



FCEyN UBA

BASES DE DATOS

---

# INTRODUCCION A SQL

---

Autor:

Gerardo ROSSEL



# Contents

<b>1</b>	<b>Introducción y Orígenes</b>	<b>3</b>
<b>2</b>	<b>DDL Definición de datos</b>	<b>3</b>
2.1	Creación de tablas . . . . .	4
2.2	Tipos de datos . . . . .	4
2.3	Eliminación de tablas . . . . .	5
2.4	Modificación de tablas . . . . .	5
<b>3</b>	<b>DML Manipulación de datos - Básico</b>	<b>6</b>
3.1	Select . . . . .	7
3.2	Insert . . . . .	8
3.3	Update . . . . .	9
3.4	Delete . . . . .	10
3.5	Junta básica . . . . .	10
3.6	Alias . . . . .	11
3.7	Orden (ORDER BY) . . . . .	11
3.8	DISTINCT y LIKE . . . . .	12
<b>4</b>	<b>Operador JOIN</b>	<b>13</b>
4.1	Junta Interna (INNER JOIN) . . . . .	13
4.2	Junta Externa (OUTER JOIN) . . . . .	13
4.3	Ejemplos de Junta Externa . . . . .	14
<b>5</b>	<b>El valor NULL</b>	<b>15</b>
<b>6</b>	<b>Agregación y agrupamiento</b>	<b>17</b>
6.1	Funciones de agregación . . . . .	17
6.2	Agrupamiento . . . . .	18
<b>7</b>	<b>Subqueries</b>	<b>20</b>
7.1	Anidamiento multinivel . . . . .	23
<b>8</b>	<b>Operadores UNION, INTERSECT, EXCEPT</b>	<b>23</b>
8.1	UNION . . . . .	23

8.2	INTERSECT . . . . .	24
8.3	EXCEPT . . . . .	24
<b>9</b>	<b>Ejercicios</b>	<b>25</b>

# 1 Introducción y Orígenes

Hasta aquí en la materia hemos vistos los lenguajes formales (álgebra relación y cálculo relacional de tuplas) que permiten establecer consultas sobre una base de datos relacional. La implementación de un lenguaje de consulta que pueda ser ejecutado sobre una computadora llevó a la creación del lenguaje SQL. SQL no sólo permite la consulta de una base de datos sino también la modificación de su estructura, la creación de estructuras y la determinación de reglas de seguridad y de integridad.

SQL fue desarrollado por IBM a principios de los 1970s, en el marco del proyecto denominado System R, con el nombre original de Sequel. Posteriormente el lenguaje fue llamado Structured Query Language o SQL.

En 1986 se transformó en un estándar ANSI (American National Standards Institute) e ISO (International Standards Organization). A partir de allí hubo otros estándares. En 1989 se definen los siguientes tipos de sentencias dentro de SQL:

- Sentencias DDL (Data Definition Language): Permiten crear/modificar/borrar estructuras de datos.
- Sentencias DML (Data Manipulation Language): para manipular datos
- Sentencias DCL (Data Control Language): Contiene aquellas instrucciones para dar y revocar permisos de acceso.

Luego hubo otros estándares en 1992 y posteriormente 1999 (conocido como SQL-99) y extensiones a este último en 2003. Luego en 2005 con la especificación de la forma en que SQL se puede utilizar conjuntamente con XML. Luego se definió el estándar 2008. El SQL es un lenguaje declarativo de alto nivel de tal manera que las consultas se resuelven especificando *que* se quiere obtener en lugar de *cómo* obtenerlo.

## 2 DDL Definición de datos

Comenzaremos por las sentencias que permiten manipular la estructura de la base de datos, es decir que permiten la creación, eliminación y modificación de tablas.

## 2.1 Creación de tablas

La creación de tablas se realiza mediante la sentencia *CREATE TABLE*. La sintaxis de esta sentencia es la siguiente:

```
CREATE TABLE <table_name> (  
  <column_name> <type> [<default value>] [<column constraints>],  
  : : :  
  <column_name> <type> [<default value>] [<column constraints>],  
  <table constraint>,  
  : : :  
  <table constraint>  
);
```

Por ejemplo:

```
CREATE TABLE empleados (  
  enombre char(15) NOT NULL,  
  ecod integer NOT NULL,  
  efnac date,  
  dcod integer  
);
```

Crea la tabla empleados con 4 columnas. La tabla no tendrá ninguna fila, hasta que no se ejecute una sentencia de inserción de datos(*insert*)

## 2.2 Tipos de datos

Hemos visto que en la sentencia *CREATE TABLE* se especifican los tipos de datos para las columnas. Esto establece una restricción porque limita por dominio los datos que pueden almacenarse en cada columna. Los tipos de datos soportados dependen mucho del motor de base de datos que se este utilizando, no todos los motores soportan los mismos tipos de datos. La siguiente tabla 1 lista algunos tipos del estándar de SQL.

Hay un tipo valor especial que SQL utiliza cuando una columna en alguna fila no tiene un valor asignado ya sea porque todavía no se sabe que valor debería ir o porque específicamente se necesita que no tenga valor asignado. Ese valor es el valor **NULL**. Por ejemplo si una columna debería guardar la clave de otra tabla

lógico	BOOLEAN			
character	CHAR	VARCHAR		
numérico	NUMERIC	DECIMAL	INTEGER	SMALLINT
numérico aproximado	FLOAT	REAL	DOUBLE PRECISION	
Fecha y tiempo	DATE	TIME	TIMESTAMP	
intervalo	INTERVAL			

Table 1: Tipos de datos

para relacionarlas y resulta que no hay ninguna fila de la otra tabla relacionada entonces el valor será **NULL**.

## 2.3 Eliminación de tablas

Para eliminar una tabla de la base de datos se utiliza la sentencia *DROP TABLE*

```
DROP TABLE <table_name>
```

Por ejemplo:

```
DROP TABLE empleados;
```

En este ejemplo se borra la tabla empleados, no sólo los datos sino la estructura en si.

## 2.4 Modificación de tablas

La sentencia *ALTER TABLE* es la que nos permite:

- Agregar columnas.
- Cambiar la definición de columnas.
- Agregar o borrar *constraints*.

La sintaxis para agregar una columna es la siguiente

```
ALTER TABLE table ADD (column datatype [DEFAULT expr]);
```

La modificación de columnas tiene algunas variantes según el motor:

- Modificar Columna SQL Server

```
ALTER TABLE table ALTER COLUMN column datatype
```

- Modificar Columna PostgreSQL

```
ALTER TABLE table ALTER COLUMN column TYPE datatype  
ALTER TABLE table ALTER COLUMN column SET NOT NULL;
```

- Modificar Columna Oracle

```
ALTER TABLE table MODIFY column datatype;
```

Ejemplo agregar clave primaria

```
ALTER TABLE tabla ADD CONSTRAINT pktbl PRIMARY KEY( idTabla)
```

Ejemplo agregar una clave foranea

```
ALTER TABLE Hincha ADD CONSTRAINT FK_Hincha_Club FOREIGN KEY(  
    ClubId) REFERENCES Club (ClubId)
```

Ejemplo agregar una constraint

```
ALTER TABLE club ADD CHECK (fechafundacion > '25/01/1900');  
ALTER TABLE club ADD CONSTRAINT fechafundacioncorrecta  
    CHECK (fechafundacion > '25/01/1900')  
    ;
```

### 3 DML Manipulación de datos - Básico

Instrucciones DML: Permiten Manipular (leer y modificar) los datos almacenados en las tablas. Básicamente hay cuatro instrucciones para ello:

- INSERT: Crear nuevas filas en una tabla
- SELECT: Leer filas (o columnas) de tablas.
- UPDATE: Modificar filas existentes en una tabla
- DELETE: Borrar filas de una tabla.

### 3.1 Select

Vamos a describir la sentencia que permite la recuperación de la información desde la base de datos. Una vez creada y con datos la operación mas importante es extraer la información almacenada. Para ello utilizamos la sentencia SELECT cuya sintaxis general es:

```
SELECT [ALL/DISTINCT] select_list
FROM table [table alias] [,...]
[WHERE condition]
[GROUP BY column_list]
[HAVING condition]
[ORDER BY column_name [ASC/DESC] [,...]]
```

Veamos una comparación de un SELECT básico con el álgebra relacional, supongamos que queremos obtener los valores de los atributos de  $a_1$  a  $a_n$  desde las tablas  $t_1$  a  $t_n$  con la condición *cond*. La sentencia SELECT sería algo así como:

```
SELECT a1, ..., an
FROM t1, ..., tn
WHERE <cond>
```

Y en algebra relacional se lee:

$$\pi_{a_1 \dots a_n}(\sigma_{<cond>}(t_1|X|\dots|X|t_n))$$

Es decir a continuación del SELECT se listan los nombres de los atributos a mostrar (que es equivalente a una proyección del álgebra relacional) luego del FROM se listan las tablas desde las que se quiere obtener la información y a continuación del WHERE las condiciones por las cual voy a filtrar los datos (es equivalente a la selección del álgebra relacional). Supongamos que tenemos una tabla de empleados con los atributos (columnas): *ecod*, *enombre* y *dcod* (código, nombre y numero de departamento del empleado) y queremos averiguar los códigos de empleados del departamento número 5. La sentencias equivalentes en álgebra relacional y SQL serían:

```
SELECT ecod, enombre
FROM empleados
WHERE dcod=5;
```

y en algebra relacional:



$\pi_{ecod, enombre}(\sigma_{<dcod=5>}(empleados))$

Más adelante veremos las otras partes de la sentencia SELECT referidas a grupos, orden, etc. La tabla 2 muestra los operadores mas comunes.

Operador	Descripción
=	Igualdad
>, <	Mayor, Menor
>=, <=	Mayor o igual, Menor o igual
<>	No Igual
[NOT] BETWEEN	Entre dos valores (inclusive)
IS [NOT] NULL	Comparación con valores nulos
[NOT] LIKE	Comparación de strings con comodines
[NOT] IN	Pertenencia a una lista de valores
NOT	Negación

Table 2: Operadores básicos

## 3.2 Insert

La instrucción INSERT permite agregar filas en una tabla, es la única sentencia que provee SQL para agregar filas. Existen 2 Formas de ejecutar el INSERT:

1. Usando la cláusula **VALUES** cuya sintaxis es la siguiente:

```
INSERT INTO tabla_nombre [(column [, column...])]
VALUES (value [, value...]);
```

Se crea una nueva fila en la tabla. La nueva fila contendrá los valores determinados por las expresiones en la lista VALUES.

Ejemplos de uso:

```
INSERT INTO empleados
VALUES (1, 'Juan Perez', '04/04/98', 100)
```

```
INSERT INTO deptos (dcod, ddescr)
VALUES (50, 'CONTABILIDAD')
```

```
INSERT INTO deptos DEFAULT VALUES;
```

En el primer caso se insertan en la tabla de *empleados* los valores especificados segun el orden en el que se definieron las columnas en la tabla al momento de su creación. Por lo cual el primer elemento de VALUES es el código de departamento, luego el nombre, la fecha de ingreso, etc. En el segundo caso se especifica a que campos corresponde cada valor, así el 50 es el código de departamento mientras que *CONTABILIDAD* es el nombre del mismo. La última sentencia crea una fila con los valores por defecto en todas las columnas

2. Usando la cláusula **SELECT** (agrega un conjunto de filas mediante un solo INSERT). La sintaxis es:

```
INSERT INTO <tabla_nombre>
    [( <columna>, : : :, <columna>)]
<SELECT statement>;
```

Supongamos que a partir de la tabla de empleados queremos guardar los datos de los gerentes en otra tabla llamada *gerentes*. En dicho caso hacemos lo siguiente

```
INSERT INTO gerentes(gcod, gnombre, gsalario)
SELECT ecod, enombre, esalario
FROM empleados WHERE ecargo = 'GERENTE';
```

La cantidad de columnas y tipos que devuelve el SELECT deben coincidir con la cantidad y tipo de las columnas de la Tabla.

### 3.3 Update

La sentencia UPDATE modifica filas existentes en una tabla. La sintaxis es la siguiente:

```
UPDATE <tabla_nombre> [[AS] <alias>]
SET <columna>=<expression>, : : :, <columna>=<expression>
[WHERE <condicion>]
```

Supongamos que queremos que el empleado con codigo 7782 se incorpore al departamento con codigo 20, entonces deberíamos hacer lo siguiente:

```
UPDATE empleados
SET dcod = 20
WHERE ecod = 7782;
```

Si bien el UPDATE se aplica a sólo una tabla la condición puede afectar a varias tablas.

### 3.4 Delete

La manera de eliminar una o varias filas de una tabla es utilizando la sentencia DELETE:

```
DELETE FROM <tabla_nombre> [[AS] <alias>]
[WHERE <condicion>];
```

Para eliminar el departamento de *finanzas* de la tabla *departamentos*:

```
DELETE FROM departamentos
WHERE ddescr = 'FINANZAS';
```

La sentencia DELETE sin parte WHERE borra todas las filas, pero la tabla permanece creada (sin filas)

### 3.5 Junta básica

Supongamos tres tablas vinculadas como indica la figura 1. En la tabla empleados tenemos la clave foranea *deptoid* que permite relacionarla con la tabla *deptos* y lo mismo ocurre en la clave foranea *provid* en la tabla *deptos* que permite relacionarla con la tabla provincias.

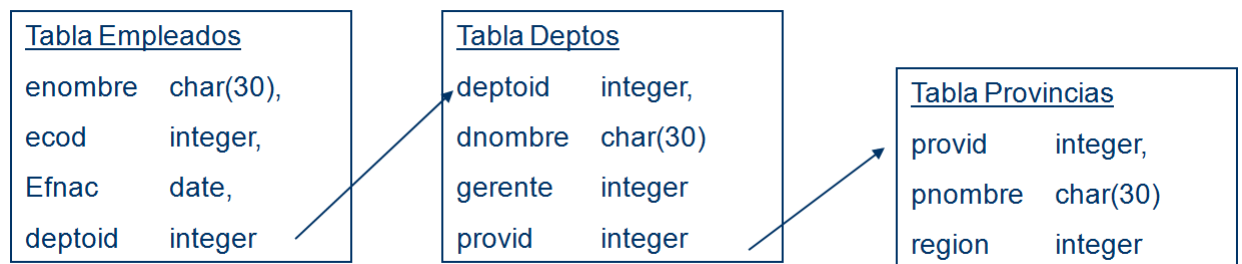


Figure 1: Junta de tres tablas

Si quisiéramos, por ejemplo los nombres de los empleados del departamento de sistema junto a la provincia deberíamos realizar una junta entre las tres tablas como se ve en el siguiente código. En donde se utiliza la clausula WHERE para

igualar las claves foráneas con las claves primarias correspondientes y además filtrar el departamento de nombre *sistemas*.

```
SELECT empleados.nombre, provincias.nombre
FROM empleados, deptos, provincias
WHERE empleados.deptoid = deptos.deptoid
AND deptos.provid = provincias.provid
AND deptos.nombre = 'Sistemas'
```

Si no hubiéramos puesto la condición de junta en el WHERE entonces se realizaría un producto cartesiano entre las tablas del FROM. Las juntas que especificamos son *equijoins* armadas mediante el operador =. También es posible especificar lo que se conoce como *theta joins* utilizando otros operadores.

### 3.6 Alias

En muchos casos para poder simplificar la consulta o en algunos porque es necesario desambiguar nombres como veremos mas adelante, se pueden utilizar *alias*. En la consulta de la sección anterior donde realizábamos una junta podríamos usar alias para no tener que escribir el nombre cada tabla en cada momento.

```
SELECT e.nombre, p.nombre
FROM empleados e, deptos d, provincias p
WHERE e.deptoid = d.deptoid
AND d.provid = p.provid
AND d.nombre = 'Sistemas'
```

En este caso pusimos a *e* como alias de *empleados*, *d* como alias de *deptos* y *p* como alias de *provincias*. Es optativo el uso de la palabra AS indicando por ejemplo: *provincias AS p*

### 3.7 Orden (ORDER BY)

Es posible ordenar las filas que retorna una consulta SQL y para ello se utiliza la clausula ORDER BY , la misma puede ordenar en forma ascendente o descendente siendo la primera la forma por defecto.

Si quisiéramos obtener todos los empleados ordenados por nombre podríamos realzar la siguiente consulta:

Comodín	Utilidad
%	Coincide con cualquier substring que contenga uno o más caracteres
- (guión bajo)	<b>Coincide con un caracter</b>

Table 3: Comodines del LIKE

```
SELECT ecod, nombre, dcod, salario
FROM Empleados
ORDER BY nombre
```

En este caso el orden es ascendente por nombre pero si quisieramos que sea descendentes podríamos agregar el modificador DESC a la clausula ORDER BY

```
SELECT ecod, nombre, dcod, salario
FROM Empleados
ORDER BY nombre DESC
```

### 3.8 DISTINCT y LIKE

SQL no elimina automáticamente las tuplas duplicadas. Para hacerlo se usa la clausula DISTINCT. Por ejemplo para obtener de la tabla de empleados todos los códigos de departamento sin repetición (de esta forma obtendríamos los departamentos que tienen empleados)

```
SELECT DISTINCT dcod
FROM empleados
```

El operador LIKE de SQL provee una capacidad de pattern-matching usando comodines. La tabla 3 muestra los dos comodines con los que se cuenta y su utilidad.

Si quisiéramos obtener los empleados en cuyo nombre haya una H podríamos ejecutar la siguiente consulta.

```
SELECT *
FROM empleados
WHERE nombre LIKE '%H%';
```

## 4 Operador JOIN

En la sección anterior vimos como realizar juntas mediante el operador = , en esta sección vamos a ver otra sintaxis específica que tiene SQL para realizar juntas, se trata del operador JOIN .

### 4.1 Junta Interna (INNER JOIN)

La junta interna o INNER JOIN es equivalente a la junta que vimos anteriormente. Esta junta toma dos tablas y una especificación que describe como dichas tablas deben juntarse. La especificación de la condición de junta se establece a continuación de la palabra reservada ON. Las dos consultas siguientes serían equivalentes

```
SELECT enombre, dnombre
FROM empleados e INNER JOIN deptos d ON d.deptoid=e.deptoid;
```

```
SELECT e.nombre, p.nombre
FROM empleados e, deptos d
WHERE e.deptoid = d.deptoid ;
```

Gráficamente INNER JOIN puede verse con el siguiente diagrama (figura 2) la intersección serian las tuplas que coinciden en los valores de la condición de junta.

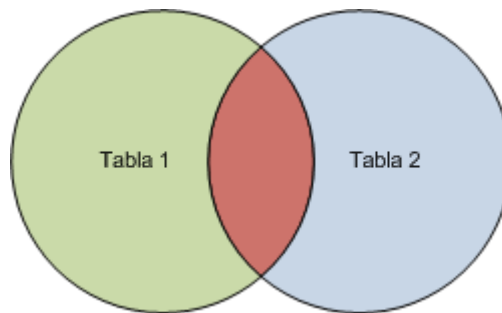


Figure 2: INNER JOIN

### 4.2 Junta Externa (OUTER JOIN)

Hay tres tipos de junta externa u OUTER JOIN, los cuales que describimos brevemente a continuación

**FULL OUTER JOIN** Este tipo de junta retorna:

1. Todas las filas que satisfacen el predicado de junta (como en INNER JOIN)
2. Todas las filas de la primer tabla que NO satisfacen el predicado de junta
3. Todas las filas de la segunda tabla que NO satisfacen el predicado de junta

**LEFT OUTER JOIN** Retorna:

1. Todas las filas que satisfacen el predicado de junta (como en INNER JOIN)
2. Todas las filas de la primer tabla (la de la izquierda de ahí el nombre LEFT) que NO satisfacen el predicado de junta

**RIGHT OUTER JOIN** Retorna:

1. Todas las filas que satisfacen el predicado de junta (como en INNER JOIN)
2. Todas las filas de la segunda tabla (la de la derecha) que NO satisfacen el predicado de junta

La figura 3 refleja mediante diagramas de VENN las diferentes tipos de juntas externas u OUTER JOIN.

### 4.3 Ejemplos de Junta Externa

Veamos algunos ejemplos de junta externa.

Todos los nombres de empleados y si corresponde la descripción del departamento (aunque no tengan departamento):

```
SELECT enombre, ddescr  
FROM empleados e LEFT OUTER JOIN departamento d ON d.dcod = e.dcod
```

Todos los nombres de empleados y departamentos , aunque los departamentos no tengan empleados:

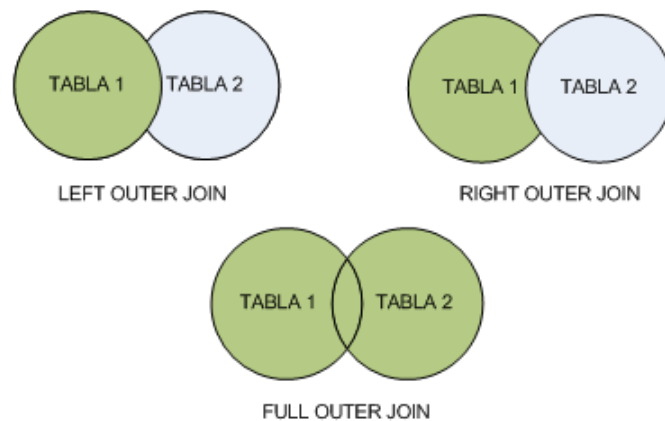


Figure 3: Diferentes modos de OUTER JOIN

```
SELECT enombre, ddescr
FROM empleados e RIGHT OUTER JOIN departamento d ON d.dcod=e.dcod
```

Todos los elementos de la tabla1 que no estén relacionados mediante una columna (digamos columna1) con la tabla2 y todos los de la tabla2 que no estén relacionados con la tabla1 por la misma columna.

```
SELECT * FROM tabla1 t1 FULL OUTER JOIN tabla2 t2
ON t1.Columna1 = t2.Columna1 WHERE t1.Columna1 IS NULL or t2.
Columna1 IS NULL
```

En este caso vemos que se esta haciendo uso del valor *NULL*. para realizar la consulta.

## 5 El valor NULL

Anteriormente mencionamos que SQL provee un tipo de valor específico para una columna no inicializada. Lo mismo ocurre en las juntas cuando no hay un valor en un atributo de alguna de las tablas. Se trata del valor NULL. A no ser que sea explícitamente prohibido como restricción en la tabla NULL es un valor valido para una columna de *cualquier* tipo de dato.

La existencia del NULL genera algunas complicaciones. Si queremos poner una condición cómo *edad > 21* ¿Que debe responder cuando edad es NULL? SQL trata el valor NULL como desconocido y si se debe comparar algo que es



desconocido con cualquier valor (incluso un valor desconocido) el resultado sera también desconocido. Esto nos obliga a una lógica de tres valores. La lógica de tres valores tienen tres posibles resultados para cualquier operación: *true*, *false* o *unknown*( desconocido). ¿Cómo entonces puede saberse si una columna tiene un valor NULL, para ello utilizamos un operador especial: IS NULL y IS NOT NULL.

La lógica de tres valores afecta al AND, al OR y al NOT. Para saber como funciona debemos ver las tablas de verdad de cada uno de ellos.

<b>AND</b>	<b>true</b>	<b>false</b>	<b>unknown</b>
<b>true</b>	true	false	unknown
<b>false</b>	false	false	false
<b>unknown</b>	unknown	false	unknown

Table 4: AND

<b>OR</b>	<b>true</b>	<b>false</b>	<b>unknown</b>
<b>true</b>	true	true	unknown
<b>false</b>	true	false	unknown
<b>unknown</b>	true	unknown	unknown

Table 5: OR

<b>NOT</b>	
<b>true</b>	false
<b>false</b>	true
<b>unknown</b>	unknown

Table 6: NOT

En SQL el WHERE elimina toda fila que NO evalua a *true* en el WHERE (condiciones que evaluan a *false* o *unknown* no califican.) Es diferente a sacar las filas que evalúan *false*.

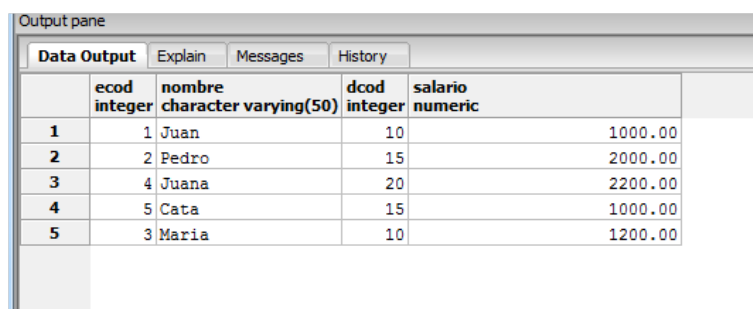
## 6 Agregación y agrupamiento

### 6.1 Funciones de agregación

En muchas situaciones es necesario obtener información que no esta almacenada como tal en la base de datos aunque si esten los datos que permiten obtenerla. Por ejemplo se puede querer saber el salario promedio de un grupo de empleados, el producto más barato o la cantidad de ordenes que realizó un cliente. En la base de datos tenemos las ordenes individuales, el precio de cada producto y los salarios particulares de cada empleado. Es un escenario donde se necesita resumir, sumarizar o realizar la agregación de los resultados. Para ello se utilizan las funciones de agregación. Hay cinco funciones de agregación básicas: COUNT, SUM, MAX, MIN, AVG .Estas funciones operan sobre un grupo de filas y devuelven un resultado. La función SUM suma valores, las funciones MAX y MIN obtienen el máximo y el mínimo respectivamente mientras que AVG permite calcular el promedio. La función COUNT es la función que cuenta filas, dependiendo de si el parámetro es un \* o el nombre de un campo contará todas las filas o sólo aquellas en las que el campo especificado no es *NULL*.

Supongamos que tenemos la siguiente consulta que obtiene datos de una tabla de empleados con el resultado de la figura 4

```
SELECT ecod, nombre, dcod, salario FROM empleados
```



	ecod integer	nombre character varying(50)	dcod integer	salario numeric
1	1	Juan	10	1000.00
2	2	Pedro	15	2000.00
3	4	Juana	20	2200.00
4	5	Cata	15	1000.00
5	3	Maria	10	1200.00

Figure 4: Datos en tabla empleados

Si quisiéramos obtener cuál es el salario mas bajo y cuál es el más alto podemos usar las funciones de agregación MIN y MAX cómo se ve en la siguiente consulta obteniendo el resultado que se muestra en la figura 5

```
SELECT MIN(salario), MAX(salario) FROM empleados
```

	min numeric	max numeric
1	1000.00	2200.00

Figure 5: Salario más alto y más bajo de la tabla empleados

La tabla 7 resume el comportamiento y las características de las funciones básicas de agregación. El uso del DISTINCT se refiere a poder evaluar solo los valores distintos dentro de la función de agregación. Hay que tomar en cuenta que cualquier condición que se ponga en el WHERE se ejecutará antes de la agregación, es decir que la agregación será sobre las filas donde la condición del WHERE evalúa verdadero.

Función	Retorno	Tipo de dato	Ignora Null	Uso de DISTINCT
AVG	Promedio	Numerico	Si	Si
MAX	Máximo	Cualquiera	Si	No
MIN	Minimo	Cualquiera	Si	No
SUM	La suma	Numerico	Si	Si
COUNT	Cantidad de no nulos	Cualquiera	Si	Si
COUNT(*)	Cantidad de filas	Cualquiera	No	Ilegal

Table 7: Funciones de agregación

## 6.2 Agrupamiento

Cómo decíamos las funciones de agregación trabajan sobre grupos de filas. En el ejemplo del salario máximo y mínimo el grupo estaba compuesto por todas las filas de la tabla. Obviamente podríamos aplicar una condición para filtrar algunas filas y obtener valores sobre ellas. En muchas ocasiones se necesita obtener valores para diversos grupos de filas, por ejemplo podríamos querer saber el salario mínimo y máximo para cada departamento. Si bien podríamos hacer una consulta por departamento esto no es práctico ( de hecho primero deberíamos saber cuantos

departamentos hay y cuales son) . SQL cuenta con la clausula GROUP BY que permite especificar por que atributos se quiere agrupar y agregar resultados. Se ve mas claro con un ejemplo, supongamos el caso que mencionamos de salarios máximos y mínimos por departamento, la consulta quedaría

```
SELECT dcod, MIN(salario), MAX(salario)
FROM empleados GROUP BY dcod
```

El resultado del agrupamiento se ve en la figura 6

Data Output	Explain	Messages	History
	dcod integer	min numeric	max numeric
1	20	2200.00	2200.00
2	15	1000.00	2000.00
3	10	1000.00	1200.00

Figure 6: Salario más alto y más bajo por departamento

El funcionamiento del GROUP BY en el ejemplo anterior ocurre de la siguiente manera: la tabla de empleados se divide en grupos tomando como base el código de departamento, es decir un grupo por cada código diferente. Para cada uno de los grupos se aplica el SELECT con la función de agregación. De esta manera se obtiene *una fila por cada grupo* con la agregación aplicada al grupo. Si hubiera algún código de departamento con valor *NULL* se crearía un grupo para ese valor.

Se puede agrupar por varios atributos en cuyo caso se arma un grupo por cada conjunto de valores iguales de los atributos por los que se agrupa. Los atributos por los cuales se agrupa pueden incluirse en la proyección (es decir en la lista de atributos del SELECT) pero sólo pueden incluirse esos atributos, constantes o funciones de agregación ninguna otra cosa.

**Si en SELECT hay funciones de agregación, entonces solo puedo proyectar columnas que estén en el GROUP BY o constantes**

Si se especifica una condición en la clausula WHERE entonces las filas serán removidas **antes de agrupar** y no serán tomadas en cuenta. También puede desearse no devolver todos los grupos sino aquellos grupos que cumplan con alguna condición. Para lograrlo se utiliza la clausula HAVING

Supongamos que queremos obtener el salario promedio por departamento, junto a la cantidad de empleados en ese departamento pero solo para aquellos departamentos que tengan más de 10 empleados. Es un caso claro donde debemos eliminar del resultado grupos, aquellos que tengan mas de 10 filas. Usando la clausula HAVING tendríamos la siguiente consulta:

```
SELECT dcod, COUNT(*) , AVG(salario)
FROM empleados
GROUP BY dcod
HAVING COUNT(*) > 10  -- cond/restric sobre el grupo
```

Usar funciones de agrupación en el HAVING es válido ya que ellas trabajaría por cada grupo. En el ejemplo la tabla se divide en grupos, uno por cada dcod, luego se sacan los grupos cuya cantidad de filas sea mayor a 10 y finalmente se proyectan el código de departamento, la cantidad y el salario promedio. Es importante tener en claro la diferencia entre el WHERE y el HAVING, la primera actuá **sobre filas** mientras que el HAVING trabaja *sobre grupos*, por ello no puede haber funciones de agregación en el WHERE y si en el HAVING .

## 7 Subqueries

Los *subqueries* o *select anidados* son una poderosa herramienta para resolver consultas complejas. Veamos el siguiente caso: tenemos una tabla con los datos de los empleados y el salario de cada uno, si quisiéramos obtener los nombres de los empleados que cobran el salario mínimo deberíamos comparar el salario de cada empleado con el mínimo salario de la misma tabla. La utilización de *subqueries* permite resolver este problema en forma sencilla y elegante.

```
SELECT nombre, salario
FROM empleados
WHERE salario = (SELECT MIN(salario)
                 FROM empleados);
```

En el ejemplo mostrado SQL comienza ejecutando la consulta interior (*inner query*) y los resultados son utilizados por la consulta exterior (*outer query*). Los parentesis sirven para definir los límites de la *subquery*. Se debe tomar en cuenta que:

- Sólo las columnas de la consulta exterior pueden aparecer en el resultado.
- Las *subqueries* tienen restricciones sobre lo que pueden devolver dependiendo del operador utilizado para vincular. En el ejemplo se utilizó el = por lo que la *subquery* debe retornar sólo un valor para ser comparado. Es el programador el que debe garantizar que la *subquery* retorne los valores adecuados en cantidad y tipo (salvo conversión implícita que realice el SQL).

No sólo en el WHERE se pueden anidar *subqueries* también puede hacerse en el HAVING. Supongamos que queremos saber el departamento y salario mínimo de dicho departamento tal que ese salario mínimo sea mayor al salario promedio de los empleados. Es decir los departamentos en los cuales todos los empleados ganan más que el salario promedio. Para ellos hay que agrupar por departamento para obtener el mínimo de cada grupo y además hay que filtrar por ese mínimo para que sea mayor al promedio de todos los empleados, es lo que se ve en la siguiente consulta.

```
SELECT dcod, MIN(salario)
FROM empleados
GROUP BY dcod
HAVING MIN(salario) > (SELECT AVG(salario)
                        FROM empleados);
```

La tabla 8 muestra otros operadores especiales que son usados en *subqueries*.

Operador	Descripción
IN	Retorna TRUE si está incluido en los valores retornados por el subquery
ANY	Retorna TRUE si la comparación es TRUE para al menos un valor retornado por el subquery
ALL	Retorna TRUE si la comparación es TRUE para todos los valores retornados por el subquery
EXISTS	Retorna TRUE si el subquery devuelve al menos una fila. FALSE si devuelve 0 filas

Table 8: Operadores para Subqueries

Veamos la siguiente consulta:

```

SELECT dcod, ddescr
FROM departamentos d
WHERE NOT EXISTS (SELECT *
                  FROM empleados e
                  WHERE d.dcod = e.dcod);

```

Lo interesante de la consulta es que se utiliza un valor de la consulta exterior para restringir la consulta interior (esto en general obliga a usar alias para evitar ambigüedades). **No puede hacerse a la inversa.** Por cada tupla candidata a devolver el SQL realiza la consulta interior usando el valor de la exterior. En este caso por cada departamento utiliza el código (d.cod) para restringir la consulta interior o *inner query* a aquellos empleados que tengan ese código. Como el WHERE tiene como condición NOT EXISTS sólo permitirá departamentos para los cuales la *subquery* no devuelva ninguna fila dejando aquellas tuplas tal que no haya un empleado con ese código, es decir los departamentos que no tengan empleados. Las consultas anidadas o subqueries pueden usarse en también en sentencias UPDATE o DELETE como se ve en los siguientes ejemplos:

```

UPDATE empleados
SET (cargo, dcod) = (SELECT cargo, dcod
                    FROM empleados
                    WHERE ecod = 7499)
WHERE ecod = 7698;

```

```

DELETE FROM empleados
WHERE dcod =
      (SELECT dcod
       FROM departamentos
       WHERE ddescr = 'VENTAS');

```

Otra forma de usar *subqueries* es en el FROM, de forma que se toma como si fuera una tabla o vista.

```

SELECT salario
FROM (SELECT salario, egeren, dcod
      FROM empleados
      WHERE geren IS NOT NULL)
WHERE dcod = 7698;

```

En la cátedra se recomienda, si es posible, no usarlo , en los parciales se pide a veces expresamente no usarlos. Pero hay algunos ejercicios de la práctica que sólo

pueden resolverse usando este método. Supongamos que una *subquery* retorna un resultado vacío y veamos que ocurre.

```
SELECT nombre FROM empleados
WHERE dcod = (SELECT dcod
              FROM departamentos
              WHERE ddescr = 'No Existe');
```

La *subquery* retorna un resultado vacío porque no hay departamentos con esa descripción, por cual retorna NULL. Por cada empleado la consulta exterior compara el código de departamento, dcod, con NULL. Esta comparación retorna *unknown* por cada empleado por lo que la consulta exterior retornara una tabla vacía.

## 7.1 Anidamiento multinivel

Es posible realizar más de un nivel de anidamiento en subqueries, por ejemplo si queremos actualizar el salario de los empleados de los departamentos 1020 y 1040, sumandole el ultimo premio asignado podríamos realizar la siguiente consulta:

```
UPDATE empleados e
SET e.salario = (SELECT e.salario + p1.premio
                FROM   premios p1
                WHERE  p1.ecod = e.ecod
                AND    p1.fecha_premio =
                    (SELECT MAX(p2.fecha_premio)
                     FROM   premios p2
                     WHERE  e.ecod=p2.ecod) )
WHERE dcod IN ('1020', '1040')
```

## 8 Operadores UNION, INTERSECT, EXCEPT

Es posible combinar varias consultas y aplicar operadores similares a los de conjuntos (aunque al permitir duplicados no son estrictamente conjuntos sin *bags*) como unión, intersección y resta.

### 8.1 UNION

Dadas dos consultas se pueden combinar retorna las filas pertenecientes a ambas consultas con las siguiente sintaxis



`<izq SELECT> UNION [{ALL | DISTINCT}] <der SELECT>`

Por ejemplo si queremos obtener los empleados actuales y los históricos podríamos realizar la siguiente consulta:

```
SELECT nombre, cargo
FROM empleados
UNION ALL
SELECT enombre, efuncion
FROM emp_hist;
```

La restricción que se debe respetar es que ambas consultas sean compatibles. Dos consultas son compatibles si tienen el mismo número de atributos y cada uno es compatible con el correspondiente de la otra. Dos atributos son compatibles si SQL puede implícitamente realizar un *cast* al mismo tipo. Por defecto el operador UNION elimina los duplicados considerando al NULL como un valor simple. Si se quieren mantener los duplicados se debe usar UNION ALL.

## 8.2 INTERSECT

El operador INTERSECT es equivalente a la intersección de conjuntos y devuelve los elementos comunes.

`<izq SELECT> INTERSECT [{ALL | DISTINCT}] <der SELECT>`

Cómo ejemplo podemos obtener los empleados actuales y que se encuentran también en el histórico.

```
SELECT nombre, cargo
FROM empleados
INTERSECT
SELECT enombre, efuncion
FROM emp_hist;
```

El operador ALL permite mantener los duplicados. Si un valor esta duplicado en ambas tablas ( $i$  veces en una y  $d$  veces en otra) la cantidad de duplicados del resultado será la cantidad mínima entre  $i$  y  $d$ .

## 8.3 EXCEPT

El operador EXCEPT realiza la diferencia entre conjuntos. Si se tienen dos tablas I y D entonces el resultado serán todas las tuplas de I que no estén en D. La

sintaxis es:

```
<izq SELECT> EXCEPT [{ALL | DISTINCT}] <der SELECT>
```

Si queremos los empleados que no hayan sido pasados al histórico de empedados podríamos hacer:

```
SELECT nombre, cargo
FROM empleados
EXCEPT
SELECT enombre, efuncion
FROM emp_hist;
```

Del mismo modo que UNION o INTERSECT el operador EXCEPT elimina los duplicados. Si se quiere mantener los duplicados se debe utilizar EXCEPT ALL. Si la tabla de la izquierda tiene  $i$  veces un valor y la tabla de la derecha  $d$  veces, la cantidad de duplicados es el mínimo entre  $i - d$  y 0.

## 9 Ejercicios

1. Dado el siguiente esquema:

Fragancia(idFragancia, nombre, idTipo, precio)  
TipoFragancia(idTipo, nombre)  
MateriaPrima(idMateriaPrima, nombre, idProveedor)  
Proveedor(idProveedor, nombre)  
CompuestoCon( idFragancia, idMateriaPrima, cantidad)

Realizar en SQL una consulta para obtener los nombres de los proveedores de más de dos materias primas que sean utilizadas en aquellas fragancias cuyo precio es más caro que el de todas las fragancias tipo “Florales”.

2. Dado el siguiente esquema:

Club( idClub, nombre)  
Jugador(idJugador, nombre, posicion, idClub)  
Partido(idPartido, idClubLocal, idClubVisitante, fecha, resultado)  
Estadisticas(idPartido, idJugador, idInfraccion )

Realizar en SQL:

- a. Una consulta que devuelva la cantidad de partidos jugados de local por jugador sin contar los partidos donde el jugador cometió falta. Es decir deberán devolver el nombre del jugador seguido por el numero de partidos en los cuales no cometió falta.
- b. Los clubes conflictivos. Un club es conflictivo cuando promediando la cantidad de infracciones por partido, esta es mayor a 3. Es decir, los nombres de los clubes que tengan más de 3 infracciones por partido, promediando todo los partidos jugados.

## References

- [1] *SQL Practical Guide for Developers*. Michael J. Donahoo y Gregory D. Speegle. ELSEVIER (2005)
- [2] *SQL Antipatterns - Avoiding the Pitfalls of Database Programming* Bill Karwin. The Pragmatic Bookshelf (2010)
- [3] *SQL and Relational Theory - How to Write Accurate SQL Code* C.J.Date. Second Edition. O'REILLY (2012)