Apunte Alumnos

Creado miércoles 24 mayo 2023

SQL - Apunte informal

¿Que es una base de datos relacional?
https://www.oracle.com/ar/database/what-is-a-relational-database/

SQL es un lenguaje de computación para trabajar con conjuntos de datos y las relaciones entre ellos. Los programas de bases de datos relacionales, como Microsoft Office Access, usan SQL para trabajar con datos.

Instalacion de software:

XAMPP:

- PHP
- Apache
- MariaDB
- phpMyAdmin
- Perl
- OpenSSL

https://dev.mysql.com/downloads

Crear una base de datos de prueba

- Usamos phpMyAdmin
- Creamos una nueva base de datos vacia, usando el menu o con

```
CREATE DATABASE empresa
```

- Desde **Import** buscamos el archivo y lo importamos. Aparecen los datos en la base de datos.

Sentencias DDL: CREATE, ALTER y DROP

DDL (Data Description Language) Lenguaje de Descripción de Datos.
Las sentencias incluidas en este grupo son normalmente usadas por el administrador de la BD (Base de Datos) debido a que permiten definir gran parte del nivel interno de esta. Se trata de sentencias para crear la BD, crear, eliminar o modificar estructura de tablas, definir relaciones entre tablas, entre otras.

1. Se empiezan por un verbo indicando la acción a realizar.
2. Continúan completando con un objeto sobre el cual se realiza la acción.
3. Se sigue por una serie de cláusulas, obligatorias y opcionales, que especifican a detalle lo que se quiere hacer.

Creando una Base de Datos

CREATE: Crear una base de datos
Con **CREATE DATABASE** creamos una base de datos con el nombre indicado en esa orden.
Es necesario tener permisos del sistema de base de datos. Un equivalente es CREATE SCHEMA.
Si la base de datos ya existe se produce un error, para salvarlo podemos especificarle **IF NOT EXISTS**.

```
CREATE DATABASE empresa;

CREATE DATABASE IF NOT EXISTS empresa;
```

Crear una Tabla

Si intentamos crear una tabla con un nombre ya existente (existe otra tabla con ese nombre), mostrará un mensaje de error indicando que la acción no se realizó porque ya existe una tabla con el mismo nombre.

Se produce un error si la tabla ya existe y no le especificaste IF NOT EXISTS.

```
USE empresa;
CREATE TABLE IF NOT EXISTS empleados (
        id int (4),
        apellido varchar (30),
        nombre varchar (30),
        genero varchar (2),
        domicilio varchar (40),
        provincia varchar (20),
        estadoCivil varchar (15),
        hijos int (2),
        sueldo int (7),
        PRIMARY KEY (`id`)
);
```

Ver las tablas de una Base de Datos

```
SHOW TABLES;
```

Ver la estructura de una Tabla

```
DESCRIBE empleados;
```

Modificar una Tabla

Para agregar, eliminar o modificar una columna utilizamos la sentencia **ALTER**
Para agregar una columna:

```
ALTER TABLE empresa.empleados ADD correo varchar(50);
DESCRIBE empleados;

ALTER TABLE empresa.empleados ADD (correo varchar(50), ciudad varchar(40) )
```

Para eliminar una columna:

```
ALTER TABLE empresa.empleados DROP COLUMN correo;
DESCRIBE empleados;
```

Eliminar una Tabla

Para eliminar una tabla usamos **DROP TABLE**. Tipeamos:

```
DROP TABLE empresa.empleados;
```

Si tipeamos nuevamente:

```
DROP TABLE empresa.empleados;
```

Aparece un mensaje de error, *indicando que no existe, ya que intentamos borrar una tabla inexistente.* Para evitar este mensaje podemos tipear:

```
DROP TABLE IF EXISTS empresa.empleados;
```

Eliminar una Base de Datos:

DROP DATABASE empresa

Sentencias DML: Lenguaje de Manipulación de Datos

Este grupo está compuesto por las instrucciones más usadas por el usuario ya que se trata de aquellas que requieren el manejo de datos como insertar nuevos registros, modificar datos existentes, eliminarlos y hasta recuperar datos de la BD. Para este grupo, el usuario solo debe indicar mediante las instrucciones lo que quiere recuperar de la BD y no precisamente el cómo se debe de recuperar puesto que no influye cómo estén almacenados los datos.

Sentencias SQL - Insertar y modificar datos

Para agregar registros utilizamos **INSERT INTO** nombredelatabla **VALUES** y estos datos van separados por comas en el mismo orden en que fueron incorporados los campos.

```
INSERT INTO empleados (id, apellido, nombre, genero, domicilio, provincia, estadoCivil, hijos, sueldo) VALUES (250,"Lopez",
"Marta", "F","Mitre 111","Jujuy", "Casado/a",0, 123000);
```

- Hubo que decidir que id usar. Si estaba repetido da un error.
- Si probamos de cargar dos veces un registro con el mismo id, nos va a dar un error.
- Hubiese sido mejor tener el id AUTO_INCREMENT, asi:

```
CREATE TABLE `empleados`(
        id int (4) AUTO_INCREMENT,
        apellido varchar (30),
        nombre varchar (30),
        genero varchar (2),
        domicilio varchar (40),
        provincia varchar (20)A,
        estadoCivil varchar (15),
        hijos int (2),
        sueldo int (7),
        PRIMARY KEY (`id`)
 );
```

Agregar mas de un registro:

```
INSERT INTO empleados
VALUES (250,"Perez", "Ana", "F","Godoy 311","Jujuy", "Casado/a",2, 223000),
(250,"Gomez", "Julia", "F","Altes 411","Córdoba", "Casado/a",1, 187800),
(250,"Mazzola", "Carlos", "M","Azcona 11","Buenos Aires", "Casado/a",0, 155000);
```

Borrar registro:

```
DELETE FROM empleados WHERE id = 2
SELECT
/* OJO CON EL WHERE !!!!!   */
```

IMPORTANTE: De no poner la instrucción WHERE se eliminarán todos los registros de la tabla. La tabla seguirá existiendo, pero será una tabla vacía, sin ningún registro (sólo se mantiene la estructura).

Modificar registro:

UPDATE empleados SET nombre = "Ana Maria", sueldo = 333000 WHERE id = 1;

OJO CON EL WHERE !!!!!

Sentencias SQL - Consultas

- **SELECT**: cláusula utilizada para especificar qué atributo (dato) se pretende obtener.
- **FROM**: es utilizada en conjunto con el **SELECT** para especificar desde qué tabla
- (entidad) se pretende traer el dato.
- **WHERE**: cláusula para proponer una condición específica
- **ORDER BY:** es utilizada para especificar por qué criterio se pretende **ordenar** los registros de una tabla.

- **GROUP BY**: es utilizada para especificar por qué criterio se deben **agrupar** los registros de una tabla.
- **LIMIT:** Limita el nro de registros devueltos

```
-- 1) Seleccionar todos los datos de los empleados
SELECT * FROM empleados;

-- 2) Seleccionar solamente apellido, nombre y domicilio los empleados:
SELECT apellido, nombre, domicilio FROM empleados;

-- 3) Mostrar todos los datos ordenados por apellido:
SELECT * FROM empleados ORDER BY apellido;

-- 4) Mostrar todos los datos ordenados por apellido y nombre:
SELECT * FROM empleados ORDER BY apellido, nombre;

-- 5) Mostrar todos los datos ordenados provincia en forma descendente y sueldo en porma ascendente:
SELECT * FROM empleados ORDER BY provincia DESC, sueldo;

-- 6) Mostrar un ranking de los 10 mejoreas sueldos
SELECT * FROM empleados ORDER BY sueldo DESC LIMIT 10;

-- 7) Mostrar todos los datos de aquellos empleados que tengan 3 hijos
SELECT * FROM empleados WHERE hijos = 3;

-- 8) Mostrar todos los datos de aquellos empleados que no estén Casados
SELECT * FROM empleados WHERE estadoCivil != 'Casado/a';

-- 8) Mostrar todos los datos de aquellos empleados que ganen menos de 100.000
SELECT * FROM empleados WHERE sueldo < 100000;

-- 9) Mostrar apellido y nombre de aquellos empleados que ganen menos de 100.000 (no mostrar sueldo)
SELECT apellido, nombre FROM empleados WHERE sueldo < 100000;

-- 10)  Mostrar nombre, apellido y provincia de aquellos empleados cuyo sueldo se encuentre entre 150.000 y 250.000
SELECT nombre, apellido, provincia, sueldo FROM empleados WHERE sueldo >= 100000 AND sueldo <= 250000;

-- 11)  Repetir el ejercicio anterior, utilizando BETWEEN
SELECT nombre, apellido, provincia, sueldo FROM empleados WHERE sueldo BETWEEN 100000 AND 250000;

-- 12) Mostrar todos los datos de los empleados que vivan en Buenos Aires o Jujuy
SELECT * FROM empleados WHERE provincia = "Buenos Aires" OR provincia ="Jujuy";
```

Operador LIKE

se utiliza para comparaciones con campos de tipo de cadenas de texto.
Esta sentencia se podría utilizar para consultar cuáles son los clientes que viven en una calle que contiene el texto “San Martín”.
Al colocar el % al comienzo y al final estamos representando un texto que no nos preocupa cómo comienza ni cómo termina, siempre y cuando contenga la/s palabra/s que nos interesa.

Operador	Descripción
LIKE	Define un patrón de búsqueda y utiliza % y _
NOT LIKE	Negación de LIKE
IS NULL	Verifica si el Valor es NULL
IS NOT NULL	Verifica si el Valor es diferente de NULL
IN ()	Valores que coinciden en una lista
BETWEEN	Valores en un Rango (incluye los extremos)

```
-- 13) Mostrar todos los datos de los empleados de apellido Carrizo
SELECT * FROM empleados WHERE apellido LIKE 'Carrizo';

-- 14) Mostrar todos los datos de los empleados cuyo apellido no sea Ayala
SELECT * FROM empleados WHERE apellido NOT LIKE 'Ayala' order by apellido;

-- 15) Mostrar todos los datos de los empleados cuyo nombre comience con "A"
SELECT * FROM empleados WHERE nombre LIKE 'A%';

-- 16) Mostrar todos los datos de los empleados cuyo apellido termine con "S"
SELECT * FROM empleados WHERE apellido LIKE '%S';

-- 17) Mostrar todos los datos de los empleados cuyo domicilio contenga "Z"
SELECT * FROM empleados WHERE domicilio LIKE '%Z%';

-- 18) Mostrar todos los datos de los empleados que tengan provincias
SELECT * FROM empleados WHERE provincia IS NOT NULL;

-- 19) Mostrar todos los datos de los empleados que no tengan provincias
SELECT * FROM empleados WHERE provincia IS NULL;

-- 20) Mostrar nmbre, apellido, domicilio y estado civil de todos los datos de los empleados
--     Renombrar las columnas para que se llamen Apellido, Nombre, Dirección y Estado Civil
SELECT apellido AS "Apellido", nombre AS "Nombre", domicilio AS "Dirección", estadoCivil AS "Estado civil" FROM empleados;

-- 21) Mostrar todos los datos de los empleados que vivan en las provincias de Buenos Aires, Córdoba y Mendoza
SELECT * FROM empleados WHERE provincia IN ('Buenos Aires', 'Córdoba', 'Mendoza');
```

SELECT DISTINCT:

Se usa para devolver solo valores distintos (diferentes) de una columna que puede tener registros duplicados. Sintaxis:
SELECT DISTINCT column1, column2, ...

FROM table_name;

Dentro de una tabla, una columna a menudo contiene muchos valores duplicados; y a veces solo quieres enumerar los diferentes valores (distintos).
La instrucción SELECT DISTINCT se usa para devolver solo valores distintos (diferentes).

```
-- 22) Obtener un listado de las provincias sin repeticiones
SELECT DISTINCT provincia FROM empleados;
```

SELECT: uso de ALIAS

Es recurrente en el desarrollo de consultas o sentencias SQL extensas el uso de ALIAS.

Esta propiedad es extensible tanto para tablas como para campos y permite renombrar los nombres originales de tablas o campos de manera temporal.
El uso de ALIAS presenta algunas ventajas:

- Permite acelerar la escritura de código SQL
- Mejorar la legibilidad de las sentencias
- Ocultar/Renombrar los nombres reales de las tablas o campos a usuarios
- Permite asignar un nombre a una expresión, fórmula o campo calculado

Ejemplo: renombrar tablas y atributos calculados

```
SELECT country.name, country.population FROM country WHERE country.name BETWEEN "Paraguay" AND "Spain" ORDER BY country.name

SELECT C.name, C.population FROM country AS C WHERE C.name BETWEEN "Paraguay" AND "Spain" ORDER BY C.name
```

Indices

Tipos de índices disponibles en SQL:
Se indican los tipos de índice disponibles en SQL Server y se proporcionan vínculos a información adicional.
https://docs.microsoft.com/es-es/sql/relational-databases/indexes/indexes?view=sql-server-ver16

Guía de diseño y de arquitectura de índices de SQL:
Los índices mal diseñados y la falta de índices constituyen las principales fuentes de atascos en aplicaciones de base de datos. El diseño eficaz de los índices tiene gran importancia para conseguir un buen rendimiento de una base de datos y una aplicación. Esta guía de diseño de índices contiene información sobre la arquitectura de índices y prácticas recomendadas que le ayudarán a diseñar índices eficaces que resuelvan las necesidades de la aplicación.
https://docs.microsoft.com/es-es/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16

Cláusula JOIN

El **JOIN** se utiliza para indicar la manera en que se están relacionando las tablas, es decir, con qué atributos se está plasmando la relación entre ellas.

¿Por qué tenemos que relacionar? **Para no duplicar datos.** Si tenemos una tabla escuelas y un alumno que pertenece una escuela no debemos repetir los datos de la escuela en la tabla alumnos.
Las tablas deben estar **normalizadas**, esto quiere decir que los datos deben estar almacenados en forma eficiente para poder hacer consultas más rápidas, para que la BD no pese tanto, para que las tablas no tengan tantos campos (esto es área del diseño de BD).
Para unir las tablas vamos a necesitar un dato que relacione a ambas tablas, que las una.

Usamos la base de datos casa_electrodomesticos como ejemplo.

INNER JOIN:

Muestra solamente los registros que coincidan entre ambas tablas.
Por ejemplo: si un cliente no tiene una provincia relacionada esa consulta no los mostrará, del mismo modo si una provincia no tiene asignada algún cliente tampoco lo mostrará.

```
-- Mostrar todos los clientes que tienen una provincia asignada
SELECT clientes.id, clientes.apeynom , provincias.provincia
FROM clientes
INNER JOIN provincias ON clientes.provincia = provincias.id;
```

Podemos usar alias para que sea mas corta la sentencia, y mas entendible:

```
-- Mostrar todos los clientes que tienen una provincia asignada
SELECT c.id, c.apeynom , p.provincia
FROM clientes AS c
INNER JOIN provincias AS p ON c.provincia = p.id;
```

Y por supuesto, podemos usea WHERE, BETWEEN, etc:

```
-- Mostrar todos los clientes que tienen una provincia asignada
SELECT c.id, c.apeynom , p.provincia
FROM clientes AS c
INNER JOIN provincias AS p ON c.provincia = p.id
WHERE c.id < 20
ORDER BY c.id;
```

Ejemplos de JOINS:

```
-- 1) Mostrar id, apellido y nombre y provincia de los clientes
SELECT clientes.id, clientes.apeynom as "Nombre completo", clientes.provincia, provincias.provincia
FROM clientes
INNER JOIN provincias ON clientes.provincia = provincias.id;
```

```
-- 2) Mostrar id, modelo y el
 nombre de la marca de los productos, utilizar alias de tabla
SELECT prod.id, prod.modelo, mar.marca
FROM productos prod
INNER JOIN marcas mar ON prod.marca = mar.id;
```

```
-- 3) Obtener un listado completo de productos, incluyendo tablas relacionadas
SELECT prod.id, cate.categoria, mar.marca, prod.modelo,
prov.proveedor, prod.pantalla_pulg, prod.camara_mpx, prod.mem_int, prod.precio
FROM productos prod
INNER JOIN categorias cate ON prod.categ = cate.id
INNER JOIN marcas mar ON prod.marca = mar.id
INNER JOIN proveedores prov ON prod.prov = prov.id;
```

```
-- 4) Obtener un listado completo de ventas, incluyendo nombre del cliente, producto y precio total
SELECT vtas.id, cli.apeynom AS 'Nombre completo', mar.marca, prod.modelo, vtas.cant, prod.precio,
(vtas.cant * prod.precio) AS 'Total', vtas.cuotas,
(vtas.cant * prod.precio)/vtas.cuotas AS 'Valor Cuota'
FROM ventas vtas
INNER JOIN clientes cli ON vtas.id_cli = cli.id
INNER JOIN productos prod ON vtas.id_prod = prod.id
INNER JOIN marcas mar ON prod.marca = mar.id;
```

LEFT JOIN:

Tiene como condición que figure en, al menos una tabla. Left indica que va a tomar como tabla principal la de la izquierda. **De esa tabla muestra todos los registros de esa tabla, sin importar si tiene registros asociados en la otra tabla.**

```
-- 1) Mostrar id, apellido y nombre y nombre de provincia de los clientes,
--    tengan o no una provincia asignada
SELECT *
FROM clientes cli
LEFT JOIN provincias prov ON cli.provincia = prov.id;
```

Si un cliente no tiene provincia asociada, aparece igual en el resultado.

RIGHT JOIN

Tiene como condición que el resultado figure en al menos una tabla. **Right indica que va a tomar como tabla principal la de la derecha.** De esa tabla muestra todos los registros, sin importar si tiene registros asociados en la otra tabla.

```
-- RIGHT JOIN
SELECT *
FROM clientes AS cli
RIGHT JOIN provincias AS prov
ON cli.provincia = prov.id;
```

FULL OUTER JOIN

Devuelve todos los elementos, esten en una O en otra tabla.

Esto NO ANDA en MariaDB

```
SELECT *
FROM clientes AS cli
FULL OUTER JOIN provincias AS prov
ON cli.provincia = prov.id;
```

Pero se puede obtener el resultado necesario uniendo los dos JOIN anteriores:

```
-- LEFT JOIN
SELECT *
FROM clientes AS cli
LEFT JOIN provincias AS prov
ON cli.provincia = prov.id

UNION

-- RIGHT JOIN
SELECT *
FROM clientes AS cli
RIGHT JOIN provincias AS prov
ON cli.provincia = prov.id;
```

Funciones de agregación:

Las funciones de agregación más comunes disponibles en el lenguaje son: **SUM**, **AVG**, **MAX**, **MIN**, **COUNT**.
Reúnen un conjunto de registros para agrupar los datos y llevar a cabo la operación en cuestión (suma, promedio, cuenta, etc), de ahí la necesidad de la cláusula **GROUP BY**.

```
-- Función SUM para calcular la suma total de los precios de los productos:
SELECT SUM(precio) AS total_precios
FROM productos;
```

Este ejemplo calcula la suma total de la columna "precio" en la tabla "productos" y la muestra como resultado en una columna llamada "total_precios".

MIN, MAX: Obtiene el valor mínimo/maximo de una columna.
COUNT: Cuenta la cantidad:

```
SELECT COUNT(*) AS total_productos
FROM productos;
```

Y podemos poner todo lo visto junto:

```
SELECT c.categoria, SUM(p.precio) AS total_precios
FROM productos AS p
JOIN categorias AS c ON p.categ = c.id
GROUP BY c.categoria;
```

En este ejemplo, se realiza una unión (JOIN) entre la tabla "productos" y la tabla "categorias" utilizando la columna "categ" en ambas tablas como criterio de unión. Luego, se utiliza la cláusula GROUP BY para agrupar los resultados por la columna "categoria" de la tabla "categorias". Por último, se aplica la función de agregación SUM en la columna "precio" de la tabla "productos" para calcular la suma total de precios para cada categoría.

El resultado de esta consulta mostrará las categorías de productos y la suma total de precios correspondiente a cada categoría.

HAVING / WHERE

Es importante entender la forma en como las clausulas WHERE y HAVING actuan sobre las funciones de agregacion y agrupacion en SQL, la diferencia fundamental entre WHERE y HAVING es la siguente: WHERE selecciona las filas a mostrar antes de que sean agrupadas o procesadas por una funcion de agregacion, mientras HAVING selecciona las filas despues de que estas hayan sido procesadas o computadas, por lo tanto, la clausula WHERE no debe contener funciones de agrupacion o agregacion, mientras que la clausula HAVING siempre contiene funciones de agregacion, es permitido escribir clausulas HAVING que no contengan agrupacion, pero rara vez es util, la misma condicion podria ser usada mas eficientemente en la clausula WHERE

Consulta utilizando HAVING:
Esta consulta te mostrará las provincias que tienen al menos 3 clientes registrados

```
SELECT p.provincia, COUNT(c.id) AS total_clientes
FROM provincias p
INNER JOIN clientes c ON p.id = c.provincia
GROUP BY p.provincia
HAVING COUNT(c.id) >= 40;
```

La cláusula INNER JOIN se utiliza para unir las tablas "provincias" y "clientes" basándose en la clave foránea "provincia".
Utilizamos GROUP BY para agrupar los resultados por provincia.
La cláusula HAVING se utiliza para filtrar los resultados y mostrar solo aquellas provincias que tienen al menos 40 clientes registrados.

Consulta utilizando WHERE:
Es posible obtener los mismos resultados utilizando la cláusula WHERE en lugar de HAVING. Sin embargo, hay una diferencia importante en cómo se aplican estas cláusulas:

- La cláusula **WHERE** se utiliza para filtrar filas **antes** de que se realice la operación de agrupación (GROUP BY).
- La cláusula **HAVING** se utiliza para filtrar los resultados **después** de que se ha realizado la operación de agrupación.

En el caso de la consulta del ejemplo, el uso de HAVING es necesario porque se está filtrando por una condición basada en la función de agregación COUNT().

Dicho esto, si deseamos obtener los mismos resultados utilizando WHERE en lugar de HAVING, necesitamos realizar una consulta anidada para aplicar el filtro de COUNT() antes de la operación de agrupación.

```
SELECT subconsulta.provincia, total_clientes
FROM (
  SELECT p.provincia, COUNT(c.id) AS total_clientes
  FROM provincias AS p
  INNER JOIN clientes AS c ON p.id = c.provincia
  GROUP BY c.provincia
) AS subconsulta
WHERE total_clientes >= 30;
```

En esta consulta anidada, se realiza una **subconsulta** que calcula el número de clientes por provincia y se agrupa por provincia. Luego, en la consulta principal, se utiliza WHERE para filtrar las provincias con al menos 30 clientes.

Es importante tener en cuenta que, en este caso, utilizar HAVING es más eficiente y legible que utilizar una subconsulta con WHERE. Sin embargo, si tienes una situación específica en la que necesitas filtrar datos agregados antes de la operación de agrupación, puedes utilizar una subconsulta con WHERE para lograrlo.

Subconsultas:

```
SELECT id, apeynom, domcalle, domnro
FROM clientes
WHERE id IN (
  SELECT v.id_cli
  FROM ventas v
  INNER JOIN productos p ON v.id_prod = p.id
  INNER JOIN marcas m ON p.marca = m.id
  WHERE m.marca = 'LG'
);
```

En este ejemplo, la subconsulta se utiliza para obtener los ID de los clientes que han realizado compras de la marca especificada. Luego, la consulta principal selecciona los datos de los clientes que coinciden con esos ID.

La subconsulta (SELECT v.id_cli FROM ventas v INNER JOIN productos p ON v.id_prod = p.id INNER JOIN marcas m ON p.marca = m.id WHERE m.marca = 'nombre_de_la_marca') se encarga de seleccionar los ID de los clientes que han comprado productos de la marca específica.

Después, en la consulta principal, utilizamos la cláusula WHERE id IN (...) para seleccionar los datos de los clientes cuyo ID está presente en la subconsulta.

Recuerda reemplazar 'LG' con el nombre de la marca que deseas consultar.

Ambas formas (utilizando JOIN o subconsultas) pueden lograr el mismo resultado. La elección entre ellas dependerá de tus preferencias personales y de la estructura de tu base de datos.

WITH

WHITH c AS (SELECT p.provincia, COUNT(c.id) AS total_clientes
FROM provincias AS p
INNER JOIN clientes AS c ON p.id = c.provincia
GROUP BY c.provincia)
SELECT * FROM c WHERE total_clientes >= 40;