**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.

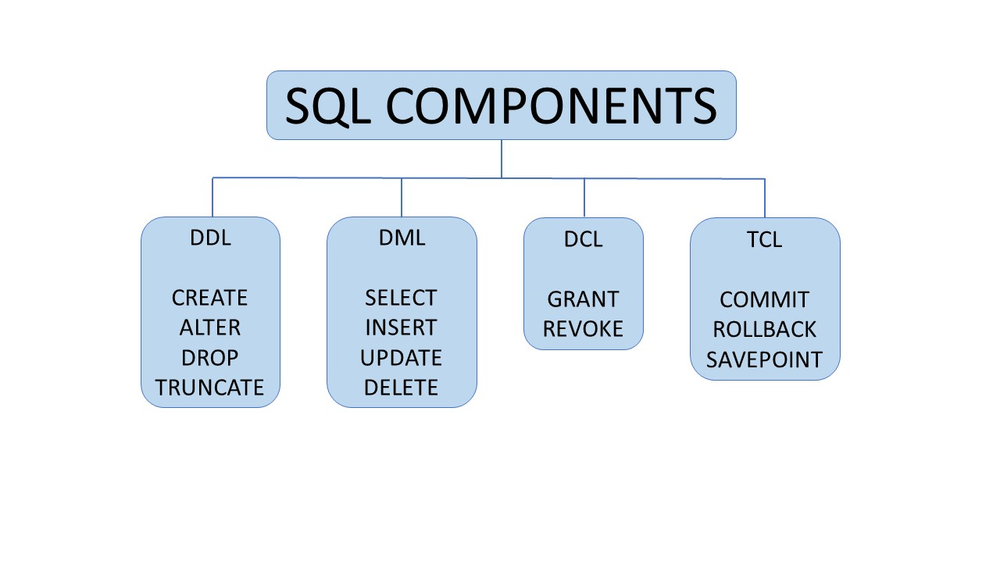
Operaciones sobre una DB: Las operaciones más frecuentes sobre una DB, se denominan bajo el término **C.R.U.D.** (*Crear, Leer, Actualizar y Eliminar*). Estas operaciones se pueden realizar sobre los datos almacenados y también sobre cualquier objeto que compone la base de datos.

**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.

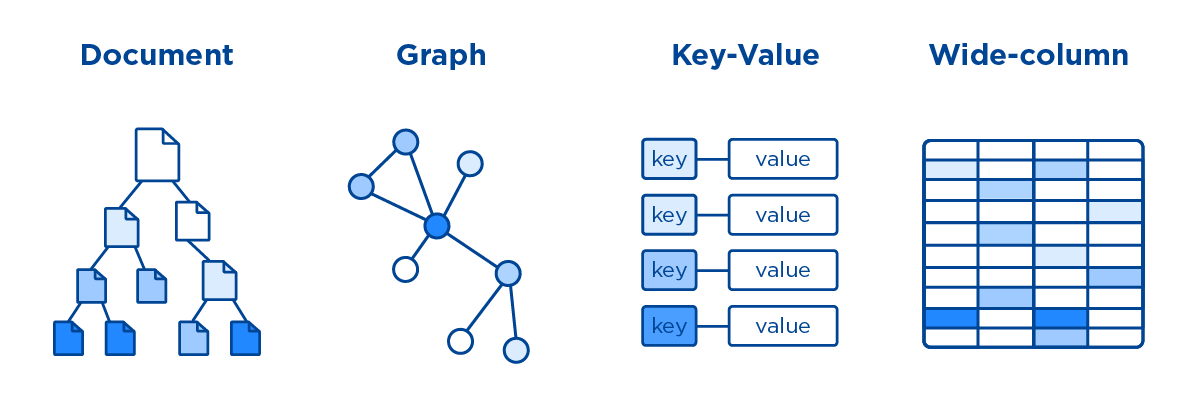
**Lenguaje y Sublenguajes**

* 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**

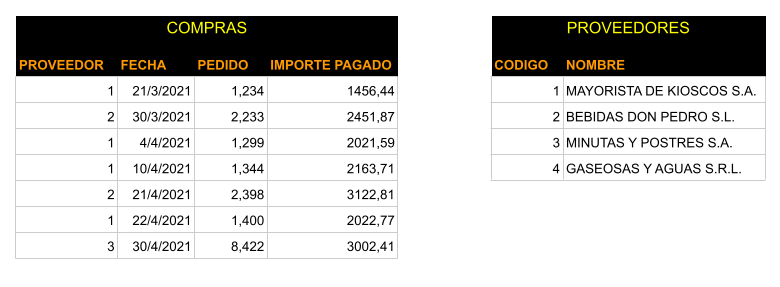
Surgen porque atacan algunas falencias/limitaciones de las DB relacionales, en particular se enfocan en mejorar, el rendimiento, la disponibilidad y la escalabilidad.



\*\* 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

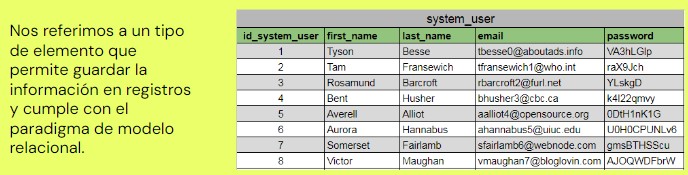
**Modelo Relacional**

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:

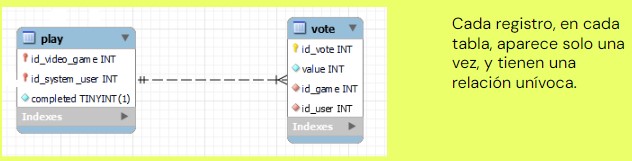
* Favorece el proceso de normalización, el cual permite eliminar la redundancia evitando la duplicidad de información en los registros guardados.
* Permite realizar consultas y obtener reportes de forma ágil y rápida por medio de SQL u otro lenguaje de base de datos estructurado.
* Se pueden crear una 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**

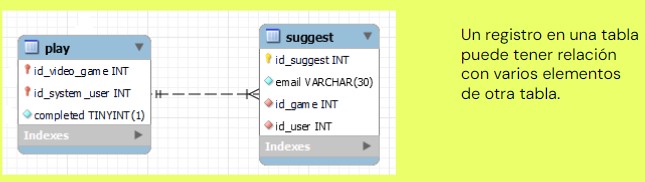


**Tipos de relaciones**

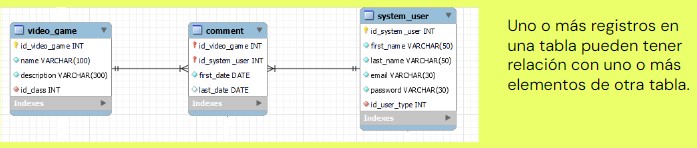
Uno a uno



Uno a Muchos



Muchos a Muchos



**Claves**

*Lógicas*

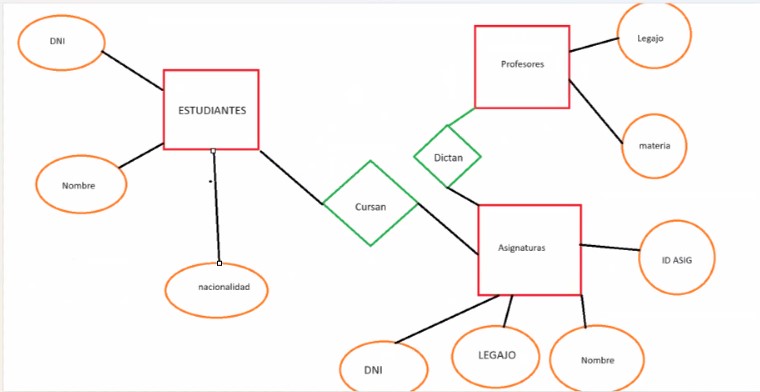
* **Clave primaria PK:** también llamada llave primaria o primary key, hace que el registro sea unívoco y obligatoriamente no nulo
* **Clave foránea FK:** también llamada foreign key, clave secundaria o clave externa, puede ser -o no- una clave primaria dentro de la tabla. Su característica es que es 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*

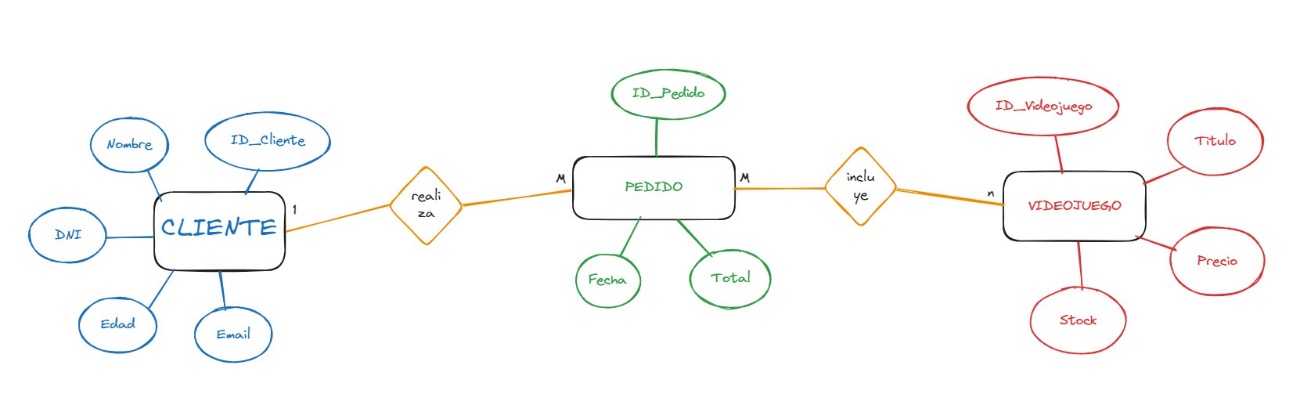
* **Clave concatenada CK:** ayuda a encontrar la singularidad en una tabla combinando dos campos, ya que no hay una llave primaria.
* **Claves candidatas:** cuando existe más de una clave primaria dentro de la tabla, por ejemplo, legajo y dni.

**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.

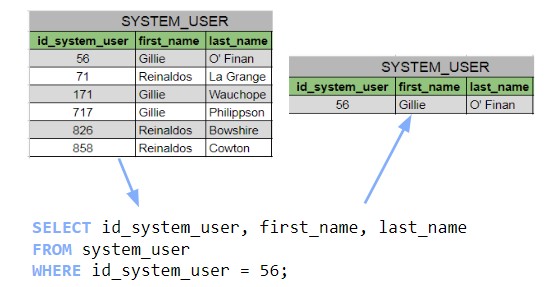
**Operadores de Comparación**

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**

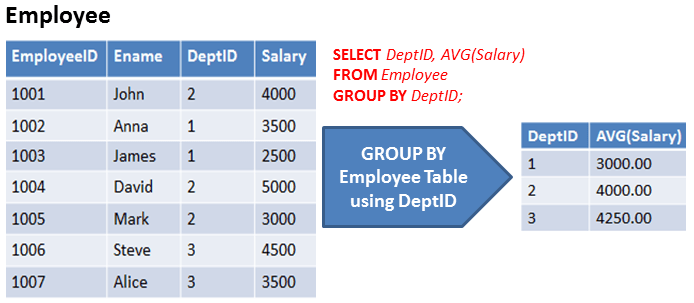
Así como SQL nos permite obtener datos de una o más tablas, también nos permite obtener valores simplificados o resumidos, sobre datos específicos que necesitemos. Esto se conoce como Funciones de Agregación o Agrupación. Podemos combinar funciones de totalización, conteo, promedios, valores mínimos 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()

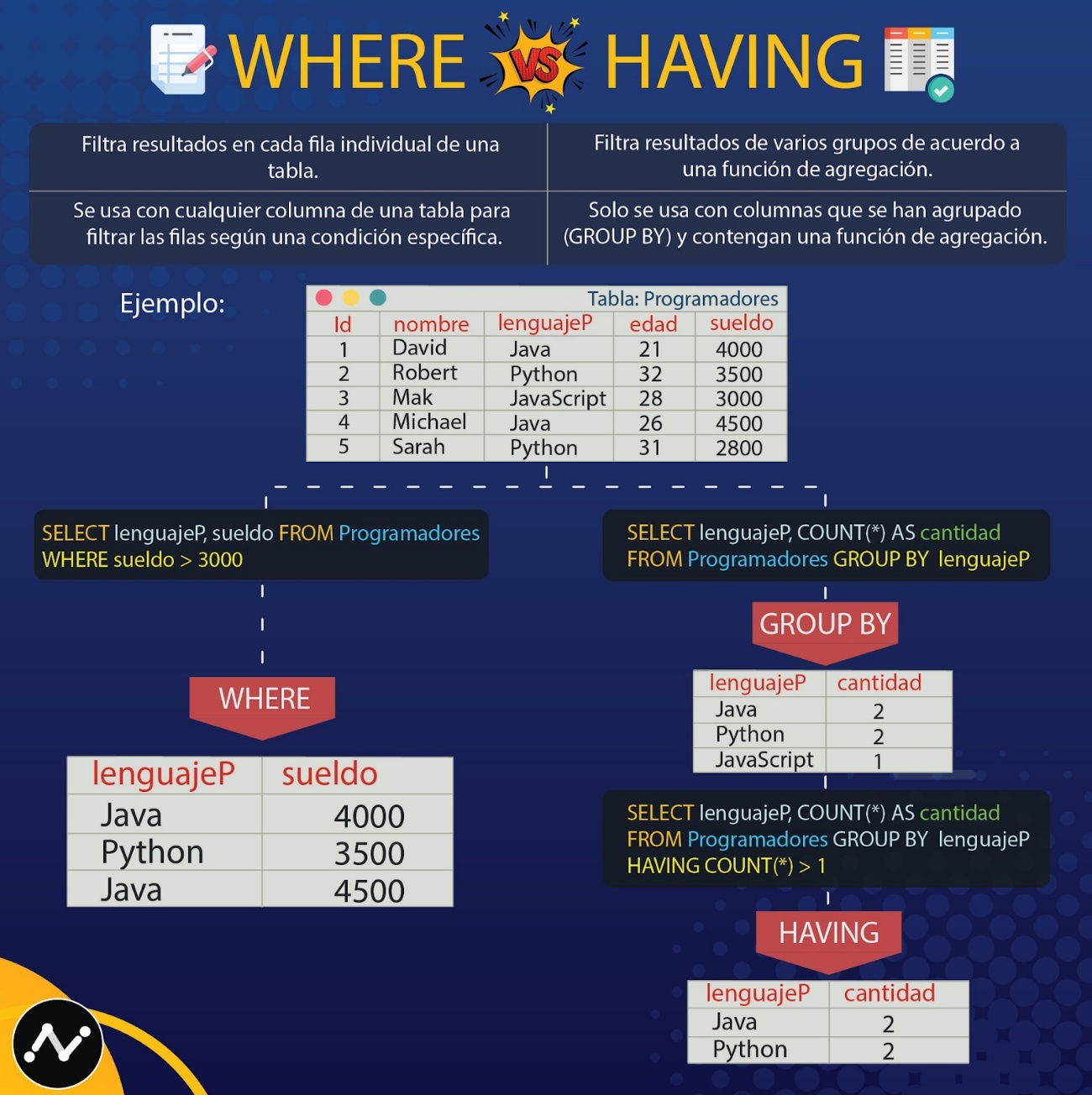
**Group By**

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().



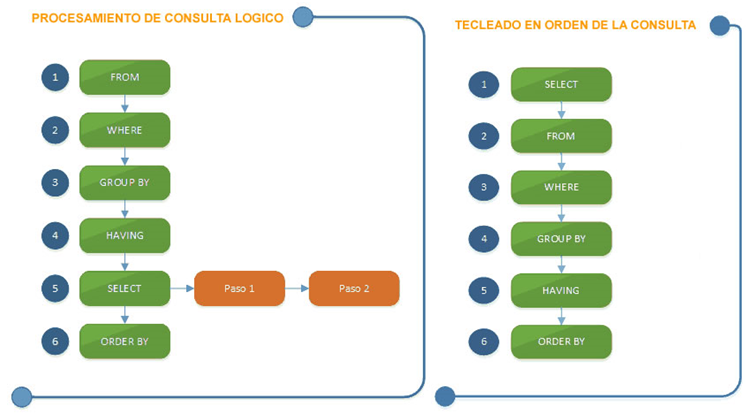
**HAVING**

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**

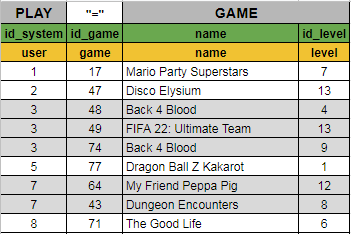
1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. LIMIT

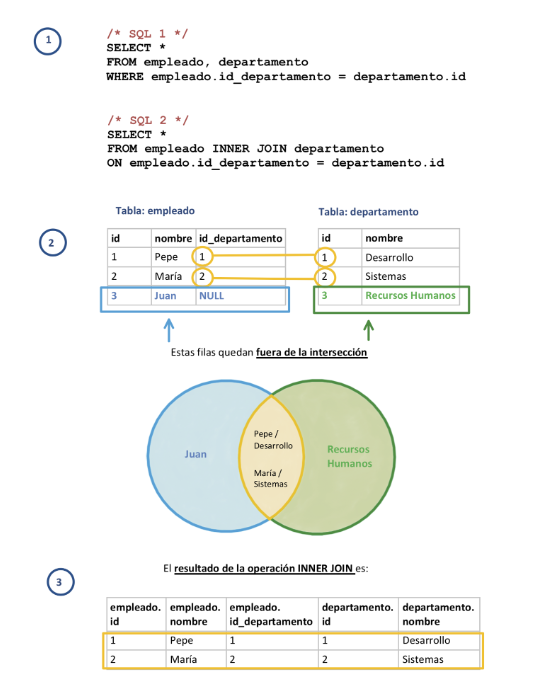
****

**JOIN**

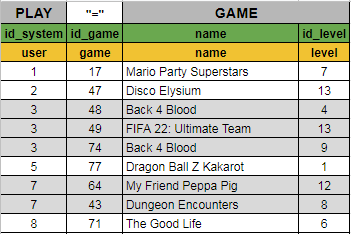
JOIN permite combinar registros de diferentes tablas, complementándose con la cláusula ON, la cual establece la condición por la cual queremos unir las tablas. Generalmente son campos comunes entre tablas. Los tipos de JOIN importantes, son cuatro: Inner Join, Left Join, Right Join, Full Join.

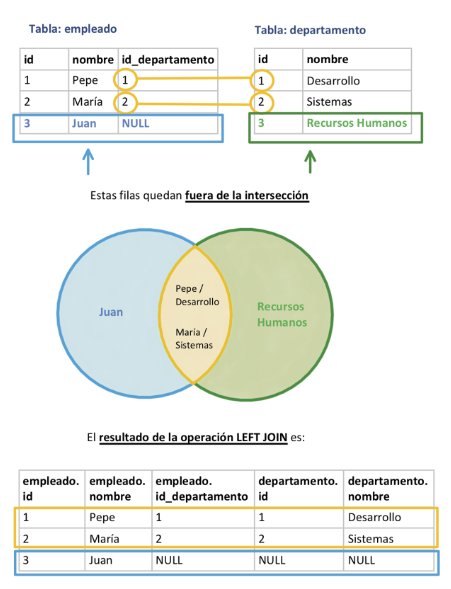
Inner Join: INNER JOIN, o JOIN, retorna todas las filas de las dos tablas siempre que haya coincidencia por el campo declarado en el ON. El resultado es NULL cuando no hay coincidencia alguna.



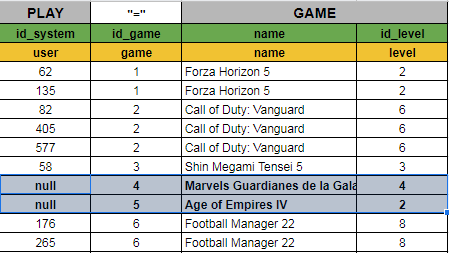


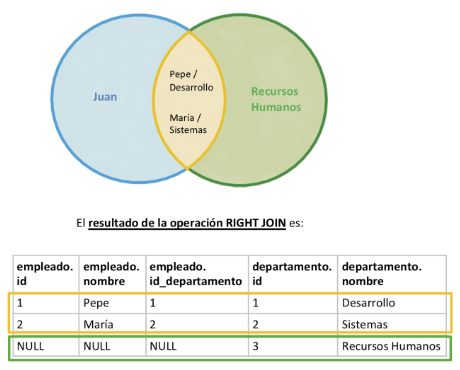
Left Join: LEFT JOIN retorna todas las filas de la tabla izquierda que coincidan con las filas de la tabla derecha. El resultado es NULL del lado derecho, cuando no hay coincidencia.



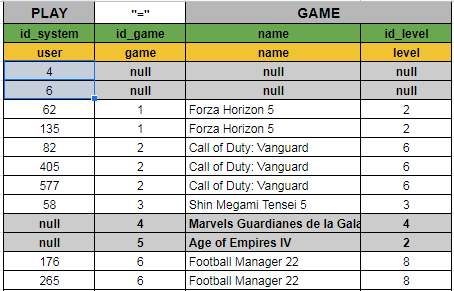


Right Join: RIGHT JOIN retorna todas las filas de la tabla derecha que coincidan con las filas de la tabla izquierda. El resultado es NULL cuando no hay coincidencia del lado izquierdo





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.

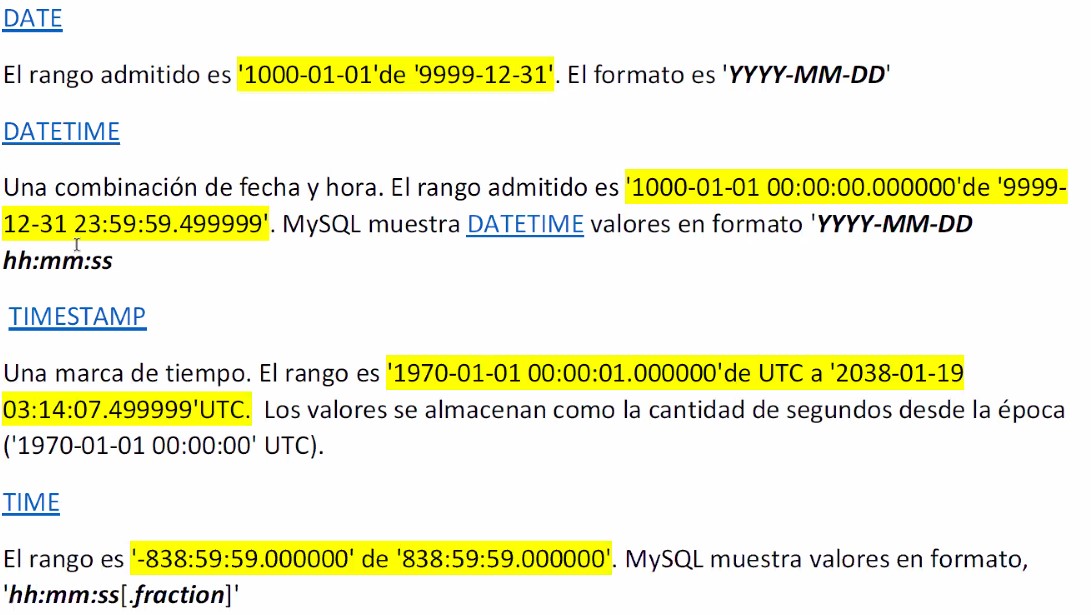
UNION ALL

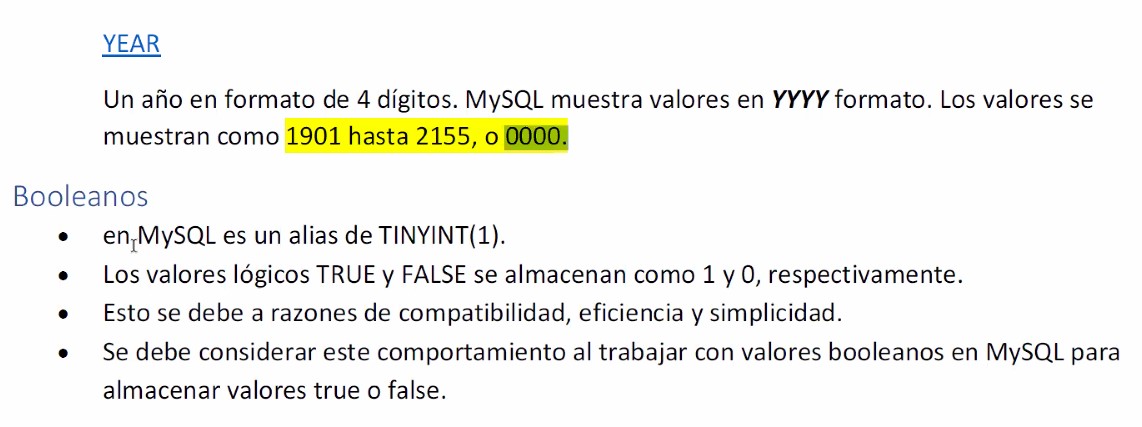
* **Incluye Duplicados:** El operador UNION ALL también combina los resultados de dos o más consultas SQL, pero a diferencia de UNION, no elimina los registros duplicados. Muestra todos los registros, incluidos los duplicados.
* **Uso de Recursos:** UNION ALL es más eficiente en términos de rendimiento 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*





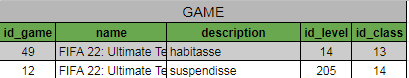
**LIKE ‘%’**

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%’;

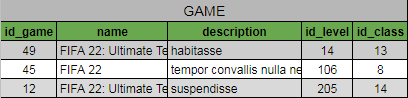


E implementando el caracter comodín “\_”, podemos también definir el desconocimiento de un solo caracter. Se puede combinar con el caracter %.

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:

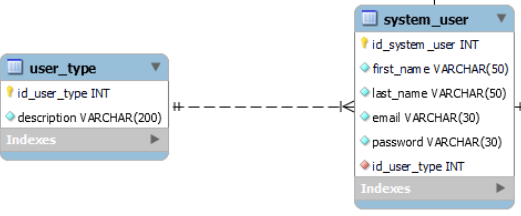
* La subconsulta debe ir entre paréntesis.
* La subconsulta debe tener una sola columna o expresión.
* No podemos utilizar BETWEEN o LIKE en la subconsulta.
* No debemos colocar la cláusula ORDER BY en la subconsulta.
* Otras cuestiones más con UPDATE y DELETE, que veremos oportunamente cuando abordemos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



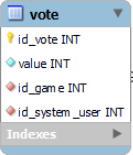
**Subconsultas en una misma tabla**

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

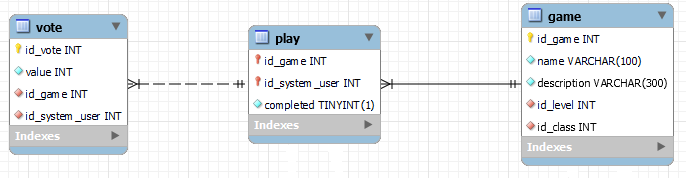
FROM vote WHERE value > (SELECT FLOOR(AVG(value)) FROM vote);



**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

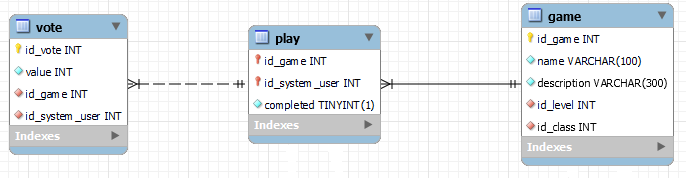
id\_game IN

(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:**

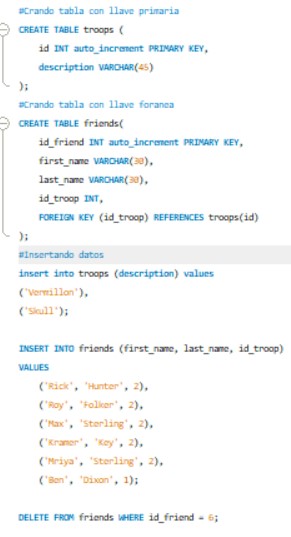
### **Objetos de una base de datos**

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

* Tablas
* Funciones
* Trigger
* Stored Procedures

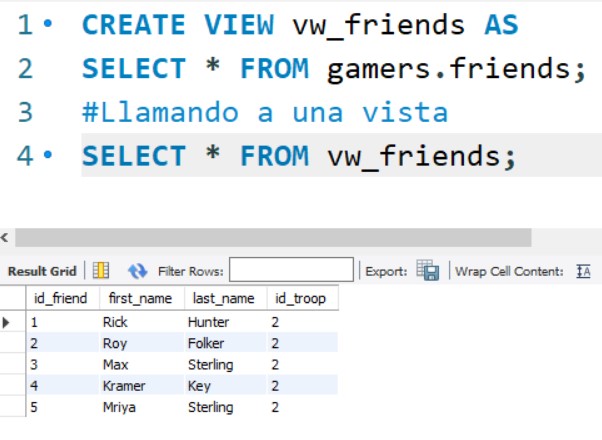
**Tablas (*entidad*):**

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.  


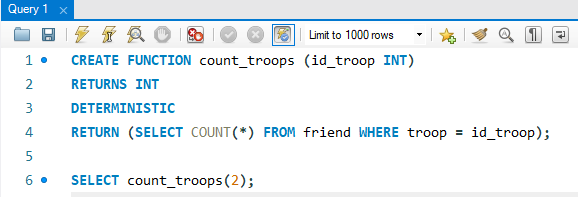
**FUNCIONES:**

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.

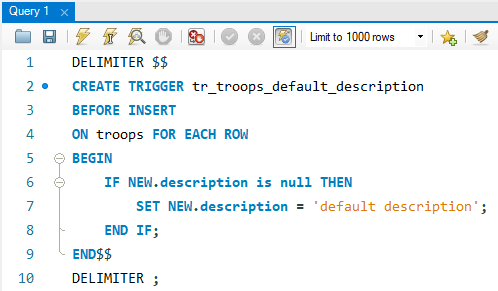
Si desarrollas en algún lenguaje de programación, encontrarás un parecido con las funciones personalizadas que creas en cualquier otro programa.  


**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:**

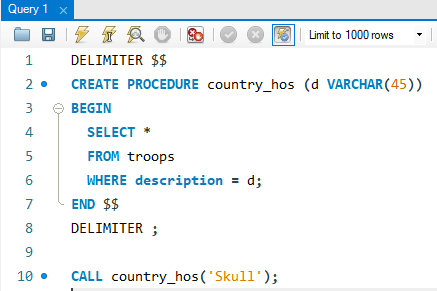
La ejecución de un Stored Procedure no está disponible para cualquier usuario. Es necesario que el perfil de éste, en el esquema de base de datos, tenga habilitado el permiso de ejecución (Execute). Un SP puede contener y ejecutar en su interior cualquier consulta del tipo DML. Incluso puede combinar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

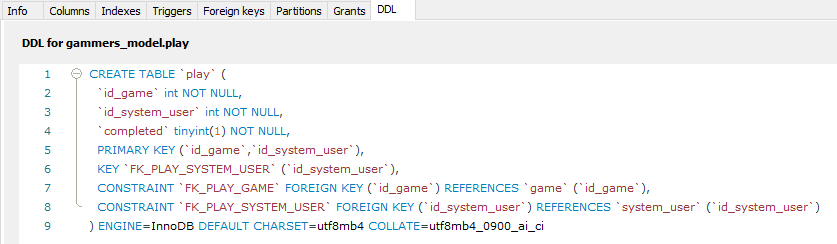
* Columns
* Indexes
* Foreign Keys
* Triggers

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]

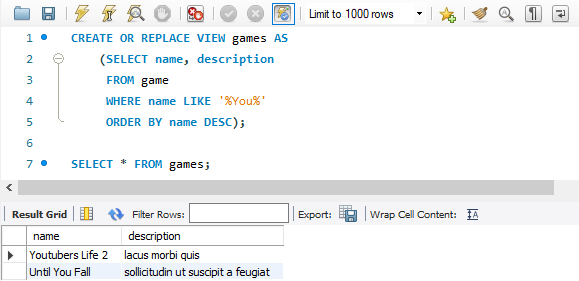
AS consulta\_sql

**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:**

### **Sublenguaje DML 1**

Comenzamos a profundizar aún más el uso del DML, **Data Manipulation Language**. En esta oportunidad conoceremos las otras sentencias SQL que nos permitirán **insertar, actualizar y eliminar** registros en las diferentes tablas de una base de datos.

**Componentes del DML**

* **Insert**: Se utiliza para insertar o agregar registros en una tabla.
* **Update**: Se utiliza para actualizar registros existentes en una tabla.
* **Delete**: Se utiliza para eliminar registros de una tabla.

**INSERT**

Se usa para agregar o insertar datos en una tabla. La información que agregamos puede ser:

* De forma individual (1 registro).
* De forma plural (varios registros a la vez).

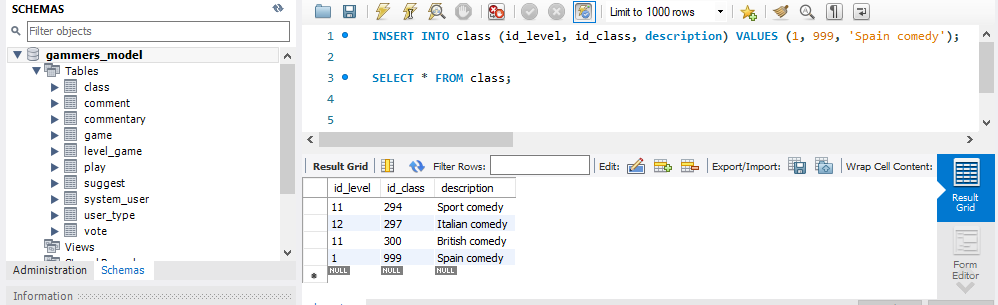
Su sintaxis se elabora mediante la cláusula INSERT INTO, seguida del nombre de la tabla, la cláusula VALUES, y los datos que se insertarán en la misma.

INSERT INTO nombre\_de\_la\_table (campo1, campo2, campo3, …)

VALUES (dato1, dato2, dato3, ...);

**Por ejemplo:**

INSERT INTO class (id\_level, id\_class, description) VALUES (1, 999, ‘Spain comedy’);



**Insert: Campo autoincrement**

En el caso de contar con campos AUTOINCREMENT, tenemos la posibilidad de permitir que el DBMS resuelva el valor correlativo para éste, pasando como parámetro el valor NULL o no colocándolo en la lista de campos. Tomemos como ejemplo la tabla PAY que creamos en la clase 6.

INSERT INTO pay VALUES (NULL, 250, ‘U$S’, ‘2021-07-22’, ‘Paypal’, 850, 77);

**UPDATE**

Su sintaxis se elabora mediante la cláusula UPDATE tabla, seguida la palabra SET y luego la(s) columna(s) o campo(s) con su(s) nuevo(s) valor(es).

UPDATE nombre\_de\_la\_tabla SET campo2 = ‘dato2’;

Para actualizar el valor de más de un campo, debemos separar cada uno de éstos con una coma. Especificamos campo = valor, otroCampo = otroValor, y así con cada campo que deseamos actualizar

UPDATE nombre\_de\_la\_tabla SET campo1 = ‘dato1’, campo2 = ‘dato2’, campo3 = ‘dato3’;

**Update: Condicionar la actualización**

Podemos limitar la actualización de datos que cumplan una determinada condición. Por ejemplo, cambiar la fecha de pago de aquellos realizados en el día. Para esto debemos integrar la cláusula WHERE, especificando la o las condiciones que deben cumplirse.

UPDATE pay  
SET date\_pay = CURRENT\_DATE - 1

WHERE date\_pay = CURRENT\_DATE;

**DELETE**

Finalmente, para eliminar registros de una tabla, debemos utilizar la sentencia DELETE. Se ocupa de eliminar todos los registros, o sólo aquellos que coincidan con determinados parámetros que le indiquemos en la condición del WHERE.

Su sintaxis se elabora mediante la cláusula DELETE FROM seguido del nombre de la tabla. Además, debemos agregar la cláusula WHERE, para indicarle cuál o cuáles registros se deben eliminar.

DELETE FROM nombre\_de\_la\_tabla WHERE (campo = ‘dato’);

**Delete: Posible errores**

Si intentamos eliminar registros de una tabla cuya PRIMARY KEY es FOREIGN KEY en otra u otras tablas, SQL no realizará la operación y nos advertirá dicho impedimento a través de la consola.

Por ejemplo, delete from level\_game where id\_level = 5;

Ante esta situación, debemos eliminar primero el registro asociado mediante la FOREIGN KEY en la otra tabla, para luego proceder con la eliminación de este registro.

**TRUNCATE TABLE**

Recordemos que para eliminar todos los registros de una tabla podemos utilizar TRUNCATE en lugar de DELETE sin where. Internamente, TRUNCATE borra todos los registros mientras que DELETE, recorre uno a uno y los va eliminando.

TRUNCATE nombre\_de\_la\_tabla;

### **Sublenguaje DML 2**

**2da-parte.sql**

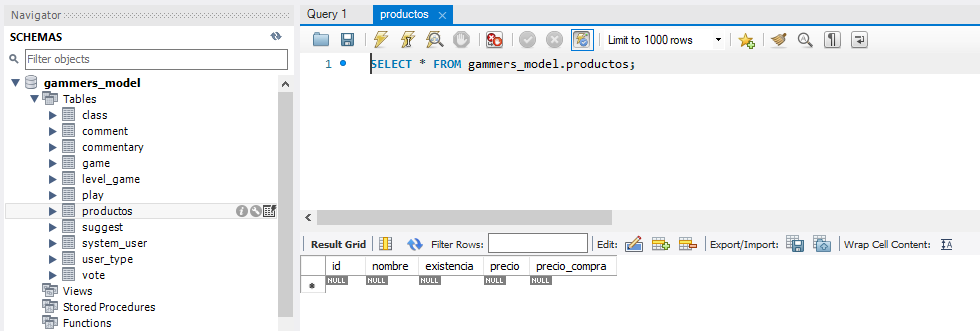
## **CLASE 7:**

### **Proceso de Importación de Datos**

Para llevar adelante el proceso de intercambio de información, Mysql nos provee un set de herramientas mediante la línea de comandos. Como también a través de Mysql Workbench, nos permiten trabajar con información (o registros), que pueden ser importados desde una fuente externa.

Herramientas del SGBD – Asistente de importación

Todos los SGDB cuentan con herramientas para importar y exportar contenido a sus tablas de datos. En Mysql, estas herramientas se pueden utilizar directamente desde MySQL Workbench. Podemos dar con esta característica abriendo la tabla (o consultando los datos de la tabla) y ubicando en su barra de herramientas el botón Export/Import.



Tipos de archivos para importar datos

Los formatos más populares para **importar**/**exportar** datos, son:

* **CSV**: Archivos separados por coma (Comma Separated Values)
* **TXT**: Archivos de texto convencionales
* **JSON**: Archivos JSON (JavaScript Object Notation)

**TXT** y **CSV** se utilizan prácticamente de la misma forma. Ambos se engloban bajo el formato “archivo de texto”, estructurando su contenido separado por una coma. Cada bloque de texto condice con un campo de la tabla de datos.

**JSON** es un formato de transporte muy utilizado en el mundo de la programación. Intercomunica diferentes tecnologías de software normalizando la información bajo un mismo “idioma”. El contenido de cada “registro”, se encierra entre las llaves { y }. Cada dato se separa del otro por una coma, y se utiliza el formato clave - valor. La clave equivale al nombre del campo y, el valor, al dato que se almacena en el mismo.

- Ejemplo en vivo (Diapositivas 36 a 41)

- Herramientas de consola – Importar desde consola (Diapositivas 44 a 60)

**Integridad Referencial**

La integridad referencial garantiza que, la clave externa de una tabla de referencia, coincida siempre con una fila válida que exista en otra tabla, asegurándose así que la relación entre éstas dos tablas continúe sincronizada durante las operaciones UPDATE y DELETE.

Existen tres tipos de integridad referencial:

* **Integridad referencial débil:** Si en una tupla de R, todos los valores de los atributos de K, poseen un valor que no es del tipo nulo, entonces debe existir una tupla en S que tome esos mismos valores en los atributos de J.
* **Integridad referencial parcial:** Si en una tupla de R, algún atributo de K cambia a valor nulo, entonces debe existir una tupla en S que tome en los atributos de J, los mismos valores que los atributos de K con valor nulo.
* **Integridad referencial completa:** En una tupla de R, todos los atributos de K deben tener el valor nulo, o bien, todos tienen un valor que no es el nulo por lo cual debe existir una tupla en S que tome en los atributos de J, los mismos valores que toman los de K.

**Restricciones de Integridad**

Las restricciones de integridad se especifican en el esquema de la base de datos, debiendo ésta cumplir en cualquier extensión del mismo. Esto hará que el Sistema de Gestión de Base de Datos vele en todo momento por la integridad de la información almacenada.

El modelo relacional contempla cuatro tipos de restricciones:

* Restricción de unicidad
* Restricción de valor no nulo
* Restricción de clave primaria
* Restricción de integridad referencial

**Establecer las condiciones de restricción**

Cuando definimos una tabla con sus respectivos índices, claves y relaciones, tenemos la posibilidad de especificar cuatro tipos de acciones diferentes sobre las tablas relacionadas y cómo afectará la información de éstas en el momento de eliminar o actualizar el dato clave de un registro.

Estas acciones a definir, son:

* CASCADE
* SET NULL
* NO ACTION
* RESTRICT

Y se aplican directamente sobre las cláusulas ON DELETE y ON UPDATE.

## **CLASE 8:**

### **Funciones**

Las funciones customizadas permiten procesar y manipular datos de forma procedural y eficiente. Dichos datos son enviados a través de uno o más parámetros, al momento de invocar la función y retornando un único resultado. Podemos crear funciones, a la medida de nuestra necesidad, combinando las mismas con funciones ya existentes del lenguaje Mysql, para así obtener los resultados deseados tal como necesitamos.

Para crear funciones personalizadas, necesitamos conocer el lenguaje de programación SQL, el cual nos permitirá definir una estructura básica que toda función personalizada debe cumplir para operar como tal.

**Beneficios**

* Mejoran la integridad y seguridad de los datos: Crear funciones en el servidor, garantiza que, los resultados de los datos procesados serán iguales para cualquier Stack de Programación.
* Optimizan el rendimiento de la base de datos: Las funciones customizadas envían los datos procesados por el servidor al cliente, minimizando el tráfico de información.
* Otorgan una mejor lectura del código: Puedes normalizar o personalizar su nombre, de acuerdo a las convenciones.

**Tipos de funciones**

* Funciones simples
* Funciones con parámetro(s)
* Funciones con parámetro(s) y retorno

**Sintaxis**

**CREATE FUNCTION `nombre\_de\_la\_funcion`**: Esta es la sentencia DDL que nos permite crear una función partiendo desde una ventana de Scripting. El nombre de la función va especificado entre comillas simples del tipo back tick o, acento grave.

**RETURNS tipoDeDato**: Las funciones devuelven o retornan usualmente algún tipo de dato. Éste puede ser boolean, char o number, entre otros, según lo que estamos procesando.

Junto a la sentencia CREATE FUNCTION debemos especificar qué tipo de datos retornará la misma.

CREATE FUNCTION `nombre\_de\_la\_funcion` RETURNS CHAR(60)

**Parámetros**

**(param1 INT, param2 INT)**: las funciones suelen recibir uno o más parámetros de entrada, los cuales provienen de las columnas de una tabla. Debemos especificar el nombre de cada parámetro, junto al tipo de dato de éste y, si son más de uno, debemos separarlos por una coma.

CREATE FUNCTION `nombre\_de\_la\_funcion` (param1 INT, param2 INT) RETURNS CHAR(60)

**Cuerpo de la función**

**BEGIN...END**: El cuerpo de la función se define a través de una estructura de inicio y fin. Y es allí, dentro de este cuerpo donde definiremos, a posteriori, el código que le dará vida a la misma.

CREATE FUNCTION `nombre\_de\_la\_funcion` (param1 INT, param2 INT) RETURNS CHAR(60)

DETERMINISTIC

BEGIN

…

END

**Definir una variable**

**DECLARE resultado\_id INT**: para manejar los diferentes valores dentro de una función, podemos definir una o más variables. Éstas se definen a través de la palabra reservada DECLARE seguido del nombre de la variable y el tipo de dato que manejará.

CREATE FUNCTION `nombre\_de\_la\_funcion` (param1 INT, param2 INT) RETURNS CHAR(60)

DETERMINISTIC

BEGIN

    DECLARE resultado INT;

…

END

**Establecer el valor de una variable**

**SET resultado\_id = 1 + 1**: cuando deseamos establecer un valor en una variable, debemos utilizar la palabra reservada SET seguida de la variable, el operador de definición, y el valor que se le asignará.

**Retornar un resultado**

**RETURN resultado\_id**: Finalmente, cuando la función realice la o las operaciones pertinentes, devolverá el resultado de las mismas a través de la palabra reservada RETURN seguido de la variable que almacena dicho resultado.

**EJEMPLOS EN VIVO 🡺 functions.sql & functions-2.sql**

### **Stored Procedures**

Un Stored Procedure o Procedimiento Almacenado representa un conjunto de sentencias almacenado físicamente en una DB, creado para cumplir tareas específicas. Permite también establecer niveles de seguridad y manipular operaciones complejas o extensas del lado del servidor, evitando un ida y vuelta de datos que termine sobrecargando una red o servidor.

Su estructura es similar a las Funciones SQL que vimos en la clase anterior pero, a diferencia de éstas, en un Stored Procedure su objetivo es resolver desde una operación simple hasta operaciones complejas que requieran modificar varias tablas y/o datos almacenados en una DB. El Lenguaje de programación SQL es el lenguaje usado para crear su lógica e integra también muchos comandos del tipo condicional, variables de entrada y de salida, potenciando así su poder de acción.

En el manejo de procesos complejos los S.P. se usan como una especie de API que ejecuta consultas, compara resultados, actualiza datos en algunas tablas y/o elimina datos de otras. Todo esto bajo un proceso controlado mediante una transacción, la cual puede deshacerse si algo falla.

**Beneficios**

* El motor de DB controla las operaciones.
* Se ejecuta en un servidor independiente.
* Devuelve al cliente el resultado final, evitando sobrecargar su computadora con procesos.
* Evita programar una lógica compleja del lado del cliente.
* Minimiza los errores concentrando las operaciones.

**Tipos de Stored Procedures**

* Procedimiento Almacenado básico.
* Procedimiento Almacenado con parámetro(s) de entrada.
* Procedimiento Almacenado con parámetro(s) de salida.
* Procedimiento Almacenado con parámetro(s) de entrada y salida.

**Sintaxis de un S.P.**

La sintaxis de un Stored Procedure se asimila en muchos puntos a lo que vimos en la primera parte de esta clase (Functions). Pero, a diferencia de las funciones, un S.P. puede o no tener parámetros de entrada, salida, y/o combinar ambos.

DELIMITER //

CREATE PROCEDURE `nombre\_del\_sp`

BEGIN

SELECT id, name FROM mi\_tabla;

/\* Algún comentario específico \*/

...

END //

**Parámetros de entrada**

Dentro de un S.P., podemos definir parámetros de entrada, los cuales recibirán valores cuando ejecutamos el S.P., de la misma forma que le enviamos parámetros a una Función SQL. Para definirlos, debemos utilizar la palabra reservada IN, y especificar el tipo de dato que soportan.

DELIMITER //

CREATE PROCEDURE `nombre\_del\_sp` (IN parametro1 CHAR(40))

BEGIN

SELECT \* FROM productos WHERE nombre LIKE parametro1;

...

END //

**Parámetros de salida**

En el S.P. puedes definir también parámetros de salida, los cuales funcionan como un curso, recibiendo valores directamente del código del S.P. Para definirlos, debemos utilizar la palabra reservada OUT, y especificando también el tipo de dato.

DELIMITER //

CREATE PROCEDURE `nombre\_del\_sp` (OUT total INTEGER)

BEGIN

SELECT COUNT(\*) INTO total FROM productos   
 WHERE habilitado = TRUE;

...

END //

**Uso de condicionales en un S.P.**

El lenguaje de programación SQL soporta el uso de condicionales, como cualquier otro lenguaje de programación.

CREATE PROCEDURE `nombre\_del\_sp` (IN condicion INTEGER)

BEGIN

If *condicion = 1* THEN

SELECT \* FROM...

END IF;

END

**IF ... THEN** es el condicional más utilizado y común a todos los lenguajes de programación. El código que se defina dentro de este bloque solo se ejecutará si se cumple la expresión que estamos evaluando (condicion = 1).

...

If *condicion = 1* THEN

SELECT \* FROM...

END IF;

...

En este caso, condicion es una variable y el valor que tenga asignado puede provenir como un parámetro IN del Stored Procedure o como el resultado de un cálculo o consulta interna, realizados dentro del S.P.

...

CREATE PROCEDURE `nombre\_del\_sp` (IN condicion INTEGER)

If *condicion = 1* THEN

SELECT \* FROM...

END IF;

...

**ELSE**: En este caso, condicion es una variable y el valor que tenga asignado puede provenir como un parámetro IN del Stored Procedure o como el resultado de un cálculo o consulta interna, realizados dentro del S.P.

...

CREATE PROCEDURE `nombre\_del\_sp` (IN condicion INTEGER)

If *condicion = 1* THEN

SELECT \* FROM tabla1...

ELSE

SELECT \* FROM tabla2...

END IF;

...

**Declara variables en un S.P. 🡺** SET @ordenamiento= ‘id’;

**Convertir strings SQL en clausulas**

Si utilizamos cadenas de strings y/o variables para armar una estructura del tipo DML con el nombre de un campo, un valor a buscar, y/o una consulta SQL, deberemos concatenar todo previo a ejecutar dicha instrucción SQL. Para que todo esto funcione, debemos integrar en el S.P. PREPARE, EXECUTE y DEALLOCATE.

PREPARE convierte una cláusula SQL a un elemento u objeto que el Motor SQL podrá interpretar mejor. Las cláusulas SQL escritas en una ventana de script o variable SQL no son más que una cadena de string. El motor SQL necesita un objeto SQL para entenderlo y ejecutarlo, y es allí donde PREPARE entra en acción.

La cláusula SQL debe almacenarse en una variable específica, declarada junto a la sentencia PREPARE, referenciando la misma con el origen de la cláusula SQL en formato string.

...

PREPARE ejecutar FROM @clausula;

...

...

**Ejecutar la cláusula**

Finalmente, con la cláusula preparada como “un objeto entendible por el motor SQL”, solo nos queda ejecutarla utilizando la sentencia EXECUTE.

Una vez ejecutada la sentencia SQL, podemos deshacer el objeto interpretable por el motor SQL, invocando la sentencia DEALLOCATE PREPARE.

...

PREPARE ejecutar FROM @clausula;

EXECUTE ejecutar;

DEALLOCATE PREPARE ejecutar;

...

**EJEMPLO EN VIVO 🡺 stored-procedures.sql**

## **CLASE 9:**

### **Triggers**

Definimos como Trigger a un conjunto de sentencias o programa almacenado en el servidor (de DB) creado para ejecutarse (dispararse) de forma automática, cuando uno o más eventos de DML específicos ocurren en la DB. El Trigger se despierta y ejecuta sus sentencias en el momento en que una operación de DML (INSERT, UPDATE y DELETE) asociada al disparador aparece.

**Usos específicos**

Los Triggers nacieron integrados a las bases de datos para funcionar de la misma forma en la cual muchos lenguajes de programación activan la detección de eventos específicos cuando el programa se ejecuta. Pero un Trigger no funciona de forma mágica. Nosotros somos quienes debemos definirlo y escribir su lógica.

La funcionalidad principal que más se les da a los Triggers en el ecosistema de bases de datos, es activarlos para alimentar las tablas de auditoría. Estas tablas funcionan como complemento dentro de una bb.dd., recopilando información adicional que no es importante dentro de las tablas principales

**Tablas de auditoría**

Tablas de Auditoría, Log, Bitácora, son algunos nombres con los cuales se definen a estas tablas secundarias, que se ocupan de almacenar información no importante para el negocio en sí, pero clave para el departamento de IT y/o de Seguridad Informática. Por ejemplo, una tabla Productos, almacena información del mismo como ser: código, descripción, fecha de alta o fabricación, precio de costo y precio de venta, entre otros.

Podemos registrar en una tabla de auditoría paralela, quién lo creó, fecha de creación, quién modificó su precio de venta o de costo, cuándo, y quién lo eliminó de la lista de Productos. Todo este registro o bitácora de cambios, lo podemos realizar activando uno o más Triggers que monitoreen todos estos pasos y cambios sobre uno o más registros.

Las tablas de LOGs o Bitácoras no suelen tener relación alguna con las entidades que monitorean y, muchas veces, almacenan información general de diferentes entidades.

**Tipos de *trigger***

El uso de Trigger se puede establecer en dos momentos diferentes de cuando se realiza una operación del tipo UPDATE, DELETE, o INSERT. Ese momento puede ser antes (BEFORE) de que ocurra la operación, o después (AFTER) de que ocurra la misma.

* Trigger **BEFORE**: Cuando el usuario envía una operación del tipo INSERT, UPDATE o DELETE sobre una tabla, y esta tiene activo el Trigger que detecta la operación, se disparará la acción BEFORE, la cual permitirá por ejemplo registrar en una tabla de auditoría que se realizará la operación xx sobre la tabla yy.
* Trigger **AFTER:** Cuando el usuario envía una operación del tipo INSERT, UPDATE o DELETE sobre una tabla, y esta tiene activo el Trigger que detecta la operación, se disparará la acción AFTER, la cual registrará en una tabla de auditoría que se realizará la operación xx sobre la tabla yy.

**Sintaxis**

Antes de crear un trigger, debemos tener previamente definidas:

* La tabla a la cual le asociaremos el trigger
* La tabla donde se realizarán operaciones relacionadas al Trigger

Resuelto estos dos puntos, queda definir si el Trigger se ejecutará antes o después de la acción a evaluar. Esto impactará en su sintaxis.

En el siguiente ejemplo, utilizamos la sentencia (CREATE TRIGGER) para definir una acción posterior al alta (AFTER INSERT) de un nuevo producto en la tabla homónima (ON productos).

CREATE TRIGGER `accion\_y\_nombre\_del\_trigger`  
AFTER INSERT ON `productos`

Nos quedan dos incógnitas a resolver para finalizar el armado del Trigger (manipular la inserción de múltiples registros e identificar los nuevos registros). Lo primero lo resolveremos con la sentencia FOR EACH ROW, mientras que, lo segundo, con el comando NEW.

**New**

La palabra reservada NEW se ocupará de detectar o ubicar a cada nuevo registro agregado. No la utilizamos para un registro (o fila) completo en sí, sino que debemos integrar con cada uno de los datos que conforma un nuevo registro. De esta forma garantizamos que se llevará una bitácora de cada uno de los nuevos registros que se inserten en la tabla principal.

CREATE TRIGGER `accion\_y\_nombre\_del\_trigger`  
AFTER INSERT ON `productos`

INSERT INTO `tabla\_auxiliar` (campo1, campo2...)

VALUES (NEW.campo1, NEW.campo2...)

**For each row**

Un Trigger es un evento que se dispara una vez por cada sentencia INSERT que se realice sobre la tabla asociada. Pero, como mencionamos anteriormente, para aquellos casos donde se inserten registros de forma masiva, necesitamos incluir la cláusula FOR EACH ROW.

CREATE TRIGGER `accion\_y\_nombre\_del\_trigger`  
AFTER INSERT ON `productos`

FOR EACH ROW

INSERT INTO `tabla\_auxiliar` (campo1, campo2...) (campo1, campo2...)

VALUES (NEW.campo1, NEW.campo2...)

**Integración de funciones 🡺 Funciones del sistema**

En casi todos los casos donde los Trigger se usan para registrar un LOG o Bitácora de cambios sobre diferentes tablas, debemos integrar datos adicionales a estas últimas, como ser el usuario que realiza el cambio, el ambiente de base de datos donde esto ocurre y/o la fecha y hora. Y para realizar esto de forma efectiva, Mysql cuenta con una serie de funciones las cuales nos facilitan el trabajo. Podemos dividir a las mismas en tres categorías diferentes:

* de Fecha y Hora
* de Usuario
* de Plataforma

**Funciones de Fecha y Hora**

* NOW()
* CURRENT\_DATE()
* CURDATE()
* CURRENT\_TIME()
* CURTIME()
* CURRENT\_TIMESTAMP()

**Funciones de Usuario**

* SESSION\_USER()
* SYSTEM\_USER()
* USER()

**Funciones de Plataforma**

* DATABASE()
* VERSION()

**Ejemplo en vivo: Diapositivas 45 a 51**

### **Workshop DML**

**guia-workshop2.sql**

## **CLASE 10:**

…