* No debemos colocar la cláusula ORDER BY en la subconsulta.
* Otras cuestiones más con UPDATE y DELETE, que veremos oportunamente cuando abordemos dichos temas.

*Ejemplo de subconsulta*

SELECT id\_system\_user, last\_name

FROM system\_user

WHERE id\_user\_type = (SELECT max(id\_user\_type) FROM user\_type);



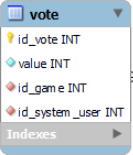
**Subconsultas en una misma tabla**

También podemos abordar subconsultas dentro de una misma tabla.

En este caso, la tabla vote cuenta con información del puntaje que cada usuario le dio a un juego en la columna value. Busquemos los usuarios que votaron con un puntaje superior al promedio. Nota: La función floor convierte float a entero.

SELECT id\_system\_user

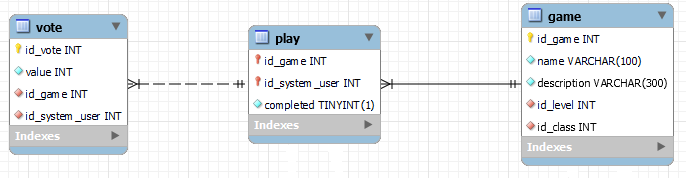
FROM vote WHERE value > (SELECT FLOOR(AVG(value)) FROM vote);



**GroupBy en consultas y subconsultas**

Acoplamos la sentencia GROUP BY dentro de una consulta con subconsulta asociada.

Veamos que nuestro ejemplo presta más información que antes, destacando ahora en qué se desempeña cada uno de ellos.



Obtener la suma de votos por juego, solo de aquellos juegos de nivel 1.

SELECT id\_game, SUM(value) AS votos

FROM vote

WHERE id\_game IN (SELECT id\_game

FROM game WHERE id\_level = 1)

GROUP BY id\_game;

**HAVING en consultas y subconsultas**

Seleccionaremos los juegos, pero sólo aquellos que hayan tenido más de un voto.

SELECT id\_game, name

FROM game

WHERE id\_level = 1 AND

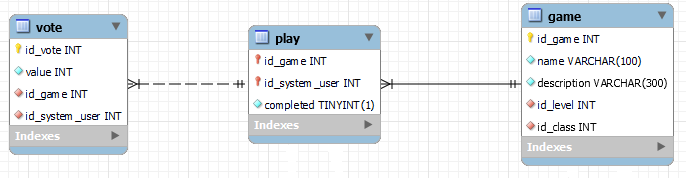
id\_game IN

(SELECT id\_game

FROM vote

GROUP BY id\_game

HAVING count(\*) > 1);



**SQL Performance Tuning**

La optimización de consultas es el proceso de escribir la consulta de manera que pueda ejecutarse rápidamente. Por ejemplo, si nuestra tabla game tiene 100000000000 filas de datos y necesitamos averiguar la fila donde el id\_game es 77 , existen tres formas diferentes de encontrar el resultado:

* O(1): Índice hash en id, o resultado almacenado en caché en id\_game= 77 de una consulta anterior.
* O(n): Si hacemos un escaneo completo de la tabla game y miramos todas y cada una de las filas, entonces tomaría mucho tiempo.
* O(log(n)): Si ordenamos la tabla y hacemos una búsqueda binaria en esta tabla, entonces necesitamos alrededor de 36 búsquedas (la base de registro 2 de 100000000000 es ~ 36) para averiguar nuestro valor, pero ordenar la tabla llevaría un tiempo (depende totalmente del optimizador del motor SQL).

La opción con menor tiempo de respuesta sería O(1).

**Buenas Prácticas**

1. **Usar SELECT \* FROM solo si es necesario**

No utilicemos ciegamente SELECT \* en el código. Si hay muchas columnas en la tabla, se devolverán todas, lo que ralentizará el tiempo de respuesta, especialmente si envía el resultado a una aplicación front-end.

Es recomendable escribir explícitamente los nombres de las columnas que realmente se necesitan.

1. **Usar la cláusula ORDER BY sólo si es necesario**

Si deseamos mostrar el resultado en la aplicación frontal, dejar que ORDENE el conjunto de resultados. Hacer esto en SQL puede ralentizar el tiempo de respuesta en el entorno multiusuario.

1. **Usar UNION ALL en vez de UNION**

La cláusula UNION ALL responde más rápido que UNION.

UNION ALL busca filas duplicadas, mientras que la declaración UNION lo hace independientemente de si existen o no.

1. **Usar la cláusula EXISTS donde sea necesario**

Si deseamos comprobar la existencia de datos, no es recomendable utilizar:

**IF** (**SELECT** **COUNT** (\*) **FROM** TABLA **WHERE** Columna=**’Algún Valor’**)>0

En su lugar, podemos usar la cláusula EXISTS:

**IF EXISTS** (**SELECT** **COUNT** (\*) **FROM** TABLA **WHERE** Columna=**’Algún Valor’**)

### **Sublenguaje DDL**

Fundamentos de Data Definition Language: Se ocupa de modificar la estructura de objetos de una DB. Lo conforman diferentes sentencias que nos permiten crear, modificar, borrar o definir la estructura de las tablas que almacenan datos.

Las sentencias disponibles a través de DDL, son:

* CREATE
* ALTER
* DROP
* TRUNCATE

Con ellas creamos, modificamos, alteramos y eliminamos objetos.

**DDL: CREATE**

La sentencia CREATE cumple la función de crear nuevos objetos en la base de datos. Los tipos de objetos a crear pueden ser: tablas, índices, stored procedures y hasta nuevas bases de datos, además, usuarios específicos.

Sentencia => CREATE TABLE [nombre de la tabla] (

[definiciones de columnas]),

([parámetros de la tabla]);

* [nombre de la tabla]: definimos el nombre distintivo de la tabla a crear. Ej: friend.
* [definiciones de columnas]: definimos las columnas o campos y sus propiedades o tipo de datos.
* [parámetros de la tabla]: definimos otras particularidades de la tabla como, por ejemplo, los índices.

## **Clase 4:**

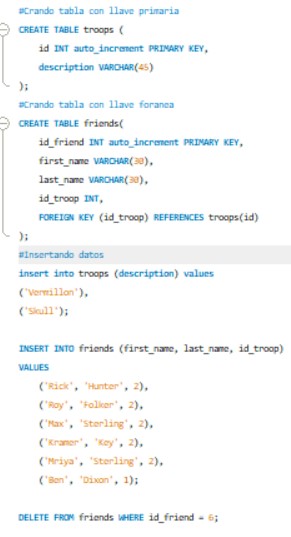
### **Objetos de una base de datos**

Objetos: Todos los componentes que conforman a una base de datos.

* Tablas
* Funciones
* Trigger
* Stored Procedures

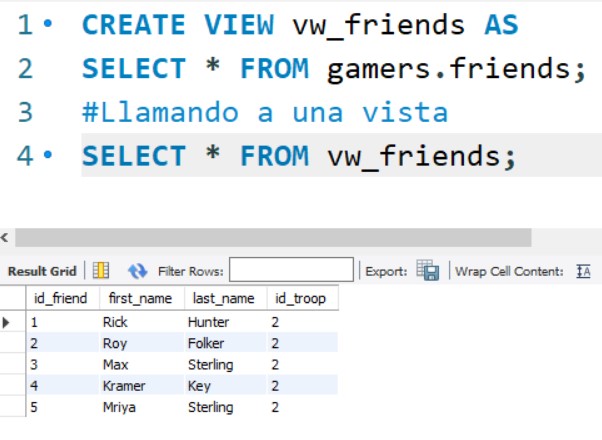
**Tablas (*entidad*):**

Ya sabemos que las tablas se ocupan de almacenar la información en forma de registros. Cada tabla puede trabajar de forma autónoma, aunque en una DB relacional suele establecerse al menos una relación entre la tabla más importante y una o más tablas secundarias, terciarias, etcétera.

*Ejemplo en vivo:*  
****

**Normalización de datos:**  
  
 **VISTAS:**

* Una Vista es un conjunto de resultados de una tabla o más tablas de un DB.
* Podemos definirlas también como “una tabla virtual”, que se genera a partir de una o más tablas de una BD relacional.  
  Están compuestas por la misma estructura que una tabla: filas y columnas.
* Pueden ser almacenadas con el mismo nombre de una tabla, o si se combinan dos o más tablas en la vista se suele definir combinando ambos nombres.

Las vistas nos sirven para entornos de **pre-producción** ya que son NO editables, solo son para visualizaciones.  


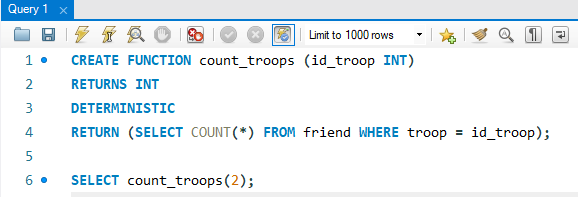
**FUNCIONES:**

Estas permiten crear una rutina específica que procese determinados parámetros, y retornar un resultado determinado. Las funciones de usuario utilizan el lenguaje SQL, y permiten incluir sentencias propias creadas por el desarrollador, como también combinar funciones SQL preexistentes.

Podemos combinar estas últimas para crear resultados personalizados que las funciones integradas no puedan resolver.

Aceptan sólo parámetros de entrada:

* Deben retornar siempre un valor con un tipo de dato definido.
* Pueden usarse en el contexto de una sentencia SQL.
* Retornan un valor individual, y nunca un conjunto de registros.

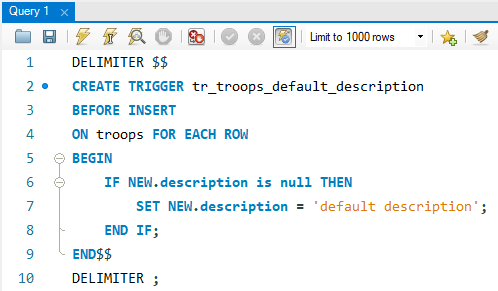
Si desarrollas en algún lenguaje de programación, encontrarás un parecido con las funciones personalizadas que creas en cualquier otro programa.  


**TRIGGERS:**

Un trigger es una aplicación almacenada (stored program), creado para ejecutarse cuando uno o más eventos ocurran en nuestra base de datos. El Trigger se dispara cuando ocurre un comando INSERT, UPDATE o DELETE, ejecutando un bloque de instrucciones que proteja o prepare la información de las tablas.

La principal tarea de un trigger es la de mantener la integridad de una bb.dd. aplicando los siguientes casos de uso:

* Validar la información
* Calcular atributos derivados
* Seguir movimientos y Logs



Entendiendo la lógica de los Triggers y cómo aplicarlos, pensemos un esquema (o dos) donde debamos aplicar un Trigger utilizando AFTER, y uno utilizando BEFORE.

Junto con el esquema a pensar, debemos contemplar también la o las tablas involucradas que serán afectadas y/o escuchadas por el trigger.

**STORED PROCEDURE:**

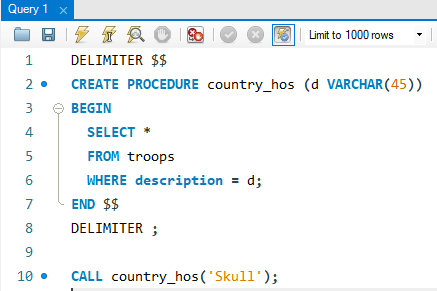
La ejecución de un Stored Procedure no está disponible para cualquier usuario. Es necesario que el perfil de éste, en el esquema de base de datos, tenga habilitado el permiso de ejecución (Execute). Un SP puede contener y ejecutar en su interior cualquier consulta del tipo DML. Incluso puede combinar varias de estas, aplicándolas en diferentes tablas.

Se inicia con el comando CREATE PROCEDURE nombre\_sp. Recibe parámetros del tipo IN, OUT e INOUT y soporta tipos de datos válidos.

Y a su vez, los SP pueden tener dos tipos de denominación:

* Determinista: Se los denomina así cuando el SP produce el mismo resultado sobre los mismos parámetros de entrada.
* No determinista: Cuando produce resultados diferentes a los tipos de parámetros de entrada.

Finalmente, un SP se ejecuta siempre del lado del servidor, y devuelve los datos filtrados y procesados al cliente que los solicitó.



### **Tablas SQL**

Las tablas almacenan la información en forma de registros o tuplas. Para ello, respetan la estructura de cada dato de un registro, el cual condice con la definición del campo que lo almacena.

En el panel Navigator-Schemas de Mysql Workbench, encontrarán un árbol de características desplegable, que permite acceder a las propiedades de una tabla donde encontrarán:

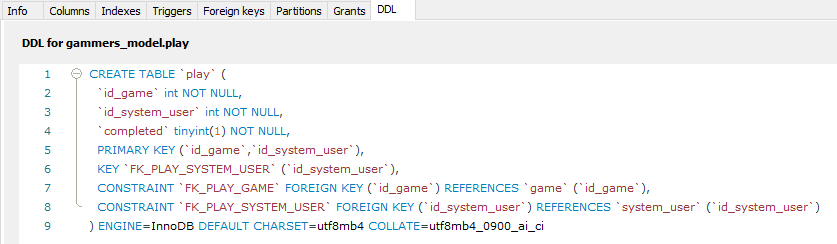
* Columns
* Indexes
* Foreign Keys
* Triggers

También realizando clic derecho sobre la tabla se tiene la herramienta Table Inspector, donde encontramos el apartado diferentes solapas (Info, Columns, Indexes, Triggers, entre otras), con posibilidad de ver el detalle completo de sus valores.

**Tablas transaccionales**

De todos ellos, InnoDB y BDB son motores transaccionales, siendo el primero el más elegido en Mysql, gracias a que InnoDB soporta bloqueos y capacidades de COMMIT, así como ROLLBACK, además de Recuperación ante Fallas.

InnoDB también permite bloqueo a nivel de filas, mejorando su rendimiento y soportando múltiples usuarios en simultáneo. También fue diseñado para procesar de manera ágil grandes volúmenes de datos.



**Tablas de hecho**

## Clase 5:

### **Vistas SQL**

Tal como mencionamos algunas clases atrás, una Vista SQL es básicamente una tabla virtual que se genera a partir de la ejecución de una o más consultas SQL, aplicada sobre una o más tablas. Su estructura corresponde a una serie de filas y columnas tal como encontramos en las tablas SQL, que almacenan la vista de la información tal como la definimos al crearla.

Su sintaxis está compuesta por:

* La sentencia CREATE VIEW
* El nombre que deseamos darle a la Vista
* Las columnas que se crearán
* La consulta SQL desde donde obtendrá los datos

CREATE VIEW nombre\_vista

[lista\_columnas]

AS consulta\_sql

Su sintaxis está compuesta por:

La sintaxis CREATE VIEW cuenta con un modificador (opcional) denominado OR REPLACE. Esta sintaxis se ocupa de, crear la Vista si no existe, o reemplazar la existente por una nueva.

CREATE VIEW OR REPLACE nombre\_vista

[lista\_columnas]

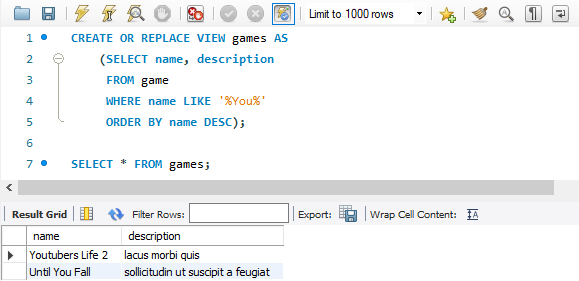
AS consulta\_sql

**Beneficios:**

* *Privacidad de la información:* los usuarios podrán ver sólo aquellos datos que creamos convenientes mostrar, en otras palabras, mejora la seguridad de la DB.
* *Rendimiento de la DB:* Crear queries sobre vistas complejas nos ahorra ejecutar una query pesada para llegar a la información.
* *Protección de datos:* Aquellos usuarios que no poseen un entorno de pre-producción, las vistas evitan errores de borrado o alteración.

**Aplicar filtros:**

Si aplicamos una cláusula WHERE y/u ORDER BY, veremos que los datos de la Vista, se listaran tal como lo indicamos en la consulta de selección. Por lo tanto, ¡en las vistas podemos aplicar todas las operaciones que vimos sobre tablas!



**Eliminar una view:**

También contamos con la posibilidad de eliminar aquellas Vistas que ya no utilizamos, haciendo uso del comando DROP VIEW <nombreDeLaVista>. Ten presente que la eliminación, mediante un script, será instantánea.

DROP VIEW vista\_productos;

### **WORKSHOP I**

* Definir nuestro proyecto.
* Armar el Diagrama E-R.
* Definir las tablas que contendrá (aplicando la normalización de datos).
* Crear las tablas estableciendo relaciones e índices (CREATE TABLE).
* Cargar datos en las tablas (mediante Mysql Workbench).
* Realizar consultas simples (WHERE, ORDER BY).
* Realizar consultas complejas (GROUP BY, JOIN, UNION, Subconsultas).
* Implementar al menos una función.
* Crear dos Vistas de datos (CREATE VIEW) - (puede ser implementando funciones escalares y/o transformación).
* Eliminar una tabla y/o sus datos cargados (DROP TABLE o TRUNCATE).

**Idea Inicial:**

* Elige bien cuál va a ser tu proyecto final.
* Investiga (buscador web mediante) y conoce diferentes proyectos realizados con bases de datos.
* Elige un rubro convencional y de baja complejidad.
* Si trabajas, aprovecha y analiza tu nicho de negocio para proyectarlo como Trabajo Final (te sentirás más cómodo porque ya conoces la temática).
* Importante los circuitos de trabajo complejos implican Tablas relacionales complejas (elegí el camino feliz obviando las trabas).

**Paper prototyping:**

* Hace referencia al paradigma de prototipado en papel.
* Es una forma efectiva de comenzar un proyecto haciendo borrón y cuenta nueva tantas veces sea necesario, antes de llevarlo al formato digital.

**Buenas Prácticas:**

* La normalización de datos es una de ellas. Busca información o consulta con tutores y/o docente ante cualquier mínima duda que tengas.

**Convenciones:**

Como parte de las buenas prácticas, las convenciones de nombre son claves en todo diseño:

* Al igual que en el mundo de la programación, evita caracteres extendidos -/ª!”·$%&&()?¿\*^¨¨:;çÇ{}[´]
* No uses todo mayúsculas en nombres de Tablas, Vistas y/o Campos
* Evita en el diseño, utilizar palabras o nombres de objetos con Ñ, ñ, acentos: á, é, í, ó, ú, diéresis, etcétera

## **CLASE 6:**