INGENIERÍA EN SISTEMAS

ASIGNATURA: BASE DE DATOS

CONTENIDISTA: Ing. Silvia Cobialca

UNIDAD 4:

OBTENCIÓN DE INFORMACIÓN

INDICE UNIDAD 4

MAPA DE LA UNIDAD 4

PROPÓSITOS

En esta unidad nos proponemos explicarle los principales comandos del lenguaje de consultas SQL como así tambien algunos conceptos más avanzados tales como las vistas, los stored procedures y los triggers. También veremos cómo se maneja la seguridad y se administran los permisos sobre los objetos de la base de datos.

OBJETIVOS

- ✓ Analizar las sentencias ddl: create, alter, drop.
- ✓ Analizar las sentencias dml: select, insert, update, delete.
- ✓ Entender cómo funcionan las subconsultas.
- ✓ Entender cómo funcionan las funciones agregadas.
- ✓ Entender cómo interactuar con nulls.
- ✓ Entender cómo funcionan las subconsultas.
- ✓ Aplicar las principales características avanzadas del lenguaje sql: vistas, stored procedures y triggers.
- ✓ Identificar los diferentes tipos de usuarios grupos y permisos.

CONTENIDOS

Para que alcance los objetivos, los contenidos que abordará son los siguientes:

1) Creación de sentencias SQL para obtener información de una base de datos
2) Creación de sentencias SQL utilizando funciones agregadas
3) Creación de vistas
4) Creación y utilización de stored procedures
5) Creación y utilización de triggers
6) Creación de usuarios y roles
7) Administración de permisos

PALABRAS CLAVES

Triggers, vistas, stored procedures, usuarios, roles, permisos, seguridad discrecional, copia de seguridad, recuperación

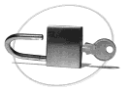
BIBLIOGRAFÍA de consulta

Elmasri, Ramez/Navathe, Shamkant (2011). *Fundamentos de Sistemas de Base de Datos.* 6ta Ed, EEUU: Pearson / Addison Wesley.

EVALUACIÓN

A lo largo de cada unidad, encontrará actividades y autoevaluaciones que le permitirán hacer un seguimiento de su aprendizaje.

INTRODUCCIÓN



ABORDEMOS EL LOGRO DEL PRIMER OBJETIVO DE LA UNIDAD:

- ✓ Analizar las sentencias DDL: create, alter, drop.

Para comenzar con esta unidad, y continuando desde la Unidad 3, donde ya estuvimos interactuando con la base de datos utilizando el lenguaje SQL, veremos cómo se crean estructuras de código más complejas. También aprenderemos a proteger la base de datos y a recuperarla de daños graves con un mínimo de pérdida de datos.

Comencemos…

1. Sentencias DDL: CREATE, ALTER y DROP

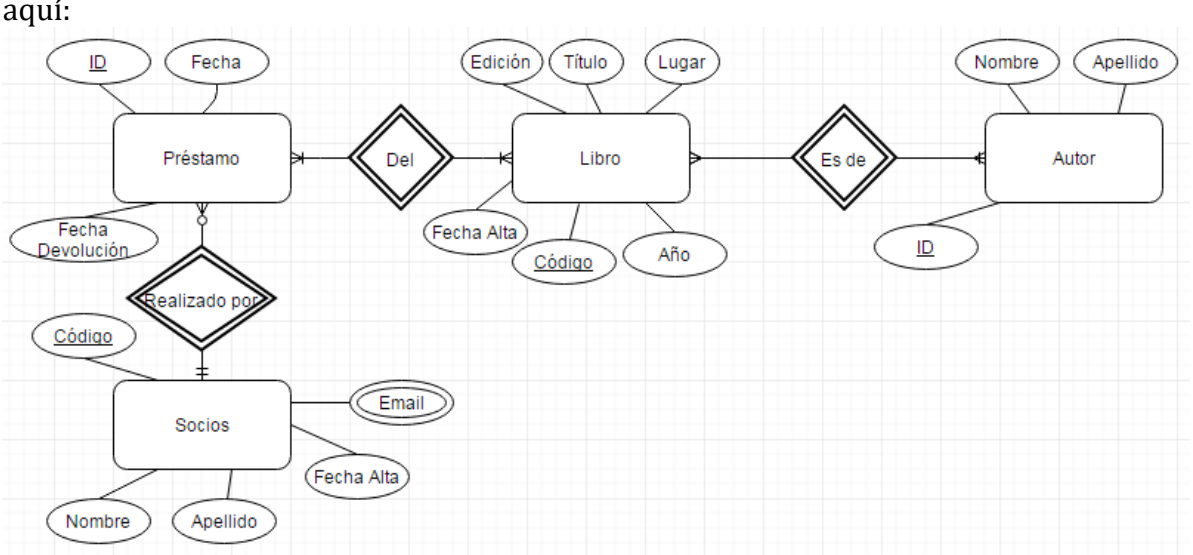
A partir de ahora comenzaremos a ver las distintas sentencias del lenguaje SQL (del inglés Structured Query Language) en su forma compatible con ANSI 92. ANSI es un standard del lenguaje, lo que nos asegura que dicha forma es compatible con todos los DBMS del mercado. Es decir que si escribimos una sentencia de código SQL ANSI 92 la podremos ejecutar exitosamente en cualquier DBMS. SI la sentencia no es ANSI, podría darse el caso que se ejecute con éxito en un DBMS (por ejemplo SQL Server) pero no en otros (por ejemplo Oracle y MySQL).
Como vimos en la Unidad 1, el lenguaje DDL o de definición de datos, contiene sentencias que permiten crear, modificar o eliminar objetos en el esquema interno de la base de datos en base al esquema conceptual. Para ello se utilizan tres sentencias muy potentes, ellas son CREATE, ALTER y DROP, las que veremos a continuación.

La sentencia CREATE tiene la siguiente estructura en su forma más simple:
CREATE <tipo de objeto> Nombre de objeto

Luego dependiendo del tipo de objeto la sentencia varía. Como nos vamos a concentrar por ahora en la creación de tablas, vamos a ver en detalle la sentencia para este caso:
CREATE TABLE <Nombre de la tabla> (
Columna1 tipo de dato NULL,
Columna2 tipo de dato NULL,
….
)
Los tipos de datos pueden ser:
Int: números enteros
Datetime: fecha y hora

Varchar (longitud en cantidad de caracteres): cadenas de caracteres o strings
Float: números decimales
Son los tipos más comunes que estaremos manejando.

Retomamos de la Unidad 2 el esquema interno que nos había quedado para el modelo de los préstamos de libros de la biblioteca, definido por el diagrama E-R que copiamos aquí:



Para crear la tabla Préstamo (cuidado que no vamos a poner el acento en la e para el nombre de la tabla):

```
CREATE TABLE prestamo (
ID int not null,
Fecha datetime null,
Fecha_Devolucion datetime null,
PRIMARY KEY (ID),
FOREIGN KEY (CodSocio) REFERENCES Socios(Codigo))
```

Hemos marcado con resaltadores de diferentes colores las palabras reservadas según si son tipos de datos: púrpura, sentencias SQL: celeste, otros: verde o gris para los NULL.
El NULL es el valor que tiene un atributo cuando no se le ha asignado aún ningún valor. Por ejemplo, al momento de crear una tabla, ésta no tiene ninguna t-upla. Luego los usuarios comienzan a agregar datos y puede suceder que no los conozcan todos, por ejemplo que al dar de alta un libro, no tengan a mano el año de impresión. Por lo tanto lo dejan vacío. El DBMS va a tomar este valor no ingresado como NULL.
Cuando creamos una tabla, y ponemos en las columnas NULL quiere decir que ese atributo no es requerido o sea que se podría dejar vacío cuando se ingresan datos. Si ponemos not NULL quiere decir que no permitiremos que esa columna quede en blanco, es decir que será obligatorio para los usuarios llenar ese campo y el DBMS se encargará de indicarle al usuario que debe ingresar un valor.
La cláusula Primary Key contiene entre paréntesis el nombre del atributo que será utilizado como clave de la tabla.

Por último, recordemos que teníamos la clave foránea que correspondía al código del socio que solicitaba el préstamo. Esa clave foránea se crea con la cláusula Foreign Key indicando en "References" el nombre de la tabla y el atributo relacionado.

De la misma manera, si queremos crear la tabla Socios, la sentencia será:

```
CREATE TABLE socios (
Codigo int not null,
Nombre varchar(30) null,
Apellido varchar(30) null,
Fecha_Alta datetime null,
PRIMARY KEY (Codigo))
```

Además tenemos el atributo multivaluado Email que habíamos visto que se pasaba como una tabla. Este se crea de la siguiente manera:

```
CREATE TABLE Email (
Codigo int not null,
Email varchar(100) null,
PRIMARY KEY (Codigo),
FOREIGN KEY (CodSocio) REFERENCES Socios (Codigo))
```

¿Podés poner a continuación la sentencia de creación de las tablas que faltan?

Si seguiste los pasos indicados entonces habrás creado las tablas restantes de la siguiente manera (recordarás de la Unidad 2 que las relaciones N-M se pasaban como una tabla adicional con los atributos clave de las tablas participantes en la relación, como clave de dicha tabla):

```
CREATE TABLE Libro (
Codigo int not null,
Título varchar(100) null,
Edición varchar(20) null,
Lugar varchar(50) null,
Año int null,
Fecha_Alta datetime null,
PRIMARY KEY (Codigo)
```

```
CREATE TABLE Autor (
ID int not null,
Nombre varchar(30) null,
Apellido varchar(30) null,
PRIMARY KEY (ID))
```

```
CREATE TABLE Libro_Autor (
CodLibro int not null,
CodAutor int not null,
PRIMARY KEY (CodLibro,CodAutor)
FOREIGN KEY (CodLibro)
REFERENCES Libro(Codigo),
FOREIGN KEY (CodAutor)
REFERENCES Autor(ID))
```

```
CREATE TABLE Prestamo_Libro (
CodLibro int not null,
CodPrestamo int not null,
PRIMARY KEY (CodLibro,CodPrestamo),
FOREIGN KEY (CodLibro)
REFERENCES Libro(Codigo),
FOREIGN KEY (CodPrestamo)
REFERENCES Prestamo(ID))
```

Ahora bien, te preguntarás ¿Qué sucede si quisiéramos cambiar algo en la tabla que creamos?, por ejemplo ¿Cómo haríamos si quisiéramos agregar una columna?

En ese caso utilizamos la sentencia ALTER. Para agregar una columna haríamos asi:

```
ALTER TABLE nombre de tabla
ADD nombre de columna tipo de dato;
```

O para agregar una columna haríamos asi:

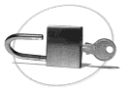
```
ALTER TABLE nombre de tabla
DROP COLUMN nombre de columna;
```

Para eliminar un objeto utilizaremos la sentencia DROP indicando el tipo de objeto y su nombre.
Por ejemplo, para eliminar la tabla socios haríamos lo siguiente:

```
DROP TABLE socios;
```

ACTIVIDAD 1

Realizar la Actividad 1 en el campus. Comente con sus compañeros en el foro de la unidad. Los ejercicios y las dudas serán revisados con el docente durante el encuentro virtual de la semana.

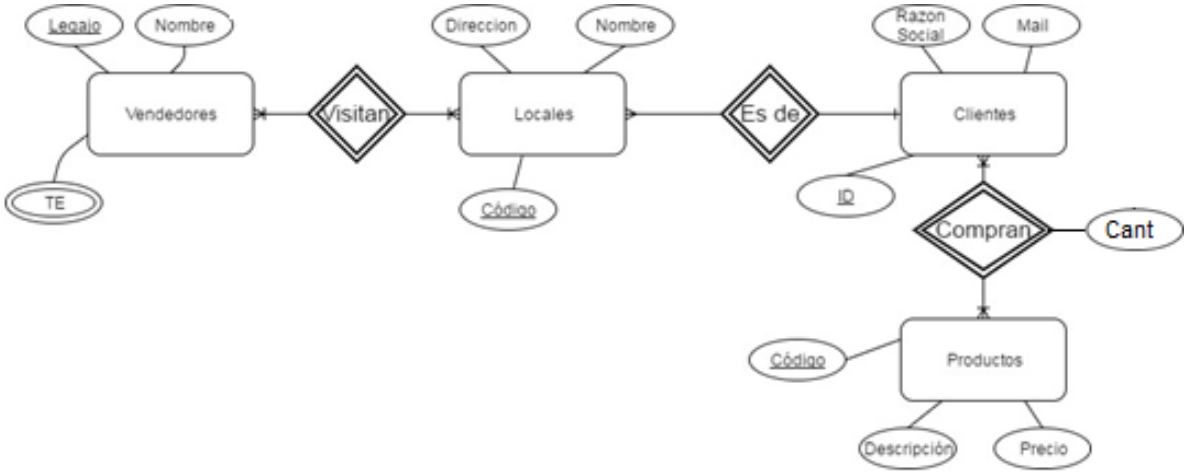


ABORDEMOS EL LOGRO DEL SEGUNDO OBJETIVO DE LA UNIDAD:

- ✓ Analizar las sentencias DML: select, insert, update, delete.

2. Uso de sentencias DML: select, insert, update, delete

A continuación vamos a aprender a leer o consultar las tablas que hemos creado suponiendo que ya están llenas de datos, por supuesto.
Tomaremos el diagrama E-R siguiente para interactuar con los objetos derivados del mismo e iremos trabajando en base a ejemplos prácticos para guiar el aprendizaje.



Para consultar los datos de una o varias tablas, se utiliza la sentencia SELECT. La forma más básica de esta sentencia es:
`SELECT` <lista de atributos> `FROM` <lista de tablas>
`WHERE` <lista de condiciones>
`ORDER BY` <lista de atributos> <asc, desc>

Si quisiéramos seleccionar todos los atributos, en lugar de listarlos de a uno separados por comas, podemos utilizar el * (asterisco) que simboliza justamente "todos los campos".
Por ejemplo utilizando esta forma del `SELECT` para hacer un listado de las descripciones y precios de los productos ordenados alfabéticamente hay que ejecutar la siguiente sentencia:

```
SELECT descripcion, precio FROM productos
ORDER BY descripcion;
```

Con esta consulta obtendríamos un listado como el siguiente:

Descripcion	Precio
Azúcar	$ 15
Harina	$ 10
Huevo	$1,70
Leche	$ 22

Si en lugar de querer todos los productos quisiéramos solamente la lista de aquellos cuyo precio es inferior a $ 20 y ordenados en forma descendente de acuerdo al precio (es decir los más caros al comienzo de la lista):

```
SELECT descripcion, precio FROM productos
WHERE precio<20
ORDER BY precio desc;
```

Obtendríamos este listado:

Descripcion	Precio
Azúcar	$ 15

Harina	$ 10
Huevo	$1,70

Vemos que el orden ascendente es el que se toma por defecto y por tanto si no se pone nada en la cláusula `ORDER BY` significa que es ascendente.

Quizás te preguntarás: ¿Cómo se hace si necesitamos obtener los nombres de los clientes y los productos que compraron, con sus cantidades?

Pues bien, se puede hacer de dos maneras, en la primera utilizaremos la cláusula `WHERE` para armar la correspondencia entre las t-uplas que participan en la relación (la unión entre la clave primaria de la tabla clientes y la clave primaria de la tabla productos). Para referirse a los atributos o campos de una tabla se utiliza la siguiente notación: tabla.atributo de manera que se puedan diferenciar los nombres de los atributos entre las tablas, sobre todo en los casos en que el nombre es el mismo (por ejemplo el código). Hacemos entonces:

```
SELECT Clientes.Razon_Social, Productos.descripcion, Productos.precio,
Clientes_Productos.cant FROM Productos, Clientes, Clientes_Productos
WHERE Productos.Codigo= Clientes_Productos.CodProducto and Clientes.Codigo=
Clientes_Productos.CodCliente;
```

Se obtendrá una lista como la siguiente:

Razón Social	Descripción	Precio	Cant
El Alfil	Harina	10	31
La Salamandra	Azúcar	15	11
La Salamandra	Huevo	1.70	12
El Alfil	Huevo	1,70	18
La Salamandra	Leche	22	13
El Alfil	Leche	22	15
La Salamandra	Harina	10	8
El Alfil	Azúcar	15	10

Si necesitamos mostrar todos los atributos de una tabla pero no de la otra ponemos tabla.*, por ejemplo cliente.* se usaría para mostrar todos los atributos de la tabla clientes. En cambio si utilizamos * sin especificar, se mostrarán todos los atributos de todas las tablas.

Se pueden usar alias para reemplazar el nombre de las tablas en la consulta y no tener que escribir tanto, por ejemplo podríamos usar c para Clientes, p para Preductos y cp para Clientes_Productos. Y quedaría entonces así:

```
SELECT c.Razon_Social, p.descripcion, p.precio, cp.cant FROM Productos p, Clientes c,
Clientes_Productos cp
WHERE p.Codigo= cp.CodProducto and c.Codigo= cp.CodCliente;
```

Si además quisiéramos calcular el monto gastado incluyendo el I.V.A. habría que hacer este cálculo: precio*cant*1,21 para cada t-upla, llamando a este cálculo con el nombre (o alias) monto, es decir se debería modificar la consulta de la siguiente manera:

```
SELECT c.Razon_Social, p.descripcion, p.precio, cp.cant, p.precio*cp.cant*1,21 monto
FROM Productos p, Clientes c, Clientes_Productos cp
WHERE p.Codigo= cp.CodProducto and c.Codigo= cp.CodCliente;
```

Obtenemos así el siguiente listado:

Razón Social	Descripción	Precio	Cant	monto
El Alfil	Harina	10	31	310
La Salamandra	Azúcar	15	11	165
La Salamandra	Huevo	1.70	12	19,40
El Alfil	Huevo	1,70	18	27,90
La Salamandra	Leche	22	13	286
El Alfil	Leche	22	15	330
La Salamandra	Harina	10	8	80
El Alfil	Azúcar	15	10	150

Habíamos dicho que había dos formas de hacer consultas que "cruzaban" varias tablas a través de sus relaciones. La primera que es la que vimos recién involucra el uso de la cláusula `WHERE` y la otra forma utiliza una nueva cláusula: el `JOIN`.
El `JOIN` se utiliza para indicar la manera en que se están relacionando las tablas, es decir, con que atributos se está plasmando la relación entre ellas. Se escribe de la siguiente forma:

```
SELECT <lista de atributos> FROM tabla1
JOIN tabla2 ON tabla1.campo1=tabla2.campo2
JOIN tabla3 ON tabla2.campo3=tabla3.campo4
```

Entonces, para escribir la misma consulta que antes pero utilizando el JOIN haríamos asi:

```
SELECT c.Razon_Social, p.descripcion, p.precio, cp.cant, p.precio*cp.cant*1,21 Monto
FROM Productos p
JOIN Clientes_Productos cp ON p.Codigo= cp.CodProducto
JOIN Clientes c ON c.Codigo= cp.CodCliente;
```

Observá que el `WHERE` no se utiliza a menos que sea para reales condiciones que correspondan a características propias de los datos, por ejemplo si deseamos que solo muestren los datos de los productos para los cuales el precio sea menor a $ 20:

```
SELECT c.Razon_Social, p.descripcion, p.precio, cp.cant, p.precio*cp.cant*1,21 Monto
FROM Productos p
JOIN Clientes_Productos cp ON p.Codigo= cp.CodProducto
JOIN Clientes c ON c.Codigo= cp.CodCliente
WHERE precio<20;
```

Cláusula IN:
Ahora veremos, cómo obtenemos los códigos de ciertos productos específicos, por ejemplo Harina, Azúcar y Leche:

SELECT código FROM producto WHERE descripción ='Harina' OR descripción ='Azúcar' OR descripción ='Leche'

Pero si tenemos una lista larga de posibilidades, escribir todas estas cláusulas OR encadenadas sería muy tedioso, entonces usamos la sentencia IN que funciona de manera equivalente:

SELECT código FROM producto WHERE descripción IN ('Harina' ,'Azúcar' ,'Leche')

Siempre después de la cláusula IN va una lista de una sola columna o atributo, y siempre antes va un atributo que debe coincidir en su tipo de dato con el tipo de dato de la lista.

Cláusula LIKE:
Otra cláusula muy potente del lenguaje SQL es la que se utiliza para comparaciones con campos de tipo de cadenas de texto. Esta sentencia se podría utilizar por ejemplo para consultar cuáles son los clientes que viven en una calle que contiene el nombre Martín, pero que no se sabe si se ha escrito Martín o Martin (o sea que podría estar sin acento). ¡O sea podríamos tener en esta lista gente que viva en la localidad de San Martí o en la localidad Martín Coronado o en Martínez!
La cláusula de la que estamos hablando es el LIKE. Veamos cómo se utiliza en este caso:

```
SELECT * FROM  Clientes c
WHERE  calle LIKE  '%Mart[ií]n%'
```

Analicemos un poco esta comparación para entender mejor como trabaja el LIKE:

Primero que nada:
Las comparaciones que trabajan con LIKE van todas entre comillas simples
Existen comodines como ser:
%: este comodín representa una cadena de cualquier largo que incluye texto, números y espacios en blanco
_: este comodín representa un solo carácter pero este puede ser una letra o un número
?: este comodín representa un solo carácter de tipo numérico (o sea un dígito)
[]: entre corchetes vamos a poder colocar todos aquellos caracteres o números posibles que pueden ir en un solo lugar. O sea es como si pusiéramos el guion bajo pero le diéramos solo una lista de opciones posibles refinando así los valores que se pueden poner en ese lugar.

Veamos entonces en detalle lo que pasó con nuestra consulta:
Al colocar el % al comienzo y al final estamos representando un texto que no nos preocupa como comienza ni cómo termina, siempre y cuando contenga la palabra que

nos interesa. Como no sabíamos si iba a estar escrito o no con acento entonces colocamos entre corchetes las dos opciones de i (o sea i e í).

Otro ejemplo:
Buscar los nombres de las calles que comiencen con N o J, luego viene una vocal y a continuación un texto cualquiera que termina con dos números. Esta condición sería así:

```
SELECT * FROM Clientes c
WHERE calle LIKE '[NJ][aeiou]%??'
```

ACTIVIDAD 2

Realizar los ejercicios 1 al 5 de la práctica 3 del campus. Los mismos serán revisados durante el encuentro virtual con el tutor de la semana.

ABORDEMOS EL LOGRO DEL TERCER OBJETIVO DE LA UNIDAD:

- ✓ Entender cómo funcionan las subconsultas.

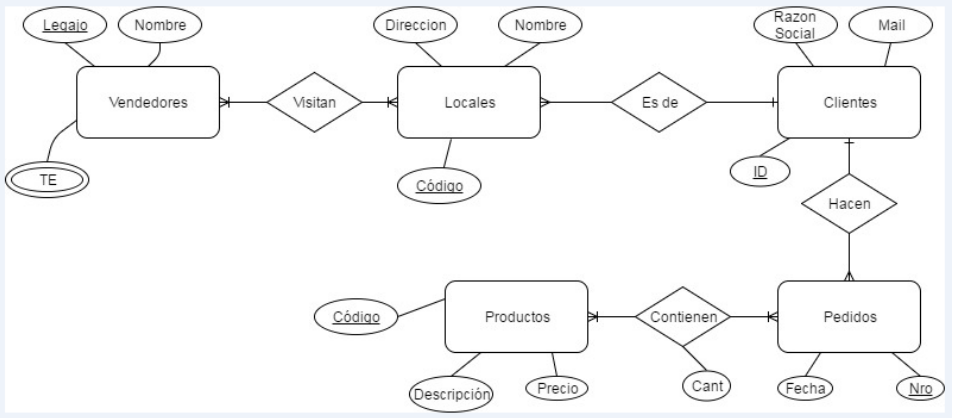
3. Uso de sentencias DML: Uso de subconsultas.

En este apartado avanzaremos un poco más sobre las consultas y veremos algunos trucos más avanzados de lenguaje SQL.
Por ejemplo, si quisiéramos saber cuáles son los clientes que NO compraron ningún producto en un cierto mes o en un rango de tiempo dado, tendríamos que seguir el siguiente razonamiento:
Primero deberíamos ver cuáles son los clientes que efectivamente compraron en el período dado, y a continuación deberíamos pedir los códigos de los clientes que no están en esa lista. Sencillo no?

Como nos iremos dando cuenta durante el estudio de esta unidad, en SQL siempre encontraremos varias formas de hacer consultar los datos y llegar al mismo resultado, por eso, en adelante te vamos a mostrar siempre más de una opción de escribir una consulta.

Veamos concretamente con un ejemplo:

Para el siguiente negocio cuyo diagrama E-R es:



suponiendo que queremos los clientes que no compraron productos en el mes de enero de 2017

Primero sacamos los clientes que compraron en enero de 2017:

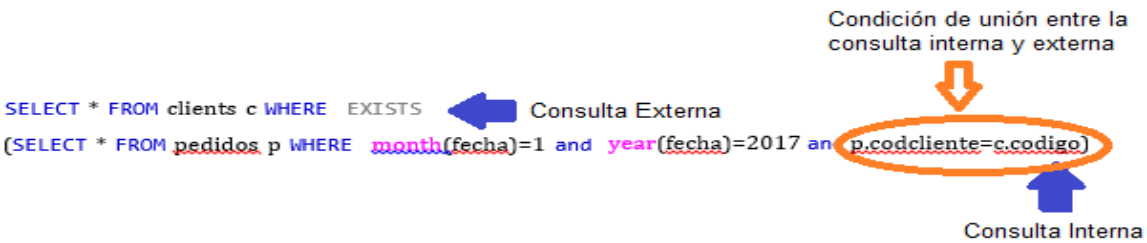
```
SELECT codcliente FROM pedidos WHERE month(fecha)=1 and year(fecha)=2017
```

Y ahora necesitariamos justamente los clientes que no son los de esta lista, por eso pedimos todos los datos de la tabla clientes siempre y cuando el código no esté en la lista de los que hicieron los pedidos del mes de enero de 2017:

```
SELECT * FROM clientes WHERE código not IN (SELECT codcliente FROM pedidos WHERE month(fecha)=1 and  year(fecha)=2017)
```

Cláusula EXISTS and NOT EXISTS:
Veamos otra manera de hacer esto mismo pero usando la cláusula `EXISTS`:
El `EXISTS` tiene una consulta interna y una externa, que están unidas por una condición `WHERE` de manera que, si la consulta interna devuelve datos para una determinada T-upla, entonces la consulta externa devuelve esa T-upla, y si la consulta interna no devuelve datos, entonces no se devuelve la T-upla.

Parece complicado pero veámoslo con el ejemplo anterior



Y ahora con los datos:
Si la tabla Pedidos y la tabla Clientes tienes las siguientes T-uplas:

Nro	Fecha	CodCliente
1	21/10/2016	14
2	13/11/2016	24
3	15/12/2016	17
4	3/1/2017	13
5	10/1/2017	22
6	28/1/2017	14
7	3/3/2017	24
8	9/4/2017	19

ID	Razon Social	Mail
14	La Salamandra	Salamandra@fuego.com
13	El Alfil	Alfil@alfil.com
17	Don Luis	DonLuis@donluis.com
22	Los Patitos	Info@patitos.com
24	Ramos Generales	rgrales@genstore.com
19	El Arca	info@elarca.com

Recordemos la consulta:

```
SELECT * FROM clients c WHERE EXISTS (SELECT * FROM pedidos p WHERE month(fecha)=1 and year(fecha)=2017 and p.codcliente=c.codigo)
```

Las T-uplas correspondientes al resultado de la consulta interna son:

ID	Fecha	CodCLiente
4	3/1/2017	13
5	10/1/2017	22
6	28/1/2017	14

Para estas T-uplas, la tabla de la relación Pedidos_Productos corresponde a:
La tabla Pedidos_Productos esta formada por las T-uplas:

CodPedido	CodProducto	Cant
4	2	15
4	3	25
5	2	21
5	4	33
5	5	29
6	1	13
6	2	12

Luego, si observamos la relación entre ambas consultas:
Lo que va a estar haciendo es tomar los clientes de la lista clientes que además estén en dicho conjunto de T-uplas, es decir el 13, el 22 y el 14.

Por lo tanto al final, como en realidad en la consulta externa estamos pidiendo todos los datos de los clientes que reúnan esa condición, lo que tendremos son las T-uplas de la tabla Clientes para los ID 13,22 y 14. Es decir:

ID	Razon Social	Mail
14	La Salamandra	Salamandra@fuego.com
13	El Alfil	Alfil@alfil.com
22	Los Patitos	Info@patitos.com

Para repasar todos los pasos seguidos utilizando el entorno visual MySQL Workbench, sería ahora un buen momento para que veas el video titulado "Subconsultas" que se encuentra en el campus.

ACTIVIDAD 3

Realice los ejercicios 6 al 15 de la práctica 3, trate de hacerlos de diferentes maneras. Comente con sus compañeros en el foro de la unidad las distintas opciones que encontraron para cada uno. Los ejercicios y las dudas serán revisados con el docente durante el encuentro virtual de la semana.

ABORDEMOS EL LOGRO DEL CUARTO OBJETIVO DE LA UNIDAD:

- ✓ Entender cómo se usan las funciones agregadas.

4. Uso de funciones agregadas

En este apartado vamos a ver una de las características que hacen del lenguaje SQL uno de los más usados y potentes cuando se trata de armar reportes. Aun hoy en los tiempos de Big Data, SQL sigue siendo popular por la funcionalidad asociada a la agregación de datos. Es decir, a la posibilidad de calcular sumas, promedios, entre otros, a partir de conjuntos de T-uplas agrupando por diferentes atributos que suelen denominarse **dimensiones**. Las funciones de agregación más comunes disponibles en el lenguaje, y aquellas en las que nos enfocaremos para los ejemplos, son: SUM, AVG, MAX, MIN, COUNT .

La sintaxis del uso de las funciones agregadas es como sigue:
`SELECT` <lista de campos>, función agregada `FROM` <tabla1 `JOIN` tabla2 `ON`....>
`GROUP BY` <lista de campos>
[`HAVING` función agregada <condición>]

Obviamente se utilizan conjuntamente con el `SELECT` ya que siempre van asociadas a una consulta. Además conceptualmente lo que hacen es reunir un conjunto de T-uplas de manera de juntar los datos para poder llevar a cabo la operación en cuestión (suma,

promedio, cuenta, etc), por lo tanto van a **agrupar** T-uplas, de ahí la necesidad de la cláusla `GROUP BY`.
Debés tener en cuenta que la <lista de campos> en el `GROUP BY` y en el `SELECT` **es la misma**.
Si no hay una lista de campos, quiere decir que vamos a obtener una suma total, por lo tanto la cláusula `GROUP BY` tampoco es necesaria.

Veamos ahora un ejemplo para entender cómo funciona la suma, y luego lo extenderemos al resto de las funciones agregadas citadas.
Si queremos saber cuánto se vendió en el mes de enero, o sea de las 3 ventas que vimos recién, quisiéramos calcular su suma total, multiplicando el precio por la cantidad y sumando obtendríamos un solo número $1936.5 calculado así:
15*10+25*1.7+21*10+33*22+29*17+13*15+12*10

Seguramente te preguntarás ahora ¿Cómo haríamos este mismo cálculo con el lenguaje SQL? Y además ¿Cuál es la potencia de SQL en este tipo de cálculo?

Pues bien, para realizar una consulta que calcule esto mismo usaremos la consulta interna del ejemplo anterior:

```
SELECT SUM(cant*pr.precio) FROM pedidos_productos pp
JOIN productos pr ON pp.codproducto=pr.codigo
JOIN  pedidos p ON p.nro=pp.codpedido
WHERE  month(p.fecha)=1 and  year(p.fecha)=2017
```

Observá que tenemos que usar la tabla pedidos para poder usar la condición del mes= enero y el año=2017 con la fecha, la tabla productos porque necesitamos los precios y la tabla pedidos_productos porque necesitamos la cantidad comprada de cada producto por cada pedido. Por eso necesitamos hacer esos tres join.
Además…
¡Claro que te habrás dado cuenta que la potencia está en que ahora podremos calcular una cuenta similar no solo para el mes de enero sino para todos los meses y podremos entonces saber la venta mensual, y a partir de allí la venta anual!

De manera similar, si quisiéramos saber cuántas unidades se vendieron de cada producto (no importa si son kg, litros o unidades), podríamos hacer así:
De manera similar, si quisiéramos saber cuántas unidades se vendieron de cada producto (no importa si son kg, litros o unidades), podríamos hacer así:

```
SELECT SUM(cant) FROM pedidos_productos pp
JOIN productos pr ON pp.codproducto=pr.codigo
JOIN  pedidos p ON p.nro=pp.codpedido
WHERE  month(p.fecha)=1 and  year(p.fecha)=2017
```

Y ahora, vamos a poner el mes para obtener la suma de las ventas de cada uno de los meses del año 2017, así veremos cómo trabaja el `GROUP BY` :

```
SELECT month(p.fecha) as Mes,  SUM(cant) as CantidadFROM pedidos_productos pp
```

```
JOIN productos pr ON pp.codproducto=pr.codigo
JOIN  pedidos p ON p.nro=pp.codpedido
WHERE  year(p.fecha)=2017
GROUP  BY  month(p.fecha)
```

Esta consulta nos retornará las siguientes T-uplas:

Mes	Catidad
1	148
3	174
4	130

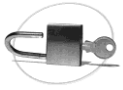
Si hubiéramos querido saber los pedidos cuyo total fuera superior a $ 1000 hubiéramos tenido que hacer lo siguiente:

```
SELECT p.Nro, SUM(cant*precio) as total FROM pedidos_productos pp
JOIN productos pr ON pp.codproducto=pr.codigo
JOIN  pedidos p ON p.nro=pp.codpedido
GROUP  BY p.nro
HAVING SUM(cant*precio)>1000
```

Antes de pasar a la siguiente actividad sería buena idea ver el video titulado “Funciones Agregadas” que encontrarás en el campus. Allí podrás ver algunos ejemplos adicionales de uso de funciones agregadas.

ACTIVIDAD 4

Realice los ejercicios de la práctica de la sección de funciones de agregación, piense si los puede hacer de más de una forma. Comente con sus compañeros en el foro de la unidad. Los ejercicios y las dudas serán revisados con el docente durante el encuentro virtual de la semana.



ABORDEMOS EL LOGRO DEL QUINTO OBJETIVO DE LA UNIDAD:

- ✓ Entender cómo interactuar con nulls.

5. Interacción con NULLs

Es posible que al ingresar los datos de una tabla de la base de datos, no conozcamos el valor de alguno de sus atributos, entonces lo que debemos hacer es no ingresar el valor correspondiente. Por ejemplo si tenemos que insertar un nuevo cliente y no conocemos su email deberíamos hacer lo siguiente:

```
INSERT INTO clientes (ID, razon_social)
VALUES (32,'El dulce cañon')
```

Y NO lo siguiente:

```
INSERT INTO clientes
VALUES (32,'El dulce cañon', '')
```

O

```
INSERT INTO clientes (ID, razon_social,email)
VALUES (32,'El dulce cañon', '')
```

Podrás ver queen el caso correcto (el primer caso), no se está listando el atributo que se desconoce en la lista de campos del insert y por lo tanto no se coloca en valor en la cláusula `VALUES`.

Te preguntarás entonces qué sucede con el valor del email cuando se ingresa el registro en la tabla, ¿Qué valor queda ingresado?

¡Justamente es de lo que estamos hablando!

El valor ingresado en ese atributo para este caso es un valor especial denominado NULL. Lo habíamos utilizado cuando definimos la sentencia CREATE TABLE y dijimos que se utilizaba NOT NULL cuando se trataba de atributos para los que no era obligatorio colocar el valor en la tabla.

¡Pues de eso se trata!

Como no sabemos qué valor poner, no ponemos nada. Eso se traduce dentro de la base de datos en un valor NULL.
El NULL y el blanco no son lo mismo, el blanco es una cadena de caracteres vacía, pero es un valor de todas maneras.

¿Pero cuál es la diferencia entre la cadena vacía y un NULL? ¿No son lo mismo?

No son lo mismo para el DBMS, porque el NULL significa que el atributo en cuestión es desconocido. El NULL no es un valor, sino un estado del atributo. El DBMS maneja los atributos cuyo estado es NULL de una manera óptima, los puede clasificar y encontrar

más rápido y además, para nosotros que interactuamos con la base de datos es mucho mejor para encontrar esos datos y poder manejarlos posteriormente.

Es siempre mejor usar NULLs cuando desconocemos el valor a ingresar que ingresar cualquier otra cosa, o 'Valor Desconocido' o incluso blancos.
¿Por qué? ¿Si es un número, que valor ingresaríamos para simbolizar el valor desconocido? ¿0? ¡No es lo mismo un precio=0 que un precio desconocido!
¡Otra muy buena razón es que el 0 ocupa lugar y el NULL no!

Ahora seguro querrás saber ¿Cómo encontramos estos NULLs si los queremos buscar?

¡Lo que haremos es simplemente consultar por ellos! Y se hace de la siguiente manera:

```
SELECT * FROM tabla
WHERE  campo IS NULL
```

No se debe usar el = ya que como dijimos anteriormente, el NULL es un valor desconocido y como tal, no es igual a nada. De hecho las dos siguiente condiciones resultan falsas:
NULL=NULL
NULL<>NULL
Por ejemplo, si queremos saber cuales son los productos cuya descripción es desconocida deberíamos hacer la siguiente consulta:

```
SELECT * FROM productos
WHERE descripcion IS NULL
```

Y no

```
SELECT * FROM productos
WHERE descripcion = NULL
```

De la misma manera sse debe preguntar si el precio es desconocido:

```
SELECT * FROM productos
WHERE precio IS NULL
```

Si querés ver algun ejemplo de estos casos y su comportamiento ante los NULL podrías mirar el video titulado "Uso de NULLs" en el campus como para terminar de entender bien el tema.

ACTIVIDAD 5

Realice los ejercicios de la práctica 3 de la sección de consultas utilizando valores nulos. Comente con sus compañeros en el foro de la unidad. Los ejercicios y las dudas serán revisados con el docente durante el encuentro virtual de la semana.
Lea el artículo "NULLS nothing to worry about them" y conteste las preguntas de la tarea relacionada.

ABORDEMOS EL LOGRO DEL SEXTO OBJETIVO DE LA UNIDAD:

- ✓ Entender cómo funcionan y para que sirven las vistas.

6. Creación de Vistas

En este apartado veremos formas en que podemos guardar dentro del motor de base de datos porciones de código que sabemos vamos a necesitar ejecutar repetidamente en el futuro.

Recordarás que en la Unidad 3 habíamos visto consultas y subconsultas que nos servían para obtener información de nuestra base de datos.

Existen casos en que hay consultas que ejecutamos de manera rutinaria, quizás algunas de ellas las necesitamos todos los fines de mes porque tenemos que generar información para reportar a nuestra gerencia.

Otras podrían ser necesarias diariamente y tal vez hay casos de consultas que solo se necesiten a fin de año.

Para todos esos casos mencionados, y otros, es que se necesitan las vistas.

Luego te preguntarás ¿Y cómo las creo? Y cuando ya las tengo creadas ¿Cómo las uso?

Y además ¿Dónde están guardadas? ¿Cómo las encuentro?

Vamos a ir respondiendo a todas estas preguntas a partir de los siguientes pasos:

Primeramente crearemos una vista de la siguiente forma:

```
CREATE VIEW <Nombre de la vista>
AS
SELECT <lista de campos> FROM <lista de tablas>
WHERE <lista de condiciones>
```

Este código SQL se guardará en el catálogo de la base de datos bajo la categoría objetos, y dentro de ella, las vistas. Para poder usar luego este código, lo que se debe hacer es simplemente un select, es decir:

```
SELECT * FROM <Nombre de la vista>
```

Y éste devolverá el resultado correspondiente al código escrito dentro de la definición de la vista.

Veamos un ejemplo:

Podemos crear una vista para obtener las ventas por mes, como vimos en el video de funciones agregadas de la Unidad 3:

```
CREATE VIEW ventas_por_mes
AS
SELECT month(fecha) as Mes, year(fecha) as Año, sum(precio*cant) as Venta
FROM  pedidos_productos pp JOIN pedidos pe ON  pe.nro=pp.codpedido
JOIN productos p ON p.codigo=pp.codproducto
GROUP BY month(fecha), year(fecha)
```

Ahora para ejecutarla simplemente hacemos:

```
SELECT * FROM ventas_por_mes
```

Cuando ejecutamos esta consulta, el DBMS busca la definición del objeto ventas_por_mes y además, como la tiene guardada en el catálogo ya precompilada, la ejecución es más rápida.

ABORDEMOS EL LOGRO DEL SÉPTIMO OBJETIVO DE LA UNIDAD:

- ✓ Aplicar las principales características avanzadas del lenguaje sql: vistas, stored procedures y triggers.

7. Creación de stored procedures

Ahora ya podemos pasar a conceptos más avanzados del lenguaje SQL y tal como aprendiste en el apartado anterior, se pueden almacenar los códigos de las consultas que vas a ejecutar con frecuencia en las vistas.

Pero ¿y si lo que queremos ejecutar es un poco más complejo que solo una consulta?

Para eso se utilizan los procedimientos almacenados! (en inglés: Stored Procedures)

Estos son porciones de código SQL ya precompilados guardados en el catálogo (al igual que las vistas) que sirven para ejecutar varias tareas proporcionando mucha más versatilidad que las vistas. En general se utilizan para automatizar tareas rutinarias.

La estructura más simple del código de un stored procedure es la siguiente:

```
DELIMITER//
CREATE PROCEDURE <Nombre del procedimiento>()
BEGIN
<lista de sentencias>;
.....
END//
```

La sentencia DELIMITER, no es parte del stored procedure pero con esta sentencia lo que haremos es permitir el uso del carácter de fin de sentencia dentro del código del store procedure, indicando que se utiliza el delimitador para indicar el inicio y fin del código perteneciente al procedimiento.

Al finalizar el código, volvemos al carácter habitual, colocando a continuación de la END el delimitador elegido.

Los paréntesis indican que este procedimiento no tiene parámetros, en los casos de procedimientos con parámetros, la declaración de los mismos va dentro de los paréntesis.

Para llamarlo se hace lo siguiente:

Call <Nombre del procedimiento>()

Por ejemplo, ¿ qué pasa si lo que queremos es que se ajusten los valores de los stocks de los productos al final del día restando al stock exitente la venta del día?

Como primer paso modifiquemos la estructura de la tabla productos y agreguémosle una columna nueva llamada Stock:

```
ALTER TABLE productos
ADD Stock int;
```

Ahora si corremos:
```
desc productos
```

Este sería el resultado:
```
Codigo int,
Descripcion varchar(40),
Precio float,
Stock int
```

Ya estamos listos entonces para crear nuestro primer stored procedure. Lo que vamos a hacer antes de crearlo es pensar los pasos que deberíamos seguir cada día para actualizar el stock de cada producto:

Primero tenemos que conseguir las ventas del día de todos los productos.

Luego, se debe hacer un update de la tabla productos seteando el stock como el valor actual menos la venta, para cada producto.

Por último, sería ideal obtener la lista con el stock actualizado de los productos vendidos hoy, ordenados de menor a mayor

Pues entonces, ¡manos a la obra!

Primeramente vamos a hacer uso del nuevo conocimiento adquirido en el apartado anterior para crear una vista que nos sirva para sacar la venta de las unidades de los productos vendidos en el día:

NOTA: utilizaremos la función `CURDATE()` que nos devuelve el día de hoy.

```
CREATE VIEW venta_diaria
AS
SELECT p.codigo, sum(cant) as Unidades_Vendidas
FROM pedidos_productos pp JOIN pedidos pe ON pe.nro=pp.codpedido
WHERE fecha= CURDATE()
GROUP BY p.codigo

DELIMITER //
CREATE PROCEDURE Actualizar_Stocks()
BEGIN
UPDATE productos JOIN venta_diaria vd ON productos.codigo=vd.codigo
SET stock=stock-vd.unidades_vendidas;

SELECT * FROM productos
ORDER BY stock asc;

END //
DELIMITER ;
```

Ahora, todas las tardes se ejecutará este procedimiento que nos asegurará tener siempre actualizados nuestros stocks a la vez que listará los productos.

Quizás te preguntarás ¿Pero qué pasaría si se ejecutara dos veces seguidas el mismo día este procedimiento?

Claramente el stock se actualizaría otra vez, por supuesto erróneamente.

Pero es fácil resolver este inconveniente. Podríamos poner una columna adicional de tipo fecha en la tabla productos para guardar la fecha de actualización. Luego al realizar el update en el procedimiento, podemos verificar que la fecha de actualización no sea la actual, y listo!

Vamos a hacer estas modificaciones a la tabla y el procedimiento…

Entonces, agregamos la columna fecha_mod a la tabla productos:

```
ALTER TABLE productos
ADD FechaMod DATE;
```

Además, como recordarás de la Unidad 3, si no le asignamos ningún valor a esta fecha, su valor será NULL, por lo tanto cuando realicemos comparaciones con ella, el resultado será siempre FALSO ya que su valor es desconocido. Por eso, lo que podemos hacer es darle el valor de la fecha actual utilizando la función CURDATE() que usamos antes.

Para modificar el procedimiento, lo debemos borrar primero (usando la sentencia DDL DROP) y luego volver a crear, entonces procedemos así:

```
DROP PROCEDURE Actualizar_Stocks();

DELIMITER //
CREATE PROCEDURE Actualizar_Stocks()
BEGIN
UPDATE productos JOIN venta_diaria vd ON productos.codigo=vd.codigo
SET stock=stock-vd.unidades_vendidas,
FechaMod= CURDATE()
WHERE FechaMod<> CURDATE() ;

SELECT * FROM productos
ORDER BY stock asc;

END //
DELIMITER ;
```

Ahora bien, ya hicimos nuestro primer procedimiento y logramos modificar su funcionalidad. Pero podríamos tambien necesitar crear un procedimiento que realice ciertas tareas dependiendo de algunos parámetros. Dichos parámetros pueden ser de entrada, de salida o de entrada/salida.

Un ejemplo muy sencillo sería, por ejemplo, obtener un listado de ventas (parecido al de la vista de ventas por mes) para un mes y año específico, que serían los parámetros que utilizaría el procedimiento.

Le pondremos de nombre ventas_para_un_mes, veamos entonces:

```
DELIMITER //
CREATE PROCEDURE ventas_para_un_mes (IN mes int,IN anio int)
BEGIN
SELECT sum(precio*cant) as Venta
FROM pedidos_productos pp JOIN pedidos pe ON pe.nro=pp.codpedido
```

```
JOIN productos p ON p.codigo=pp.codproducto
WHERE month(fecha)=mes AND year(fecha) =anio;

END //
DELIMITER ;
```

Luego, para obtener el resultado se llama al procedimiento de la siguiente forma:

```
CALL ventas_para_un_mes (1,2017);
```

Pero el resultado lo obtendremos en la ventana de resultados. Si quisiéramos, podríamos guardar el resultado en un parámetro de salida del procedimiento. Modifiquemos entonces el código para hacer eso:

```
DELIMITER //
CREATE PROCEDURE ventas_para_un_mes (IN mes int,IN anio int, OUT Venta)
BEGIN
SELECT sum(precio*cant) INTO Venta
FROM pedidos_productos pp JOIN pedidos pe ON pe.nro=pp.codpedido
JOIN productos p ON p.codigo=pp.codproducto
WHERE month(fecha)=mes AND year(fecha) =anio;

END //
DELIMITER ;
```

Y para llamar al procedimiento haremos lo siguiente:

```
CALL ventas_para_un_mes (1,2017,@monto);
SELECT @monto;
```

En la primera línea hemos llamado al procedimiento, indicando que el resultado se cargue en la variable @monto, y en la segunda línea hemos hecho un select de dicha variable para obtener su valor.

8. Creación de triggers

Como vimos en el apartado anterior, los procedimientos almacenados son porciones de código que se utilizan para realizar tareas rutinarias que los usuarios van a querer correr frecuentemente.

Existen otros escenarios en los que quisiéramos que se corra un código SQL pero, ante la ocurrencia de un evento particular. Por ejemplo, cada vez que se realice una venta, quisiéramos que se actualice el stock de los productos relacionados. Algo similar a lo que habiamos hecho antes, pero en lugar de hacer la actualización del stock una vez al día, lo queremos hacer al instante. ¡Mucho mejor! ¿No te parece?

Debés tener en cuenta que, a diferencia de los procedimientos, los triggers no los puede correr el usuario cuando el quiera, solo corren cuando ocurre el evento para el que han sido creados. Por lo tanto, es obvio que no tienen parámetros.

La sentencia típica para crear un trigger es la siguiente:

```
CREATE TRIGGER <Nombre del trigger> <evento que lo dispara> ON <objeto>
FOR EACH ROW
BEGIN
…
END;
```

<evento que lo dispara> debe ser de la forma BEFORE o AFTER y a continuación INSERT, UPDATE o DELETE.

Por ejemplo si deseamos crear un trigger que se dispare despues de un insert en la tabla productos sería de esta manera:

```
CREATE TRIGGER after_productos_insert AFTER INSERT ON productos
…
```

Recordá que debés establecer el `DELIMITER` antes y después del código del trigger al igual que con los procedimientos.

```
DELIMITER //
CREATE TRIGGER after_productos_insert AFTER INSERT ON pedidos_productos
FOR EACH ROW
BEGIN
UPDATE productos
SET stock=stock –NEW.cant,
Fechamod=CURDATE()
WHERE codigo=NEW.codproducto;
END //
DELIMITER ;
```

NEW y OLD representan el registro de la tabla que es objeto del trigger.

NEW es el registro que está siendo insertado y OLD es el viejo registro.

Se utiliza OLD cuando se está haciendo un update para representar la fila que es modificada (la vieja fila) y NEW para representar los nuevos valores que están siendo utilizados para realizar la modificación.

Por lo tanto podés ver que en este trigger, lo que estamos haciendo con los datos que se han insertado en la tabla pedidos_productos, es tomar la cantidad y restársela al stock de la tabla productos para el producto específico del renglón que se está tratando.

De esta manera, cada vez que se ingrese un renglón de pedido, se actualizará el stock del producto correspondiente.

Ahora bien, seguramente te preguntarás ¿Qué sucede si el pedido tiene muchos renglones?, ¿Cómo funciona en esos casos?

¡Es una muy buena pregunta!

Pues bien, lo que sucede es que cuando se haga el insert de cada renglón de pedido (o sea un registro en la tabla pedidos_productos) se estará haciendo además de ese insert, un update de la tabla productos para el producto afectado por la fila. Es decir, un insert se ha transformado en realidad en dos operaciones: **un insert y un update.**

Entonces, podés darte cuenta que no es muy bueno que usemos tantos triggers, porque pueden afectar el comportamiento de las aplicaciones y éstas se pueden hacer más lentas debido a tantas operaciones en la base de datos.

Ésta es precisamente una de las razones por las cuales no se recomienda usar mucho los triggers.

Una aplicación bastante común de los triggers es para auditar cambios en las tablas, o la actividad de los usuarios o, por ejemplo, para casos de sensibilidad de datos (o sea datos que son muy importantes para las empresas), en los que se quiera saber quienes fueron los usuarios que borraron los datos de una tabla, específicamente.

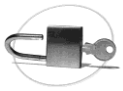
Para hacer eso, lo que se necesita primero es una tabla donde ir guardando los datos que se auditarán. La tabla de auditoría generalmente tiene la misma estructura de la tabla original pero con el agregado de las siguientes columnas:

FechaMov
TipoMov
Usuario

Para obtener el usuario, se utiliza la función `USER()`
Los demás datos provendrán de las tablas NEW, OLD o del código SQL que pongamos en el trigger.

ACTIVIDAD 6

A continuación le pedimos que realicelos ejercicios avanzados de la práctica 3 y la práctica 4 del campus. Recuerde que puede comentar con sus compañeros en el foro de la unidad sobre las diferentes formas de abordar los ejercicios. Se revisarán los mismos en el encuentro virtual de la semana.



ABORDEMOS EL LOGRO DEL OCTAVO OBJETIVO DE LA UNIDAD:

- ✓ Identificar los diferentes tipos de usuarios grupos y permisos.

9. Creación de usuarios y roles

A continuación vamos a aprender a organizar los accesos a la base de datos de tal manera que sólo aquellas personas que deben interactuar con ella tengan los permisos, y que dichos permisos sean los adecuados. Esto es, que las personas tengan asignado el mínimo permiso posible que les permita ejecutar sus tareas.

En primer lugar debemos pensar que dentro de la organización que utiliza nuestra base de datos trabajan muchas personas. Todas estas personas van a necesitar acceder a los datos de alguna u otra manera. Por lo tanto hay que crear los usuarios para ellos dentro de la base de datos.

La creación de un usuario se realiza con el siguiente comando:

```
CREATE USER <nombre de usuario>@<servidor> IDENTIFIED BY <contraseña>;
```

Los nombres de usuario y contraseña se ingresan entre comillas simples, veremos un ejemplo más abajo.

Además es posible que haya algunos usuarios que deban tener los mismas tareas dentro de la base de datos, de la misma forma que tienen las mismas tareas en la organización.

Similarmente hay que crear roles para agrupar usuarios con tareas similares, lo cual se realiza con el siguiente comando:

```
CREATE ROLE <nombre del grupo1>,<nombre del grupo2>…;
```

Es decir que se pueden crear muchos roles a la vez. Un detalle que debés tener en cuenta es que al igual que los nombres de los usuarios, los de los roles deben ingresarse entre comillas simples.

Y para finalizar veremos cómo se asignan los usuarios a los diferentes roles que se han creado en la base:

```
GRANT <nombre del grupo> TO <nombre del usuario>;
```

Si tenemos un usuario que se llama Juan y que pertenece al grupo Ventas veamos como serían todos los pasos:

```
CREATE USER 'Juan'@'localhost' IDENTIFIED BY 'pwd_segura#!?';
CREATE ROLE grventas;
GRANT grventas TO 'Juan'@'localhost';
```

10. Permisos

Hasta ahora hemos creado los usuarios y roles, pero no hemos garantizado ningún acceso, con lo cual ellos aún no pueden realizar ninguna actividad dentro de la base de datos.

Por ejemplo, si una persona pertenece al equipo de recursos humanos de una empresa, ella no debería tener permiso de ver las ventas, ni los datos de los clientes ya que no le compete a su actividad diaria. De manera similar, las personas que trabajan en el equipo de ventas o marketing, no deben tener permiso de ver los datos de los empleados, aunque sí deben conocer sus nombres, departamento al que pertenecen y su interno.

Recordarás que hemos tratado este tema en la Unidad 1 cuando hablamos de los administradores, desarrolladores y el nivel de vistas de la base de datos.

Para poder lograrlo, vamos a crear los usuarios y a éstos les daremos permisos para realizar las diferentes tareas de acuerdo a los accesos que debemos garantizarles.

Pero además crearemos los roles donde podremos agrupar a los usuarios que van a tener los mismos tipos de permisos.

Comencemos entonces pensando un ejemplo de cómo organizaremos la seguridad en nuestra base de datos ventas. Seguiremos un proceso similar al que se realiza en la realidad cuando se asignan permisos:

Para empezar, tenemos los datos de los clientes, sus locales, y los vendedores que los deben atender. Tenemos dos empleados que se ocupan de los canales de ventas que son los únicos que pueden agregar, borrar o modificar datos de estas tablas. Pero, podría ser que en el futuro tengamos más empleados con esa misma tarea.

Las ventas las manejan los empleados de ese departamento y ellos son los únicos que pueden ingresar o modificar pedidos. Pero recordemos que cuando se cargan los pedidos se dispara un trigger que modifica el stock de los productos. O sea que estos usuarios deben tener permiso de modificar los productos

Por último tenemos a los encargados de los productos que son los únicos que pueden agregar productos nuevos o modificar sus datos tales como el precio o la descripción.

Queda entonces:

Usuario	Departamento	Grupo/Role	Tabla	Permisos
Juan	Ventas	grVentas	Pedidos/Pedidos_Producto Productos	S-I-U-D U
María	Ventas	grVentas	Pedidos/Pedidos_Producto Productos	S-I-U-D U
Pedro	Ventas	grVentas	Pedidos/Pedidos_Producto Productos	S-I-U-D U
Celeste	Canales	grCanales	Clientes/Locales/Vendedores	S-I-U-D
Luis	Canales	grCanales	Clientes/Locales/Vendedores	S-I-U-D
Pablo	Producto	grProducto	Productos	S-I-U-D
Mariela	Producto	grProducto	Productos	S-I-U-D
Gabriel	Sistemas	grDBA	Todas	Administrador
Catalina	Sistemas	grDesarrollo	Todas	Create-Alter-Drop

Del apartado anterior ya tenemos creados los usuarios y los roles. Resta entonces garantizar los accesos a los diferentes roles de acuerdo a nuestra definición.

Para garantizar los accesos a un usuario o un rol, se debe ejecutar el siguiente comando:
`GRANT` <permiso> `ON` <base de datos>.<tabla> `TO` <usuario o rol>;

Por ejemplo para el primer renglón de nuestra tabla de permisos deberíamos hacer:

```
GRANT select, insert, update, delete ON ventas.pedidos TO grventas;
GRANT select, insert, update, delete ON ventas.pedidos_productos TO grventas;
```

Ahora ya podemos estar seguros que los usuarios tendrán el mínimo permiso necesario para realizar sus actividades. Estos son los tipos de permisos que se pueden asignar a usuarios y roles:

*`ALL PRIVILEGES`: son todos los permisos listados abajo.
*`CREATE`: para crear nuevas tablas o bases de datos.
*`DROP`: para eliminar tablas o bases de datos.
*`DELETE`: para eliminar registros de tablas.
*`INSERT`: para insertar registros en tablas.
*`SELECT`: para leer registros en las tablas.
*`UPDATE`: para actualizar registros seleccionados en tablas.
*`EXECUTE`: para ejecutar procedimientos almacenados.
*`GRANT OPTION`: permite otorgar o remover privilegios a otros usuarios usuarios.

Así como podemos dar permisos también los podemos quitar, el comando que se utiliza para quitar permisos es `REVOKE` y es diferente al `GRANT` en su estructura.

Por ejemplo para quitar el permiso otorgado a grVentas se haría lo siguiente:

```
REVOKE select, insert, update, delete ON ventas.pedidos_productos FROM grVentas;
```

Además de dar permisos a las tablas, recordá que con el uso de vistas podemos exponer únicamente las columnas que necesite ver el usuario con lo cual podemos agregar un nivel más a la seguridad de nuestras bases de datos. Por ejemplo podemos crear una vista de una tabla empleados que solo contenga su nombre, interno y departamento. Luego le damos permiso de `SELECT` sobre esa vista a los demás empleados. Solo los empleados de HR pueden tener permiso de `SELECT` sobre la tabla empleados completa (con todas sus columnas)

Es buen momento de ver cómo se realizan estas actividades y la interacción entre roles usuarios y los datos. Te presentamos en el campus un video titulado “Seguridad y Permisos”.

ACTIVIDAD 7

Presentamos la Actividad 7 en el campus. La solución de esta actividad la puede obtener bajando el archivo titulado “Actividad 8” del campus

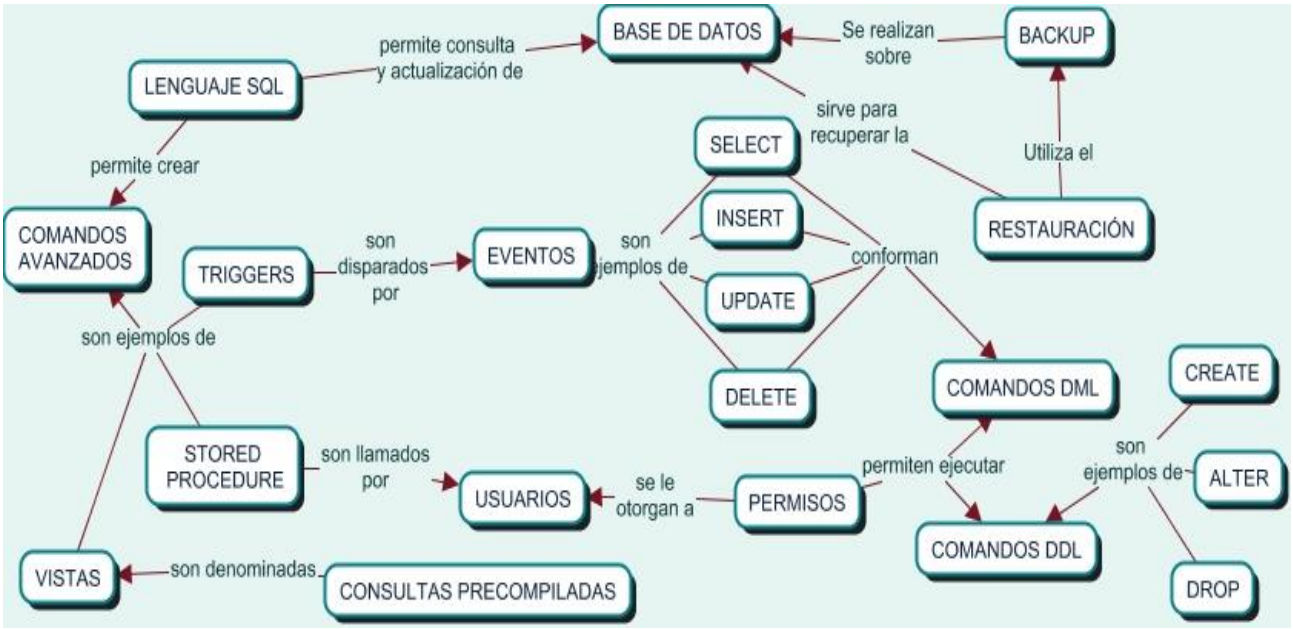
Si la Actividades fueron realizadas correctamente habrás logrado comprender las principales características del lenguaje SQL, el manejo de usuarios y permisos y la recuperación de las bases de datos.

Has logrado alcanzar los objetivos propuestos para esta Unidad.
¡FELICITACIONES!

SÍNTESIS DE LA UNIDAD

Hagamos un resumen de todo lo visto en esta Unidad.

Los COMANDOS AVANZADOS del LENGUAJE SQL incluyen VISTAS, STORED PROCEDURES y TRIGGERS. Las VISTAS son CONSULTAS precompiladas que se guardan en el DBMS para ser utilizadas posteriormente. Los STORED PROCEDURES y los TRIGGERS son porciones de código precompilado que ejecuta secuencias de acciones más complejas. Los STORED PROCEDURES pueden ser solo ejecutados por los USUARIOS pero los TRIGGERS son disparados por EVENTOS que afectan las TABLAS tales como operaciones de INSERT, UPDATE o DELETE.
Los USUARIOS tienen PERMISOS de realizar OPERACIONES DML de SELECT, INSERT, UPDATE y DELETE, y DDL como CREATE, ALTER y DROP
Se realizan BACKUPS de las BASES DE DATOS para protegerlas de catástrofes. Luego, a partir del BACKUP se realiza la RESTAURACIÓN con la que podremos recuperar la BASE DE DATOS al momento del backup.



AUTO-EVALUACIÓN

Aquí le ofrecemos la posibilidad de reflexionar sobre su aprendizaje y sobre la forma como está organizando y llevando a cabo su trabajo. Es una herramienta muy importante. Úsela y si tiene comentarios para compartir, póngalos en el Foro de Interacción con los Tutores o si no son consultas y sólo quiere compartir, en el Bar.

Aspecto a evaluar	MB	B	R	M
1. Mi distribución del tiempo de estudio y trabajo.				
2. Mi entrenamiento en técnicas de estudio en esta unidad. (¿Lo hice, aprendí, usé lo que aprendí?)				
3. Nuevos aprendizajes.				
4. Mi participación en los foros.				
5. Mi participación en el Glosario.				
6. Mi manejo del campus y medios técnicos.				
7. Mi contacto con los compañeros. (¿Intercambio ideas, socializo en el bar?)				
8. Mi contacto con los tutores (¿Pregunto, comento, pido aclaraciones?)				

Considere: ¿Qué debe mejorar?