INGENIERÍA UNIVERSIDAD DE LOS ANDES MÉRIDA VENEZUELA | DEPARTAMENTO DE COMPUTACIÓN ESCUELA DE SISTEMAS

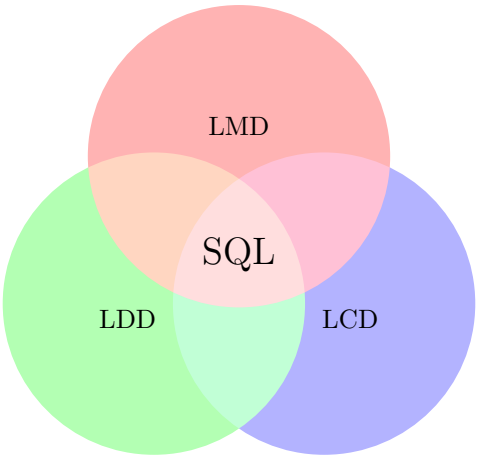
Laboratorio de Base de Datos
Práctica Nro. 5, Prelaboratorio Consultas SQL

Prof. Solazver Solé
Preps. Victor Albornoz, Yenifer Ramírez
Semestre B-2018

1. SQL

SQL por sus siglas en ingles (**Structured Querry Languaje**) [1] es un lenguaje de dominio específico, usado en la programación y diseñado para administrar los datos mantenidos por los sistemas de gestión de bases de datos relacionales (**RDBMS**).

Originalmente fue basado en el **álgebra relacional** y en **cálculo relacional de tuplas**, SQL se compone de un un lenguaje de definición de datos (**LDD**), un lenguaje de manipulación de datos (**LMD**) y un lenguaje de control de datos (**LCD**).



[1] Lenguaje estructurado de consultas

1.1. Lenguaje de definición de datos: LDD

El lenguaje SQL cuenta con sentencias de definición de datos que permiten crear tablas, alterar su definición y eliminarlas. En una base de datos relacional existen otros tipos de objetos además de las tablas, como las vistas, los índices y los disparadores, que se estudiarán más adelante. Las sentencias para crear, alterar y eliminar vistas e índices también pertenecen a este conjunto.

1.2. Lenguaje de manipulación de datos: LMD

Las sentencias de manipulación de datos en SQL permiten insertar datos en las tablas, consultarlos, modificarlos y borrarlos.

1.3. Lenguaje de manipulación de datos: LCD

SQL proporciona sentencias para que los administradores de bases de datos para realicen sus tareas relativas al control de los datos, por ejemplo crear usuarios y concederles o revocarles privilegios.

2. Operaciones de recuperación/actualización con SQL

El lenguaje SQL está compuesto por sentencias, cláusulas, operadores y funciones de agregación.(**También tiene predicados o palabras clave**). Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

2.1. Sentencias para realizar consultas y actualizaciones

SQL tiene una sentencia básica para recuperar información de una base de datos: La sentencia **SELECT**.

- SELECT : Utilizado para consultar tuplas de la base de datos que satisfagan un criterio determinado.
- INSERT : Utilizado para guardar lotes de datos en la base de datos en una única operación.
- UPDATE: Utilizado para modificar los valores de los atributos y tuplas especificadas.
- DELETE: Utilizado para eliminar registros de una tabla en una base de datos.

2.2. Cláusulas:

Son condiciones que deben cumplir los datos que se van a consultar o manipular.

- FROM: Especifica la(s) tabla(s) de la cual se van a seleccionar las tuplas.
- WHERE: Condiciones que deben reunir las tuplas a seleccionar.
- GROUP BY: Utilizada para calificar tuplas seleccionadas en grupos específicos
- HAVING: Condición a satisfacer por cada grupo.
- ORDER BY: Utilizada para ordenar tuplas específicas en un orden.

2.3. Operadores Lógicos

- AND, OR, NOT.

2.4. Operadores de comparación

- <, >, (<>,!=), <= , >=, =
- IS NULL: Devuelve true si es nulo.
- IS NOT NULL: Devuelve false si es nulo.
- x IN (v1,v2,v3...vn) Devuelve true si x=v1 o x = v2 ...
- BETWEEN: Utilizado para especificar un intervalo de valores.
- LIKE: Para comparación de una cadena de texto con una expresión regular.

2.5. Funciones de Agregación

Se usan dentro de una sentencia **SELECT**, en grupos de tuplas para devolver un único valor que se aplica a un grupo de tuplas.

- AVG: Utilizada para calcular el promedio de los valores de un atributo determinado
- COUNT: Utilizada para devolver el número de tuplas de la consulta
- SUM: Utilizada para devolver la suma de todos los valores de un atributo determinado
- MAX: Utilizada para devolver el valor más alto de un atributo especificado
- MIN: Utilizada para devolver el valor más bajo de un atributo especificado

3. Consultas básicas

Las consultas se utilizan para indicar al servidor de base de datos que devuelva información.

```
    -- bloque select-from-where-group by-having-order by

SELECT [ DISTINCT ] { * | columna [ , columna ] }
FROM nombre_tabla
[ WHERE condicion_de_busqueda ]
[ GROUP BY columna [, columna ]
[ HAVING condicion_para_el_grupo ] ]
[ ORDER BY columna [ ASC | DESC ]
[,columna [ ASC | DESC ] ];
```

3.1. Consultas de selección:

```
SELECT trackid,name,unitprice
FROM track
WHERE track.unitprice > 0.99
```

Podemos guardar la consulta como un resultado intermedio de la siguiente manera:

```
SELECT trackid,name,unitprice
INTO tracks_expensives
FROM track
WHERE track.unitprice > 0.99;
```

Luego podemos consultar el resultado intermedio como otra tabla de la base de datos.

```
SELECT *
FROM tracks_expensives;
```

Podemos modificar el orden en que se muestran los registros utilizando la clausula **ORDER BY** <lista de campos>. En donde lista de campos representan los campos a ordenar.

```
SELECT *
FROM invoice
ORDER BY invoicedate;
```

Podemos especificar el criterio de ordenamiento mediante las opciones ASC (ascendente) y DESC (descendente).

```

SELECT *
```

```
FROM invoice
ORDER BY invoicedate ASC,total DESC ;
```

3.2. Consultas con predicados:

Una manera de limitar el número de tuplas que devuelve el servidor es utilizar predicados en la selección. El predicado se incluye entre la claúsula y el primer nombre del atributo a recuperar, los posibles predicados son: * , **DISTINCT**, **DISTINCT ON** (colum_name)

```
--Sobre un campo especifico
SELECT DISTINCT ON (invoicedate) *
FROM invoice
ORDER BY invoicedate ASC;
```

```
--Sobre los registros
SELECT DISTINCT  customerid, billingcity
FROM invoice
ORDER BY customerid ASC;
```

En determinadas circunstancias es necesario asignar un nuevo nombre a alguna de los atributos devueltas por el servidor. Para ello tenemos la palabra reservada AS que se encarga de asignar el nombre que deseamos al atributo deseada:

```
SELECT DISTINCT
customerid as clienteid, billingcity as ciudad
FROM invoice as factura
WHERE factura.billingcity = 'Paris';
```

3.3. Uso de operadores en criterios de selección

Podemos utilizar los distintos operadores mencionados anteriormente para filtrar tuplas utilizando la cláusula **WHERE**.

```
SELECT *
FROM invoice as cliente
WHERE (total >= 5 and total <= 11)
and billingcountry = 'France';
```

```sql
SELECT *
FROM invoice as cliente
WHERE NOT (total >= 5 and total =< 11)
and billingcountry = 'France';
```

```sql
SELECT *
FROM invoice as cliente
WHERE (total BETWEEN 5 and 11) --inclusive
and billingcountry = 'France';
```

Operador **IN**: Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los en una lista.

```sql

SELECT *
FROM invoice as factura
WHERE NOT (total BETWEEN 5 and 11)
and  billingcountry IN ('France','USA','Canada');
```

Operador IS: Podemos usar este operador para comparar un atributo con el valor NULL.

```sql
SELECT *
FROM invoice as factura
WHERE cliente.billingstate IS NULL;
```

3.4. Agrupamiento y registros

- La cláusula **GROUP BY** forma grupos con las filas que tienen en común los valores de uno o varios atributos.
- Sobre cada grupo se pueden aplicar las funciones de atributo básicas: (**SUM**, **MAX**, **MIN**, **AVG**, **COUNT**), que pasan a denominarse funciones de grupo.
- Estas funciones, utilizadas en la cláusula **SELECT**, se aplican una vez para cada grupo.
- Se utiliza la cláusula **WHERE** para excluir aquellas tuplas que no se desean agrupar, y la cláusula **HAVING** para filtrar las tuplas una vez agrupados.
- Todos los campos de la lista de campos de **SELECT** deben incluirse en la cláusula **GROUP BY** o como argumentos de una función SQL agregada.

Para este ejemplo agrupamos la tabla invoice por su atributo billingcountry.

```sql
SELECT billingcountry
FROM invoice as factura
WHERE cliente.billingstate IS NOT NULL
GROUP BY billingcountry;
```

Ejemplo de función agregada.

```sql
SELECT customerid, SUM(total)
FROM invoice as cliente
WHERE cliente.billingstate IS NOT NULL
GROUP BY customerid;
```

Tomando un criterio de selección sobre los grupos:

```sql
SELECT customerid ,SUM(total)
FROM invoice as factura
WHERE factura.billingstate IS NOT NULL
GROUP BY customerid
HAVING SUM(total) > 40;
```

Para completar el bloque descrito anteriormente con todos las cláusulas se establece un orden en los registros.

```sql
SELECT customerid ,SUM(total)
FROM invoice as cliente
WHERE cliente.billingstate IS NOT NULL
GROUP BY customerid
HAVING SUM(total) > 40
ORDER BY SUM(total) DESC;
```

Estableciendo un resultado intermedio para encontrar el valor máximo de un atributo.

```sql
SELECT  customerid, SUM(total) INTO x
FROM invoice as factura
GROUP BY customerid;

SELECT MAX(sum)
FROM x;
```

Por último podemos utilizar el modificador **LIMIT** para limitar el número de tuplas que retorna la consulta.

```
SELECT
customerid ,COUNT(total),SUM(total),AVG(total),STDDEV(total)
FROM invoice as factura
WHERE factura.billingstate IS NOT NULL
GROUP BY customerid
HAVING SUM(total) > 40
ORDER BY customerid DESC
LIMIT 10;
```

3.5. Subconsultas

- Una subconsulta es una instrucción **SELECT** escrita entre paréntesis y anidada dentro de una instrucción **SELECT**, **SELECT ... INTO**, **DELETE** o **UPDATE**.
- En una subconsulta, se utiliza la instrucción SELECT para proporcionar un conjunto de uno o mas valores especificados para evaluar en la expresión de la cláusula **WHERE** o **HAVING**.
- Por lo general es usado con los predicados, **ALL**, **ANY/SOME**, **IN EXITS**.
    - **ALL**: El predicado ALL se utiliza para recuperar únicamente aquellas tuplas de la consulta principal que satisfacen la comparación con todas las tuplas recuperadas en la **subconsulta**.
    - **ANY/SOME**: El predicado ANY se utiliza para recuperar tuplas de la consulta principal, que satisfagan la comparación con cualquier otra tupla recuperada en la subconsulta.
    - **EXISTS**: El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve alguna tupla.
    - **IN**: El predicado IN se emplea para recuperar únicamente aquellas tuplas de la consulta principal para los que algunas tuplas de la subconsulta contienen un valor igual.

Ejemplos: Los 10 clientes que han hecho un consumo mayor a 40 dólares de manera descendente.

```
SELECT * FROM customer as cliente
WHERE customerid IN
(
SELECT customerid
FROM invoice as factura
GROUP BY customerid
HAVING SUM(total) > 40
ORDER BY SUM(total) DESC
)
LIMIT 10;
```

Listar todos los clientes si existe una factura cuyo cliente sea nulo.

```
SELECT * FROM customer as cliente
WHERE EXISTS
(
SELECT customerid
FROM invoice as factura
WHERE factura.customerid IS NULL
);
```

Seleccionar todos los clientes si existe por lo menos una facturación hecha en la misma ciudad de donde se registró a algún cliente.

```
SELECT * FROM customer as cliente
WHERE cliente.city = ANY
(
SELECT billingcity
FROM invoice as factura
);
```

3.6. JOINS

3.7. Cross Join

- La reunión cruzada (**cross join**) muestra todas las combinaciones de todos los registros de las tablas combinadas. Para este tipo de join no se incluye una condición de enlace. Se genera el **producto cartesiano** en el que el número de filas del resultado es igual al número de registros de la primera tabla multiplicado por el número de registros de la segunda tabla.

Esta consulta selecciona todos los empleados y clientes que se registraron en la misma ciudad.

```
SELECT DISTINCT
employee.firstname, employee.lastname, customer.firstname,
customer.lastname
FROM customer CROSS JOIN employee
WHERE employee.city = customer.city;
```

La siguiente consulta es equivalente a la anterior:

```
SELECT DISTINCT
employee.firstname, employee.lastname, customer.firstname,
customer.lastname
FROM customer,employee
WHERE employee.city = customer.city;
```

3.8. Inner join/ join

- La reunión interna (Inner Join), determinara cuales filas de las tablas participantes se retornaran mediante un criterio de selección. La clausula **ON** o **USING** es usada para definir el criterio de selección.
- La clausula **ON** es la mas usada para el criterio de selección del **JOIN**. Este utiliza una expresión booleana para retornar verdadero o falso tal cual como en la clausula **WHERE**.
- La clausula **USING** toma una lista de atributos (columnas) separada por comas, la cual usara para seleccionar las filas que se retornaran. Las columnas especificadas deben ser comunes a las tablas participantes. Si los atributos son iguales este retornara la fila.

La siguiente consulta retorna todas las canciones que pertenecen a un álbum de un artista.

```
SELECT track.name as Cancion, album.title as titulo,
artist.name as Artista
FROM track
INNER JOIN album ON track.albumid = album.albumid
INNER JOIN artist ON album.artistid = artist.artistid;
```

La siguiente consulta es equivalente a la anterior:

```
SELECT track.name as Cancion, album.title as titulo,
artist.name as Artista
FROM track
INNER JOIN album USING (albumid)
INNER JOIN artist USING (artistid)
```

Podemos omitir la palabra INNER y utilizar ON y USING en la misma consulta:

```
SELECT DISTINCT employee.firstname, employee.employeeid,
customer.firstname,invoice.invoicedate, album.title,
artist.name
FROM employee
JOIN customer ON employee.employeeid = customer.supportrepid
JOIN invoice USING (customerid)
JOIN invoiceline USING(invoiceid)
JOIN track USING(trackid)
JOIN album USING (albumid)
JOIN artist USING (artistid)
WHERE artist.name = 'AC/DC'
ORDER BY invoicedate;
```

- Que hace la consulta anterior?

3.9. Left outer join/ left join

- En primer lugar, se realiza un inner join. Entonces, para cada fila en T1 que no satisface la condición de unión con cualquier fila en T2, se agrega una fila unida con valores nulos en columnas de T2. Por lo tanto, la tabla unida tiene incondicionalmente al menos una fila para cada fila en T1.

```
SELECT *
FROM artist
LEFT JOIN album USING (artistid)
```

3.10. Right outer join/ Right join

- En primer lugar, se realiza una unión interna. Entonces, para cada fila en T2 que no satisface la condición de unión con cualquier fila en T1, se agrega una fila unida con valores nulos en columnas de T1. Este es el inverso de una combinación izquierda: la tabla de resultados tendrá incondicionalmente una fila para cada fila en T2.

```
  SELECT *
  FROM artist
  RIGHT JOIN album USING (artistid)
```

3.11. Full outer join

- En primer lugar, se realiza una unión interna. Entonces, para cada fila en T1 que no satisface la condición de unión con cualquier fila en T2, se agrega una fila unida con valores nulos en columnas de T2. Además, para cada fila de T2 que no satisface la condición de unión con cualquier fila en T1, se agrega una fila unida con valores nulos en las columnas de T1.

```
SELECT *
FROM artist
FULL OUTER JOIN album USING (artistid)
```

3.12. Natural join

- La reunión Natural es una operación **JOIN** que crea una cláusula de unión implícita basada en las columnas comunes de las dos tablas que se unen. Las columnas comunes son columnas que tienen el mismo nombre en ambas tablas.
- Se debe tener cuidado con este tipo de reunión ya que puede obtenerse un resultado no deseado si no conocen bien todas las columnas de tablas participantes.
- Si las tablas no tienen atributos(columnas comunes), este tipo de reunión se comportara como una reunión cruzada.

Todos artistas que tiene un álbum

```
SELECT *
FROM artist
NATURAL INNER JOIN album;
```

Una reunión mal definida.

```
/* WRONG */
SELECT *
FROM employee
NATURAL JOIN customer
```

Ejemplo de reunion cruzada con Natural Join

```
SELECT *
FROM album
NATURAL JOIN genre;
```

3.13. Insert

- **INSERT** inserta nuevas filas en una tabla.Se puede insertar una o más filas especificadas por expresiones de valor, o cero o más filas resultantes de una consulta.
- Los nombres de las columnas de destino se pueden enumerar en cualquier orden. Si no se da ninguna lista de nombres de columna, el valor predeterminado es todas las columnas de la tabla en su orden declarada; o los primeros N nombres de columna, si sólo hay N columnas proporcionadas por la cláusula VALUES o consulta. Los valores proporcionados por la cláusula VALUES o la consulta están asociados con la lista de columnas explícita o implícita de izquierda a derecha.

```
INSERT INTO tablename VALUES (valuecolumn1,valuecolumn2...)
```

Ejemplo: Insertar un nuevo artista llamado Stromae

```
INSERT INTO artist VALUES (276,'Stromae')
```

La siguiente operación es equivalente a la anterior.

```
  INSERT INTO artist (artistid,name)
  VALUES (276,'Stromae');
```

También podemos agregar varios registros. La siguiente operación registra dos albunes pertenecientes a el artista Stromae.

```
INSERT INTO album (albumid,title,artistid) VALUES
(348, 'Cheese', 276),
(349, 'Racine Carree', 276);
```

3.14. Update

- **UPDATE** cambia los valores de las columnas especificadas en todas las filas que satisfacen la condición. Solamente las columnas a modificar deben ser mencionadas en la cláusula SET; las columnas no modificadas explícitamente conservan sus valores anteriores.
- Hay dos maneras de modificar una tabla utilizando información contenida en otras tablas de la base de datos: utilizando una subconsulta o especificando tablas adicionales en la cláusula FROM. La técnica más apropiada depende de las circunstancias específicas.

```
UPDATE table
SET columna1 = valor1,
columna2 = valor ,...
WHERE
condition;
```

La siguiente consulta modifica el empleado cuyo id es igual a 2.

```
UPDATE employee
SET
firstname = 'Jesus',
lastname ='Perez',
title = 'Database Administrator'
WHERE employee.employeeid = 2;

--select * from employee
```

Podemos utilizar la clausula UPDATE con reuniones de ser necesario.

```
--Actualiza el soporte de todos los clientes que han
    comprado pistas pertenecientes a albunes de AC/DC
UPDATE customer
SET supportrepid = 2
FROM invoice
INNER JOIN invoiceline USING (invoiceid)
INNER JOIN track USING (trackid)
INNER JOIN album USING (albumid)
INNER JOIN artist USING (artistid)
WHERE invoice.customerid = customer.customerid and artist.
    name = 'AC/DC';
```

La siguiente consulta es equivalente a la anterior:

```
UPDATE customer
SET supportrepid = 2
FROM invoice,invoiceline, track,album,artist  --List tables
WHERE   invoice.customerid = customer.customerid
and customer.customerid = invoice.customerid
and   invoice.invoiceid = invoiceline.invoiceid
and   invoiceline.trackid = track.trackid
and   track.albumid = album.albumid
and   album.artistid = artist.artistid
and artist.name = 'AC/DC';
```

También podemos expresar la actualización utilizando una sub-consulta:

```
UPDATE customer
SET supportrepid = 2
WHERE customer.customerid IN(
      select distinct customer.customerid
      FROM employee
      INNER JOIN customer
      ON customer.supportrepid = employee.employeeid
      INNER JOIN invoice USING (customerid)
      INNER JOIN invoiceline USING (invoiceid)
      INNER JOIN track USING (trackid)
      INNER JOIN album USING (albumid)
      INNER JOIN artist USING (artistid)
      WHERE artist.name = 'AC/DC'
);
```

Para verificar la actualizacion puedes usar la siguiente consulta:

```
--Para verficar los resultados puedes usar esta consulta.
select distinct employee.firstname,employee.lastname,
employee.title, customer.supportrepid, customer.customerid,
    artist.name
FROM employee
INNER JOIN customer ON customer.supportrepid = employee.
    employeeid
INNER JOIN invoice USING (customerid)
INNER JOIN invoiceline USING (invoiceid)
INNER JOIN track USING (trackid)
INNER JOIN album USING (albumid)
INNER JOIN artist USING (artistid)
WHERE artist.name = 'AC/DC';
```

3.15. Delete

- La cláusula **DELETE** elimina las filas que satisfacen la cláusula WHERE de la tabla especificada. Si la cláusula WHERE está ausente, el efecto es eliminar todas las filas de la tabla. El resultado es una tabla válida, pero vacía.
- Existen dos maneras de eliminar filas de una tabla utilizando información contenida en otras tablas de la base de datos: mediante sub-selecciones o especificando tablas adicionales en la cláusula USING. La técnica más apropiada depende de las circunstancias específicas.

```
1
2 DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
3 [ USING using_list ]
4 [ WHERE condition | WHERE CURRENT OF cursor_name ]
5 [ RETURNING * | output_expression [ [ AS ] output_name ] [,
      ...] ]
```

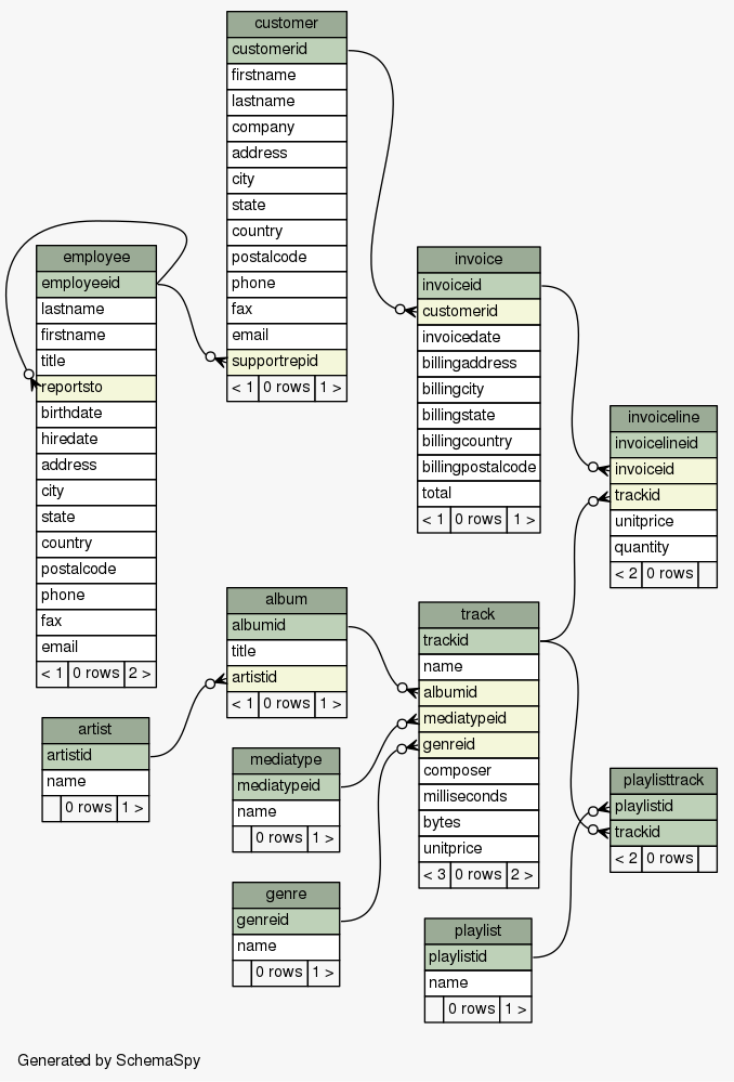


Figura 1: Esquema Entidad Relacion chinook_postgres