# **Introducción a Consultas a bases de Datos**

## ***Sobre la importancia de las consultas (o Queries) en una base de datos relacional.***

Las consultas son la parte fundamental de una base de datos. De qué te sirve hacer una consulta o un Query efectivo?

Pues bien, una consulta bien hecha podría incluso *salvar un negocio o una empresa.* Las empresas tienen una gran necesidad por saber la información correcta, exacta, oportuna; dicha información, se puede extraer generalmente de las bases de datos que ya tenemos (de aquí la importancia de las *consultas*).

En las bases de datos tenemos muchas veces *clientes, productos, proveedores* y diferentes tipos de entidades que, siendo tratadas de forma aislada, no nos revelarían mucho (serían datos sin procesar); sin embargo, cuando estos datos se comienzan a unir a través de *Queries* se podría generar conocimiento, éste se puede reflejar por medio de informes o todo tipo de herramientas que faciliten la generación de conclusiones y visualizaciones; de tal manera que, las personas que estén encargadas de *maniobrar* la empresa para un lado, o para el otro, tengan la información necesaria para tomar las decisiones eficientes y acertadas basándose en el análisis de datos.

Entonces, alrededor de toda esa cultura de “*extraer datos y transformarlo en información”,* se han creado varias especialidades en el estudio de los datos que tiene que ver con, por ejemplo, ***ETL*** (sobre la transformación de datos), ***Business Intelligent*** (sobre la extracción & utilización de datos para la toma correcta de decisiones) e, incluso, tiene que ver con: ***Machine learning*** (sobre la toma de los datos existentes de una base de datos para transformarlos de tal manera que asuman una identidad propia; pueda revelarnos algo, sea información coherente y, sobre todo, útil y práctica).

*En resumen, los Queries son una parte fundamental de una base de datos;* pues, nos permiten conocer las estructuras de los **Query** y resolver cualquier duda que tenga cualquier miembro de la empresa, que pueda ser resuelta, por medio del manejo y la manipulación de una base de datos.

En resumen, con los ***Queries*** es que se brinda valor a una organización al final del día, pues, es lo que le da la información a la empresa; con la cual, se toman decisiones de gestión e inversión diariamente. Entonces, para traer toda esa información de nuestro interés, de una base de datos, necesitas hacer ***Queries.*** Los ***Queries*** son la forma en la que estructuramos las preguntas que le vamos a hacer a la base de datos; es decir, al final del día, nuestro camino a seguir para encontrar todas las respuestas.

Ahora, lo complicado es que hay muchas formas de hacer un ***Query,*** desde el más básico al más complejo.

# ***Vamos de las Queries básicas a las complejas: Select***

Empecemos a practicar la habilidad de convertir nuestras preguntas en ***Queries.*** El ***Query*** tiene básicamente dos partes: la sentencia ***Select*** y la sentencia ***From*** y, en varias ocasiones, nos encontramos con la sentencia ***Where.***



***Select*** nos permite traer los datos que queremos mostrar de alguna ***columna*** en especifica o de todas ***(\*)***; en este caso puntual, queremos traer los datos de la columna o atributo ***city*** para contar la totalidad de sus registros con la función ***count(\*),*** al resultado de este conteo se le nombrará ***total***,pues, así fue definido con la propiedad **AS.**

*Un dato:* En caso que usted no sólo desee ver los datos de una única columna, de una tabla, puede agregar más. Simplemente, dentro de la misma sentencia ***Select,*** debe llamar al otro u otros campos en cuestión por medio del uso de comas; por ejemplo: ***SELECT city, name*** (Ahora veríamos, además de ***city,*** la columna ***name).***

En ***From*** indicamos de dónde vamos a seleccionar esas columnas que queremos mostrar con el ***select;*** es decir, más precisamente, de qué ***tabla***. En nuestro ejemplo estamos indicando que queremos ver los datos de la columna o atributo ***city*** de la tabla ***people*** *(a veces se deja expresada también la base de datos de dónde viene la tabla; por ejemplo, en el caso de la tabla people, se vería algo así:* ***FROM platziblog.people = FROM people).***

Ahora, con ***Where,*** el filtro de la consulta y recolección de datos se vuelve más exigente; pues,lo que se establece ahí es una condición; en consecuencia, dicha condición debe cumplirse para poder seleccionar los datos deseados de una columna para una tabla en especifica. En contexto, cuando definimos un *Where* con ***active = true*** le estamos diciendo a nuestra base de datos de trabajo actual que deseamos solamente los datos de la columna ***city*** de la tabla ***people*** que cumplan con la condición ***active = true****; es decir, donde la columna* ***active sea*** *igual a* ***true.*** Lo dicho, ***active*** vendría siendo también otra columna o atributo de la misma tabla ***people*** y ***true*** sería uno de los posibles valores que pudiese tener la columna ***active.***

***Group by*** lo que nos permite es agrupar los datos, justamente, según un criterio pasado; en este caso, el criterio pasado fue: ***city;*** es decir, vamos a agrupar los datos de la tabla ***people,*** según la columna ***city***, que cumplen con la condición ***active = true***. Ahora, en contexto, la orden fue *contar* todos los registros de *people* &***Group by*** lo que hará es contarlos por ciudad.

***Order by*** define de qué manera se organizarán los datos consultados, de forma ascendente o descendente; puntualmente en este caso nos ayuda a definir de qué manera ordenar los datos seleccionados de la columna ***city***, de la tabla ***people***, que cumplen con la condición ***active = true*** (la totalidad de ellos).

Por último, en ***Having,*** sedefine otro parámetro de limitación, va a aplicar otro filtro o condición mucho más específico que ***Where*** (Realmente es como un ***Where***, pero funciona con registros agrupados); en este caso, se va a filtrar (elegir) el ***total*** de los datos seleccionados de la columna ***city***, de la tabla ***people*** y que tengan un ***active = true;*** pero, además, *cuando haya 2 o más de 2 personas en cada ciudad* ***(total >= 2).***

***un dato:*** La sentencia ***Select \**** significa *selecciona todo;* sin embargo, qué es *todo?* Es decir, **de dónde sale el *todo*?** SQL interpreta esta solicitud, por sí sola, como muy ambigua y tiene razón (por lo cual te lanza un error). Es decir, a ***Select \**** se le debe acompañar en todos los casos con otra sentencia. Por ejemplo, evaluemos nuestro primer caso de ***Query:***



En efecto, con esto estamos diciéndole a la base de datos que nos traiga absolutamente todas las columnas, con todos sus registros, de ***posts*** ***(SELECT \* FROM posts),*** que nos muestre absolutamente todo de ahí (sus columnas con sus registros). De hecho, generalmente, te encuentras las ***Queries*** con una tercera sentencia declarada que es ***Where;*** que nos permite filtrar cuáles de esos datos, de la tabla ***posts***, queremos ver.



En este caso, sólo hemos filtrado los datos de la tabla ***posts*** que tienen como fecha de publicación una fecha igual o superior al año ***2024.*** Es decir, la base de datos nos arrojará todos los registros de ***posts*** que hayan sido publicados después o durante el año ***2024***.

***Nota: para SQL “>” es igual a “>=”, por lo que en el caso anterior se incluye también al año 2024. Sin embargo, “<” no es lo mismo que “<=”.* En resumen, con *Queries* y *Select,* filtramos información.**

En adelante, probaremos ***Queries*** más complejas según el nivel de complejidad que tenga, propiamente, la consulta de los datos que se desee ver; que, obviamente, deberían de ser en la medida unos datos con unas especificaciones más puntuales. Sin embargo, para hacer consultas, se requiere tener primeramente datos; dicho eso, empecemos por agregar datos de prueba a nuestras tablas de modelación.

# ***Propiedad AS***

Ahora, existe una propiedad que no es propiamente de ***Select,*** pero que sirve de asistencia a las consultas o ***Queries*** que hagamos con ***Select;*** esa propiedad es: ***AS.***

***AS*** nos permite pasarle un *alias,* o nombre temporal *que no sustituye de manera permanente al nombre original,* a una o varias columnas (o incluso a tablas). Es decir, ***AS*** nos permite sustituir temporalmente, por lo menos en el justo momento de hacer propiamente una consulta, el nombre original de una o de varias columnas o tablas; de tal manera que se le asigne un nombre temporal a la columna (o tabla) de nuestro interés.

El proceso es más o menos así: por medio o no de ***Select,*** se llama a la *o a las columnas o tablas* de nuestro interés con su nombre original **(Si las llama sobre la misma línea de código, no se olvide hacer separaciones con coma por cada columna o tabla llamada);**posteriormente, se le debe pasar a cada nombre por separado, e inmediatamente después del nombre de cada columna o tabla llamada, la sentencia ***AS;*** entonces, luego de declarar ***AS,*** se le pasaría un nuevo nombre (que sería el *Alias*), el de nuestro interés, a cada columna o tabla llamada en cuestión,... tal que así, por ejemplo:



Advertir que lo único que cambia del *dataset,* de forma *temporal,* es el nombre original al hacer una consulta con ***Select;*** del resto, todo sigue igual, los mismos datos para las mismas columnas o tablas: sólo que ahora está viendo que esas columnas o tablas afectadas están siendo llamadas con otro nombre; los cuales, fueron pasados con la propiedad ***AS.***

## **Ahora, algo interesante.**

**En adelante, de hecho, usted podrá llamar cualquiera de las dos formas a sus columnas de interés para manipular sobre ellas, ya sea que las llame con su nombre original o por medio del *Alias* recientemente pasado.** Por ejemplo, en futuras ***Queries*** más complejas puedes referirte a los campos en cuestión, ya no con su nombre original; sino, también, con su *Alias* asignado (si así lo deseas).

**Ahora, ponga atención a lo siguiente:** **AS**, como recurso, no siempre se usa como propiedad que asiste a la sentencia *SELECT* para definir un *alias* posterior a su declaración*,* No. A veces, y en los casos puntuales en los que se crean *VIEWS,* el alias o nombre que recibirá la vista en cuestión, de hecho, se define de primero y antes de hacer uso del recurso de *AS* (antes de declarar AS)*.* Ahora sí,justo después de declarar AS, en VIEWS, no se establece un *alias*, NO; si no que de entrada se establece qué información deseamos reflejar y filtrar dentro de la View en ejecución. ***Tal como lo hicimos en este ejemplo, observe:***



**En este *View* se pretende reflejar la información de la gente que viene de Brasil, puntualmente, la información de su nombre y de su contacto.**

El anterior ejemplo lo estuvimos viendo en el documento: ***1. SQL DDL,***explicado entrelas páginas 6 & 7.

# ***Count e introducción a funciones de Agregación***

Con ***Select + función de Agregación,*** también, se nos permite extraer datos de manera temporal que nos muestre información adicional; un buen ejemplo de eso sería combinar el ***Select*** con el uso de la función ***Count(),*** ¿y qué hace ***Count()***?

La función ***Count()*** se encarga de contar, ¿contar qué? Contar ***Registros.*** Es decir, a partir de unos parámetros pasados para ***Count(),*** éste últimose encargará de contar la totalidad de los registros que hay dentro de una tabla, si y sólo si, se cumplen a cabalidad unas condiciones o parámetros pasados ¿qué parámetros? Pues, en principio, los que establecemos dentro de los paréntesis de ***Count().***

El ejemplo más básico de uso de la función ***Count(),*** en connivencia con la sentencia ***Select,...*** es que en ***Count()*** se le pase, como parámetro, que cuente absolutamente todos los registros existentes para una tabla seleccionada con la sentencia ***From*** (sin filtro alguno); entonces, en este caso, ese nuevo dato que se extraería temporalmente y que nos podría agregar más información sería el dato numérico, que se nos arroja, equivalente a la totalidad de registros que se alcanzaron a contar para una tabla seleccionada en cuestión.

Pongamos, por ejemplo, el siguiente caso: 

Esto nos arrojaría un nuevo dato, un simple número, que reflejaría el conteo total de los *registros* que tenemos en nuestra tabla *posts;* es decir, prácticamente, la cantidad de *posts* que tenemos publicados. Tal que así:

***, 22*** sería el número de *registros,* o posts, que tenemos en nuestra tabla *posts* parcialmente.

Ahora, si se da cuenta, ese nuevo dato *agregado*, el dato numérico resultante; está bajo la posesión de un nuevo atributo, campo o columna creado; esa nueva columna si se percata se llama: ***COUNT(\*)***

Luego se dará cuenta que a la nueva columna resultante, ***COUNT(\*)***, se le puede pasar un ***Alias*** tambiéncon la sentencia ***AS;*** esto para recordar o renombrar a dicha columna de una manera más fácil e *identificativa*; pues, es *engorroso* tener que llamar a una columna con una palabra clave como ***Count(\*****).* Probemos:



Esto quiere decir que nuestra columna inicialmente llamada ***COUNT(\*)*** ahora puede ser llamada también como: “**numero\_posts”**; lo cual, tendría mucho más sentido ya que ***numero\_posts***, como nueva columna, nos estaría mostrando justamente el número de postsque tiene la tabla ***posts***,... Quedando así:



Por supuesto, hay que decir que realmente este dato no está almacenado de forma permanente en la base de datos. Con todo y eso, lo podemos aún así consultar por medio de ***Select.***

***Para casos de negocio, COUNT() es una función que generalmente necesita agruparse (GROUP BY);*** *y, de hecho, pasa lo mismo con todas las demás funciones de Agregación:* **Sum(), Count(), Avg(), Max(), Min()***.* Prácticamente todas reciben el mismo tratamiento, de forma, que se le da a *COUNT()*; por lo que a partir de COUNT(), trataremos de dejar claro cómo se deben tratar las demás funciones de agregación.

Dicho lo anterior, siempre que trabajen con funciones como: **Sum(), Count(), Avg(), Max(), Min(), Stddev()…** deben llamar la sentencia **GROUP BY** cuando en el ***SELECT*** se incluyan otras columnas con la cual relacionar dicha función de agregación llamada; ya sea **Sum(), Count(), Avg(), Max(), Min(), Stddev(), …** ***Group By*** lo que hará es, según el campo que se le asigne, mantener la ejecución de la función de agregación; pero no como un todo, no se ejecutará la función de forma completamente *agregada* sin discriminar los datos*,* sino sobre cada una de las *categorías* que hayan dentro del campo que ha sido pasado dentro de la sentencia ***Group By,*** distinguiendo la *Agregación* por categorías, por los datos que hayan para cada una de las categorías.

***En el siguiente apartado profundizaremos un poco más sobre cada una de las funciones de Agregación.***

# ***Funciones matemáticas o de agregación***

Las funciones de agregación en SQL nos permiten efectuar operaciones matemáticas sobre un conjunto de registros evaluados, o seleccionados para ser evaluados, dentro de un campo o columna (generalmente de tipo numérica) determinada. Al efectuar este tipo de operaciones matemáticas se devuelve un único valor agregado, como resultado, en el análisis de los valores evaluados o fijados para evaluar. *Tome como referencia la propiedad* ***Count*** *previamente recordada.* Sin embargo, a lo sumo, podríamos incluso calcular ***medias, máximos, sumas***,… sobre dicho conjunto de valores determinados; estos valores se determinan por condiciones o parámetros pasados que tienen mucho que ver con fijar qué columna(s) y/o fila(s) será(n) evaluada(s).

*Las funciones de agregación básicas que soportan todos los gestores de datos son las siguientes:*

* **COUNT**: devuelve el número total de filas seleccionadas por la consulta, como particularidad se puede usar COUNT()\* donde contará todos los registros de la tabla, incluyendo nulos.
* **MIN**: devuelve el valor mínimo del campo que especifiquemos.
* **MAX**: devuelve el valor máximo del campo que especifiquemos.
* **SUM**: suma los valores del campo que especifiquemos. Sólo se puede utilizar en columnas numéricas.
* **AVG**: devuelve el valor promedio del campo que especifiquemos. Sólo se puede utilizar en columnas numéricas.

***Modelación***



# ***Función IF()***

Estructura de control que se encarga de evaluar la validez lógica de una expresión lógica que se define por medio de operadores relacionales. Según el resultado lógico que arroje la condición evaluada, ya sea verdadera o falsa, se mostraría por pantalla lo que fue definido como salida verdadera o, en el caso contrario, como salida falsa, respectivamente. Ambas salidas, verdadera y falsa, se separan por comas dentro de la sintaxis.

IF (expresión lógica, resultado/salida\_true -de ser la expresión verdadera-, resultado/salida\_false –de ser la expresión falsa-).

# ***Función CASE()***

Es también una estructura de control, en principio mutuamente excluyente, que sirve para evaluar la validez lógica de una lista de expresiones y/o condicionales (no sólo de una). En caso de que alguna expresión lógica de este listado sea valida (y sólo una debe ser válida por ser mutuamente excluyentes) se retornaría, por pantalla, su salida o valor definido como verdadero.

La sintaxis comienza con la sentencia CASE, luego se evalúa la validez lógica de cada una de las expresiones valiéndose de la propiedad WHEN (una para cada expresión); y, en caso de encontrar valida alguna expresión lógica, dicha expresión devolverá por pantalla la salida especificada como verdadera para su propia condición evaluada (lo que se refleja por salida, como verdadero, es lo que se pone justo después de la propiedad THEN).

La clausula ELSE, que complementaría a CASE, es opcional. ELSE devolverá su salida o valor definido como verdadero en caso que todas las condiciones WHEN, previas, hayan sido falsas. Ahora, si todas las condiciones son falsas, pero tampoco existe la clausula ELSE, se devolverá un valor de tipo NULL.

La sentencia CASE es finalizada con la palabra clave END. Caso aparte, a las sentencias CASE también se les puede asignar un *alias*. De hecho, al ejemplo a continuación se le asignó uno **y es lo ideal**.

## ***Modelación***



# ***Función CONCAT()***

La función ***CONCAT*** nos permite concatenar (unir) 2 o más de 2 cadenas de caracteres. Si estamos interactuando con columnas, es común ver que se use esta función como recurso para unir los textos o cadenas de caracteres de 2 o más de 2 columnas, contiguas o no. Para unir 2 o más de 2 cadenas de caracteres; es decir, 2 o más de 2 columnas, se llaman las columnas en cuestión dentro de la función **CONCAT**()*,* entre sus paréntesis, tal que así: ***CONCAT(columnaA, columnaB).*** De la anterior forma estaríamos conectando las cadenas de caracteres, de cada columna, al poder vincularlas por medio de *comas*, las une sin guardar espacio alguno entre ellas; es decir, como si formarán una sola palabra entre sí. Si queremos guardar un espacio, por ejemplo, entra las cadenas de caracteres de cada columna (al ser concatenadas) podemos asistirnos de *comas & comillas…* Ya lo verá en la práctica. De hecho, es por medio de *comas & comillas* que podemos agregar texto adicional a nuestro propio gusto.

Por ejemplo, el siguiente *Query* sencillo de ser ejecutado nos mostraría por pantalla el nombre de una persona (un autor) para una columna A (Author) &, contiguamente, se mostraría la nacionalidad (de dicho autor) para una columna B (nationality), veamos:



*Resultado/Salida:*



Si en caso tal deseamos mostrar la información de ambas columnas en una sola (para disminuir el uso de columnas y/o presentar la información de manera más concisa) podríamos hacerlo justamente con la función *CONCAT.* Por ejemplo, concatenemos los textos de cada columna, **pero** asistiéndonos de un *guion y de un uso adecuado de espacios* que separen y le den oxigenación a los textos concatenados. Por ejemplo, que se vea de la siguiente forma el primer registro, de nuestra columna resultante, al hacer la concatenación de columnas:

**Sam Altman – USA,** *y no:* **Sam AltmanUsa**.

Veamos nuestra *Query* con la función ***CONCAT(),*** según las especificaciones pasadas. La columna resultante de la concatenación será llamada con el alias ***Author.***



## *Resultado/Salida:*

*, este recurso también lo puede usar en columnas provenientes de datos cruzados.*

# ***From + Join***

Como usted ya sabe, por cuestiones de *normalización,* siempre se busca que cada entidad se estudie por separado; esto, al asignarles tablas independientes, aunque relacionadas, a cada una de ellas. Sin embargo, a la hora de presentar un informe, a la hora de generar información que sea valiosa, conviene unir dichas tablas; pues, justamente nos ayudaría a *extraer más conclusiones e información adicional* que responda a nuestras dudas.

**From + Join** es el primer puntapié para darle fundamento práctico a la teoría que esconde el concepto *Bases de datos* ***relacionales;*** lo que trata sobre, justamente, las ***relaciones.*** Es importante relacionar entidades entre sí para poder generar información y darles un sentido práctico a los datos que tenemos al ser contextualizados al mundo real, a las consultas reales que se pretenden responder en una organización, con o sin ánimo de lucro. Las *uniones* nos permiten, con mayor facilidad, convertir el código y los datos en información útil y diciente.

**Por ejemplo**, pongamos esto en contexto, si tuviéramos un negocio de compra y venta de libros, no es muy práctico saber todo acerca de la entidad ‘*libros’* solamente, … definitivamente no si no se puede ***unir***dicha entidad con la entidad ‘*clientes’* y la entidad ‘*autores’*; pues, no podríamos relacionar un libro vendido X con un autor Y que lo escribió ni con un cliente Z que lo adquirió; entonces, para extraer la información completa sobre esta consulta, usted debe necesariamente unir 3 tablas, dado que estos datos no sólo se encuentran en una entidad, sino en 3: ‘libros’, ‘clientes’ & ‘autores’. Dicho lo anterior, en SQL (y también en Tableau),**la información se crea cruzando tablas.** Si tenemos, por ejemplo, que queremos cruzar la tabla de “autores” con la de “libros” y tenemos en cuenta que la cardinalidad es 1:M respectivamente (porque se sabe que un autor puede *escribir* cuantos libros quiera, pero el mismo libro sólo puede ser escrito una sola vez por un mismo autor o grupo de autores), y que la relación es “escribir” justamente; entonces, estaríamos extrayendo información (*insights*) tipo **“El libro ABC fue escrito por el autor XYZ en la fecha 00/11/2222”. Tenga algo en cuenta: Si dos tablas se unen (o más), puede llamar (SELECT) cualquier columna de ambas tablas! (o más)**

Explicado ese pequeño abrebocas*,* tenemos que la sentencia ***From*** puede trabajar de la mano con la sentencia ***Join.*** Entonces, con ***From*** no sólo se selecciona una ***tabla*** de la cual queremos visualizar unos datos; sino que, asistiéndose de ***join,*** también podríamos unir esa primera tabla con una segunda o, dicho de otra forma, unir varias tablas entre sí a través de las relaciones que tengan o les hayan sido asignadas en un principio; es decir, más concretamente, uniendo llaves primarias con llaves foráneas en caso de existir dicho vinculo entre dos tablas. Eso es lo que veremos ahora.

Las sentencias ***Join*** tratan o parten de una base matemática relacionada a la *teoría de conjuntos,* entonces,la sentencia ***Join*** la veremos de una forma gráfica con algo que se llama: ***Diagramas de Venn. Los Diagramas de Venn*** son simplemente círculos que se traslapan, en algún punto, para ver dónde está la intersección de conjuntos y, en últimas, ver dónde hay un *traslape* de valores de un lado y del otro. Veamos esto.

**No sin antes adelantarle algo:** La tabla ***Pivote,*** la tabla principal de la cuál se unirán las demás tablas,es ideal que contenga todas las llaves foráneas de las tablas con las que piensa unirse; idealmente foráneas en los casos en los que se pretenda hacer un *join* con *más de dos* tablas y quiera referenciarse siempre con la *Pivote*; esto para hacer el proceso de *unión* de tablas más fácil, cómo poder unir a una misma tabla (pivote) todas las demás tablas. Sin embargo, **no es obligación.**

Hay ocasiones en que la tabla *pivote* no guarda una relación directa, entre llaves, con una tabla adicional que se pretende unir en el *join*; en dicho caso especial, puede valerse de una tabla secundaria, que no es *pivote,* que sí guarde una relación directa con la nueva tabla que se pretende incorporar (idealmente la tabla secundaria sí es conveniente que guarde una relación directa, entre llaves, con la tabla pivote) *… más adelante lo entenderá mejor.*

## ***Diferentes tipos de Join***

**1. Diferencia:** Cuando hablamos de“Diferencia” nos referimos a lo que tenemos en un lado, ignorando lo que está en el otro (se seleccionan los datos que están en una tabla ***A***, por ejemplo, y se excluyen los datos que están en una tabla ***B***); es decir, hay un ***Join*** que se encarga de mostrarnos solamente la información que tiene cada tabla por aparte (el ***Join*** de la ***Diferencia***). Ahora bien, si quisiéramos ser más exigentes en nuestra toma de datos, con el tipo de ***Join “Diferencia”,*** de nosotros depende si queremos excluir también los datos de la tabla ***A***, continuando con el mismo ejemplo, que se relacionen con la tabla ***B***.

O sea, quiero que quede claro, uno podría hacer una ***diferencia*** al tomar sólo los datos de una tabla ***A***, sin importar si sus datos se relacionan o no con los de la tabla ***B***; o, bien, siendo más rigurosos, podríamos tomar la totalidad de los datos de una tabla ***A*** excluyendo los de la tabla ***B***, pero excluyendo también aquellos datos de la tabla ***A*** que se relacionen con la tabla ***B;*** es decir, ignorando la intersección entre los conjuntos (***Exclusive Left Join***),***...*** En cualquiera de los dos casos, sepa que se estaría hablando de ***Left Join*** (si se supone a priori que la tabla ***A*** está situada en la parte izquierda y la tabla ***B*** está situada en la parte derecha).

Tal que así:



Ahora, suponiendo que la tabla ***A*** estuviera situada en la parte derecha y la tabla ***B*** en la parte izquierda,... estaríamos hablando de ***Diferencia*** de ***Right Join.*** Por suerte conservamos cada tabla en su mismo lado:



**2. Intersección o Inner Join:** Cuando hablamos de “Intersección” nos referimos a que los datos, de cada tabla, además de que se encuentran relacionados por la unión de sus registros, ambos registros al ser unidos deben cargar con contenido; es decir, al unir las tablas, hay información correspondida de lado y lado al unir sus registros. Más puntualmente estamos hablando de seleccionar los datos de la intersección entre los conjuntos. Si retomamos el ejemplo anterior estaríamos hablando de seleccionar los registros con datos que están tanto en la tabla ***A*** como en la tabla **B**. Por lo general, este es el tipo de ***Join*** que más se ve y el nombre correcto que recibe es: ***Inner Join (o Join)*** que en español traduce ***Unión interna.***



**3. Unión o Full Outer Join:** Cuando hablamos de “Unión”, estamos hablando de unir *definitivamente* ambas tablas, tanto ***A*** como ***B.*** Por ejemplo: Acá se seleccionarían y se unirían la totalidad de los datos, sin excepción alguna, de ambas tablas (incluso, sin importar si no hay correspondencia informativa al unir los registros de cada una, sin importar si hace falta información en una). *Unión* trae absolutamente todos los datos de ambas tablas.



**4. Diferencia simétrica oExclusive Full Outer Join*:*** Acá también se tomarían los datos de ambas tablas,... menos los datos que se relacionan entre sí al unir los registros de ellas (en el caso de que haya contenido, en términos de datos, en los registros de lado y lado de cada tabla). Si hay datos tanto en A como en B, en la conformación de un registro único que nace de dicha unión,déjalo por fuera. En resumen, se tomarían todos los datos de ambas tablas; pero se omitirían, solamente, los datos de la intersección entre los conjuntos. Se ve así:



***Ahora sí, veamos cada uno de los 4 ejemplos por medio de código SQL en MySQL Workbench, ... Le daremos sentido a todo esto.***

# ***Utilizando la sentencia FROM en MySQL***

***Poniendo en práctica Left Join & Right Join: bajo este tipo de Join uniremos “usuarios” con “posts”.***

Primero que todo nos interesa efectuar la práctica de ***FROM*** en la tabla ***Usuarios***; pero, antes que nada, visualicemos toda la tabla ***usuarios,*** en su conjunto, corriendo las siguientes sentencias que ya conoce:



Se nos *muestra*, como resultado, lo siguiente:



# ***Left Join.***

***Si hacemos un símil con los Diagramas de Venn, “usuarios” será “tabla A” & “posts” será “tabla B”;*** entonces, en ese orden de ideas, la tabla que usted seleccione de primero será instantáneamente la de la *parte* *izquierda,...* adicionalmente, debajo de ***FROM usuarios,*** se ha declarado la sentencia ***LEFT JOIN***,... con eso estamos afirmando que la prioridad la tendrá la tabla ***usuarios***; es decir, la ***tabla izquierda,*** la ***tabla A*** (Recuerde, la 1ra tabla que se pase siempre será la tabla del *lado izquierdo* o *tabla A*)***.*** Entonces, como la ***tabla A*** tendría la prioridad, de la unión resultante con la ***tabla B***, ***los datos de la tabla A se verían primero; y, de hecho, se verían en su totalidad,... es decir, se verían todos los usuarios independientemente de si los usuarios hayan escrito o no posts. Profundicemos esto. Si ponemos todo en contexto, estaríamos hablando del primer tipo de LEFT JOIN de los dos posibles que explicamos, “el menos riguroso”: el que también muestra los registros de la tabla B mientras los registros de la tabla A los involucre, es decir, mientras haya una intersección!***

***Si contextualizamos aún más, si lo vinculamos a nuestro caso de ejemplo, estamos hablando de que se mostrarían los posts escritos mientras haya registro de usuarios que los escriban (la intersección que nos devela que, efectivamente, hay usuarios escribiendo posts; es decir, que al unir las dos tablas se evidencia datos de ambas entidades registro a registro, de lado a lado)***. Esto último es lo más importante. Y acuérdese siempre de la relación que tengan ambas entidades; pues, serán fundamentales para las conclusiones que podríamos extraer de las *Queries* que se corran.

Ahora, continuemos con la práctica para poner en evidencia lo anteriormente explicado. Recuerde que su intención es, precisamente, hacer un tipo de ***Join*** que es de ***Diferencia,*** al tratar de unir una ***tabla A*** con una ***tabla B***; entonces, para señalar cuál pasaría a ser esa ***tabla B***,... se le pasa el nombre de la segunda tabla de su interés, en este caso ***Posts,*** justo al lado de la sentencia ***LEFT JOIN.*** Tal que así:



Ahora, ya dejamos claro qué tablas vamos a unir y hemos decidido, entre las dos, cual será ***tabla A*** y cual será ***tabla B,*** y que la unión será de tipo ***LEFT JOIN*** entre ambas; sin embargo, hace falta especificar la manera en cómo lograremos que ambas tablas, en definitiva, se unan y eso se logra ***conectando la llave primaria de la tabla A***, ***usuarios,*** que es la tabla de nuestra prioridad por ser la tabla de ***la parte*** ***izquierda*** y por hacer la unión desde un ***left join,...*** ***con la llave foránea de la tabla B*** que, justamente, se relaciona o viene importada de la llave primaria de ***usuarios.*** Lo anterior se logra al *igualar* ambas columnas, la de la llave primaria de ***usuarios*** con la de la llave foránea de ***posts*** (relacionada a usuarios), por medio de la palabra reservada **ON.** Se *igualan* porque se supone que, en efecto, sus datos son iguales entre sí. Veamos esto en código:



En ***usuarios.id, “usuarios”*** hace referencia a la tabla como tal, luego hay una separación de palabras por medio de un punto (.), e ***“id”*** hace referencia, ahora sí, a un campo de la tabla ***usuarios;*** que, como ya debe saber, es el campo correspondiente a la llave primaria de dicha tabla ***usuarios.***

Ahora, en ***posts.usuario\_id, “posts”*** hace referencia a la tabla como tal, luego hay una separación de palabras por medio de un punto (.), y ***“usuario\_id”*** hace referencia, ahora sí, a un campo de la tabla ***posts;*** que, como ya debe saber, es el campo correspondiente a una llave foránea (que se relaciona con la llave primaria de la tabla ***usuarios)*** de dicha tabla ***posts.***

En resumen, las tres sentencias conjuntas, al ser ejecutadas, nos dice lo siguiente:

***Selecciona todo, de la tabla “usuarios”, y únela con la tabla “posts”; reflejando primero la tabla “usuarios”, por ser la de la izquierda, y luego la tabla “posts”, por ser la de la derecha. Ahora, muéstrame todos los registros de “usuarios”, hayan o no hayan escrito “posts”; pero, fundamental, quiero todos los registros de “usuarios”.***  Como se dijo, esta unión se logra por medio de las dos llaves que las relaciona; es por ello que “usuarios.id” debe ser igual a “posts.usuario\_id” *(como se supone).*

Con esto tenemos los datos, unidos, de ambas tablas: ***usuarios & posts.***

Ahora, si queremos saber puntualmente que usuarios NO han escrito ningún post; recuérdese de la sentencia ***WHERE*** que nos ayuda a, precisamente, buscar un dato o unos datos en especifico, de una tabla pasada (o, en este caso, de la combinación de dos tablas por medio de sus llaves en común, una propia y otra foránea), según un filtro declarado; en esta ocasión el filtro es: ***querer saber qué posts están vacíos y a que usuario le corresponde;*** más concretamente, en qué ***Registros*** las celdas de la tabla ***posts*** estén vacías o ***NULL*** para x o y ***usuario.***

***Ahora, es necesario conservar una relación entre ambas tablas debido a que con la información de una, la información de “usuarios”, no podemos verificar quién ha escrito o no “posts” (la información que está en la segunda tabla). Primordialmente nos interesa saber quiénes son los “usuarios”, listo, ¿Cuáles? Los “usuarios” que no escribieron “posts” justamente. Como se involucra a la entidad “posts” en esta consulta que se pretende hacer; es necesario, repito, que se conserve un Join entre ambas tablas, que se conserve una interacción. Más precisamente, el tipo de Join que sería útil para resolver esta Query sería también uno de tipo Left Join, pero del segundo tipo, “el más riguroso”*** (***Exclusive Left Join***),***… pues, el “más riguroso” es el que al unir las tablas muestra todos los registros con información de la tabla A, siendo la del lado izquierdo, excluyendo o dejando por fuera los registros con información de la tabla B. En contexto, e interpretando la razón de ser de la consulta de nuestro ejemplo, esto sería reflejar sólo los registros donde hayan usuarios SIN posts escritos… usuarios que no hayan escrito posts.***

Como ya a este punto toca hacer búsquedas especificas con la sentencia ***WHERE*** también***;*** pero, recuerde, **sobre dos tablas unidas**; entonces, para respetar y mantener dicha unión entre ambas tablas… a ***WHERE*** hay que pasarle la llave foránea de la tabla ***posts****;* no cualquiera, sino, la que se vincula con la tabla ***usuarios*** justamente (para mantener la relación vigente entre ambas tablas, la relación de nuestro interés, y se haga la consulta correctamente)***…*** es ***posts.usuario\_id*** la llave que relaciona a ambas tablas al fin.

Ahora, uno habla de ***posts*** porque el filtro (***Where***) se aplica es sobre dicha tabla, no sobre otra,... se pretende *filtrar* sus registros vacíos o ***NULL*** con relación a los registros existentes de ***usuarios;*** *es decir, de nuevo,* ***se quiere saber que usuarios registrados no escribieron posts***… realmente cualquier ***NULL*** que encuentre, entre una tabla u otra, sobre un ***registro***, lo arrojará (mostrará). Como en este caso la prioridad la tiene la tabla ***usuarios,*** se mostrarán todos los datos de los ***usuarios*** registrados; por lo que, entonces, se espera ver los ***NULL*** en los ***registros*** de la tabla ***posts)...*** Lo que nos interesa saber efectivamente: ***usuarios registrados que no escribieron posts***. La declaración de la sentencia ***WHERE*** quedaría así:

, y ahora todo junto se vería así:





# ***Caso anterior con Right Join...***

La ***tabla B*** tendría la prioridad de la unión resultante con la ***tabla A***; es decir, ahora, ***los datos de la tabla B se verían primero que los de la tabla A; y, de hecho, ahora se verían en su totalidad los registros con información de la tabla B; esto, en otras palabras, significa que se verían todos los posts*** escritos***, ... independientemente de si haya registro o no del usuario que lo o los escribió.***

Como usted ya sabe, la 1ra tabla pasada siempre será la ***tabla A*** o tabla de la ***izquierda***; es decir, eso no va a variar ahora. Lo único que cambiará ahora es la declaración de la sentencia ***LEFT JOIN*** por la de ***RIGHT JOIN,...*** para darle la prioridad a la ***tabla B,*** a la tabla de la ***derecha,*** en este tipo de ***Join (y ya sabe lo que significa eso). En el siguiente ejemplo se evidenciarán todos los posts que fueron escritos de forma anónima; es decir, que si bien fueron escritos, no tienen el registro del usuario que los escribió.***

*Algo como un:* ***Right Join del tipo “más riguroso” (Exclusive Right Join), por decir algo****. Y, como se ha explicado muchas veces, toca hacer un Join para que, uniendo ambas tablas, se sepa la influencia de una sobre la otra… o si definitivamente no hay ninguna.* ***En este caso puntual, para que se sepa si el post que fue escrito tiene el registro de quién lo escribió o no.***

******



Por lo general las tablas se unen para hacer *filtros* entre ellas según la relación, de cardinalidad, que tengan. Recuérdese que la cardinalidad entre ***usuarios*** y ***tablas*** era de ***1:N*** y la ***relación*** era ***usuarios escriben posts.*** Base sus ***Joins*** de tablas, y los posteriores *filtros* que haga para ambas, al estar unidas, según la relación que conserven (como lo hicimos, justamente, con ***usuarios y posts***).

# ***Poniendo en práctica Inner (Intersección)***

Bajo este tipo de Join uniremos, ahora, “usuarios” con “posts”.

## ***Inner***



La ***intersección*** tiene que ver con que, recordemos, se evidencie información en cada registro de ambas tablas (a & b) al ser unidas; en consecuencia, la intersección no tendrá en cuenta ni visualizará los registros que se unan, de ambas tablas, en caso de que en alguno de ellos no haya información registrada; lo dicho, al unir los registros con *Inner Join*. Lo que estamos pidiendo en la consulta, ***con inner join,*** es que veamos todos los registros, tanto de la tabla A como de la tabla B, cuando ninguna de sus celdas estén vacías en cada uno de sus ***registros*** entrelazados; es decir, situando esta teoría a nuestro caso de estudio actual, cuando haya un post escrito y de paso se sepa el usuario que lo escribió***.*** Entonces, esta es la condición: las celdas, en cada uno de los registros para ambas tablas, no pueden estar vacías para que el ***Query*** las pueda mostrar; de lo contrario, el ***join inner*** no las mostrará. En contexto, de nuevo, ***join inner*** sólo nos mostrará los ***usuarios*** que hayan escrito ***posts*** y/o los ***posts*** escritos de los que se sepa quién fue el usuario que los escribió.



Acá se respeta el mismo orden en el que son nombradas las tablas. La ***tabla A*** o ***de la izquierda,*** manteniendo el símil con el diagrama de Venn, será la primera tabla llamada; es decir, ***FROM*** ***usuarios.*** Luego, la tabla B o de la derecha será la segunda llamada.

La palabra ***INNER JOIN*** quiere decir: *“sólo tráeme lo que está internamente ligado, tráeme solamente los que tienen dependencia de ambos lados”* (donde no haya celdas vacías en los registros de la tabla A ni en tabla B al ser unidas).

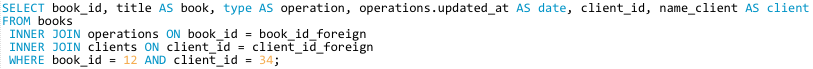


Este es uno de los ***Joins, Inner Joins,*** de los que más se van a utilizar y es, generalmente, el que más valor brinda. **Esto tiene su explicación:** resulta que con ***INNER JOIN,*** además depoder unir cuántas tablas deseemos en un mismo *Query,* es el recurso más recurrido para hacerlo debido a que, ***INNER JOIN,*** nos trae los registros, de todas las tablas que se unan, que cumplan con la condición de contener datos incorporados dentro de ellos (al ser relacionados los registros de todas las tablas unidas);esto, nos brindaría una mayor posibilidad de acceso a la totalidad de los datos disponibles de las tablas unidas y; entonces, con un mayor número de datos a nuestra disposición, en consecuencia, podríamos generar más información y extraer más conclusiones que respondan a la mayoría de tipos de preguntas generadas en la definición del problema inicial.

## ***Ej. de unión de tres tablas con INNER JOIN:***

***Hagamos un Join uniendo tres tablas llamadas: "books", "operations" & "clients" por medio de varios, dos para este caso puntual, INNER JOIN para responder a la siguiente pregunta.***

***Query general:*** Cuál es el nombre del cliente (clients) con ID 34 al que le fue vendido (operations) el libro (books), su *nombre*, con ID 12 y en qué fecha se llevo a cabo esta venta?

  
*Este ejercicio se encuentra resuelto en “2. from + join (importancia de JOIN en contexto)”, carpeta SQL.*

En la consola se ve cómo una tabla “books” se une (join), con una unión de tipo *INNER JOIN,* a una tabla “operations”. A este punto ya hay dos tablas unidas, sin embargo, nos interesan tres; he aquí donde “books”, en la línea de código inmediatamente posterior, se une a “clients” también (al igual esta unión es tipo *INNER JOIN*).

*Resultado/Salida:*



Si se da cuenta, hemos unido las tres tablas para extraer mas información al respecto, para poder responder a nuestra pregunta inicial con nuestro propio lenguaje y hacernos entender, incluso, con gente que no sabe nada de base de datos.

Viendo nuestro Query se entiende que, *el libro 'El llano en llamas' con ID 12 fue vendido el día 18 de Febrero del 2022 a la cliente María Teresa Castillo con id 34.*

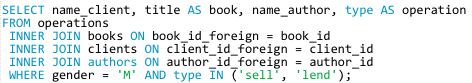
*Recuerda esto… “…* Sin embargo, **no es obligación.** Hay ocasiones en que la tabla *pivote* no guarda una relación directa, entre llaves, con una tabla adicional que se pretenda unir en el *join*; en dicho caso especial, puede valerse de una tabla secundaria, que no es *pivote,* que sí guarde una relación directa con la nueva tabla que se pretende incorporar (idealmente la tabla secundaria sí es conveniente que guarde una relación directa, entre llaves, con la tabla pivote) *… ”.* Pues bien, veamos esto bajo un ejercicio ya resuelto.

## ***Ej. de unión de cuatro tablas con INNER JOIN:***

***Hagamos un Join uniendo cuatro tablas llamadas: "books", "operations", "clients" & “authors” por medio de varios, tres para este nuevo caso puntual, INNER JOIN para responder a la siguiente pregunta.***

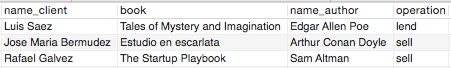
**Nota:** Tenga en cuenta que la tabla ***authors***no tiene ninguna relación directa con su tabla pivote ***operations****… En ese caso, ya se debe imaginar cómo debe proceder.*

***Query general:*** Tráeme todos los libros, sus nombres, que hayan sido vendidos o prestados a clientes varones. Quiero saber, además, quién es el cliente y quién es el autor del libro, haya sido vendido o prestado.

  
*Este ejercicio se encuentra resuelto en “2. from + join (importancia de JOIN en contexto)”, carpeta SQL.*

En la consola se ve cómo la tabla “operations”, que es la tabla pivote, se une (join), con una unión de tipo *INNER JOIN,* a una tabla “books”. A este punto ya hay dos tablas unidas, sin embargo, nos interesan cuatro ahora. Vamos con la tercera: “operations”, en la línea de código inmediatamente posterior, luego se une a “clients” también (al igual esta unión es tipo *INNER JOIN*). Ahora, esto es lo nuevo, resulta que *operations* no guarda ninguna relación directa con *authors,* la cuarta tabla que nos interesa unir. Entonces, nos hemos asistido de una tabla segundaria, *books,* que sí guarda una relación directa con *authors* para poder vincularla, vincular *authors* mediante este recurso, a la tabla pivote *operations.*

*Resultado/Salida:*



Sin embargo, hay muchas veces que es necesario utilizar ***Left*** o ***Right*** para que no te importe uno de los lados (que no te importe si uno de los lados no contiene datos en sus registros) siempre y cuando traigas todos los registros con datos del otro lado de la ecuación (por ejemplo, esto es útil para saber puntualmente qué registro se ha quedado huérfano al unir dos tablas, usted se podría empezar a preguntar el porqué de este fenómeno… A veces es necesario y nos sirve saber qué datos no están & cómo interpretarlo).

Es decir, solo nos arroja los registros de las tablas con datos incorporados; lo cual, nos brinda la máxima posibilidad de recolectar la totalidad de los datos disponibles (con lo que nos interesa trabajar para generar información y extraer conclusiones).

# ***Poniendo en práctica Unión***

Bajo este tipo de Join uniremos, ahora, “usuarios” con “posts”.

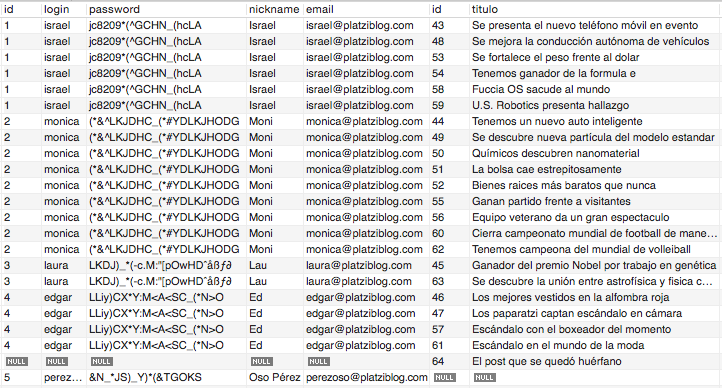
## ***Unión***



Este ***Join*** es muy claro. Como se dijo antes, cuando hablamos de *Unión,* estamos hablando de unir definitivamente ambas tablas, tanto ***A*** como ***B,*** sin importar si al unir los *registros* de cada una de las tablas hay celdas *vacías* o no*…* simplementetrae, une y muestra todos los registros, vacíos o no!

Este tipo de ***join,*** en algunos manejadores de bases de datos, no tiene una forma propia de sentencia para ser ejecutado; por lo que el *estándar* es que sea ejecutado al *unir* las sentencias ***left join*** (que muestra la totalidad de los registros con información de *la tabla izquierda* o tabla A con su registro correspondiente en tabla B; sin importar si las celdas, de cada registro de la tabla B, estén vacías o no) ***& right join*** (que muestra la totalidad de los registros con información de *la tabla derecha* o tabla B con su registro correspondiente en tabla A; sin importar si las celdas, de cada registro de la tabla A, estén vacías o no); evidenciándose así, por medio de dicha unión, la totalidad de los registros de cada tabla, esté su contraparte vacía o no. De hecho, si tú unes los diagramas de Venn del ***Left join*** con el de ***Right join*** te dará ***Unión.*** Es decir, en contexto, ***se verían todos los usuarios independientemente de si los usuarios hayan escrito o no posts; pero, también, se verían todos los posts independientemente de si haya registro o no del usuario que lo o los escribió.*** En código se vería así:





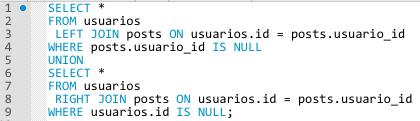
# ***Pongamos en práctica la “Diferencia simétrica”***

Bajo este tipo de Join uniremos, ahora, “usuarios” con “posts”.

## ***Diferencia simétrica***



Sin extendernos mucho, básicamente consiste en: ***ver lo que existe en A pero no en B y ver lo que existe en B pero no en A* (es justamente todo lo contrario a *Inner join*).** Tal que así:

****



## ***No lo olvide:***

La relación o vínculo de cada registro (registro por registro), al unir dos tablas, se da por medio de sus llaves; esto, al dejar señalada la equivalencia de una llave primaria (de una tabla independiente) con una llave foránea (de una tabla dependiente), … las cuales ya se encuentran relacionadas entre sí; es decir, donde la llave foránea ya fue definida por la importación de una llave primaria en su tabla dependiente.

# ***Dato extra: Sobre Dblink***

También se puede obtener la información de una base de datos remota o foránea; es decir, donde el esquema de interés, del cual nos interesa obtener información, se encuentre en otro **DBMS**.

Para obtener información de una base de datos remota se utiliza la función *dblink*, ampliamente usada en PostgreSQL. Esta función recibe dos parámetros:

1. *Configuración de conexión al DBMS remoto*
2. *Consulta SQL*

***Ejemplo de dblink.***

******

# ***Where***

***Where*** es ahora la sentencia que nos ayudará a filtrar, no columnas; sino, filas o ***registros.*** Esta vez nos va a permitir filtrar datos en específicos que, según unos parámetros pasados, desearíamos mostrar; no en el sentido totalizador de “*qué campos desearíamos visualizar en su conjunto”*, no; más bien, *qué registros puntuales ver según una condición de filtro pasada que deban cumplir.* Vamos al grano con ejemplos.

En concreto, la sentencia ***Where*** nos va a permitir filtrar los ***registros,*** de una o varias tablas combinadas con ***Join,*** que cumplan con uno o varios criterios: los que sirvan a nuestra consulta.

## ***Ejemplo 1.*** Operadores relacionales con “Igual que”

Queremos ver los ***registros*** de la tabla ***posts*** donde el campo ***id sea igual o menor*** que ***50,*** tal así:  




*Tenga en cuenta que los ID para los posts comienzan desde el 43*

## ***Ejemplo 2.*** Operadores relacionales con “No es igual que”

Queremos ver los ***registros*** de la tabla ***posts*** donde el campo ***estatus No sea igual*** a ***“activo”,*** tal así:  




**El dato**: *“No sea igual”* también se puede declarar con la siguiente simbología: “< >”, y no sólo con: “!=”.

Ahora, hay un ejemplo interesante con los tipos de dato ***string*** (cadenas de texto) que vamos a conocer. La sentencia ***Where*** dispone de una propiedad llamada ***LIKE*** que es muy utilizada en ***Queries*** para los casos en los que se desee visualizar los registros, para un ***campo (o varios),*** que contengan cadenas de texto (*strings*) que lleven consigo alguna palabra en especifica; es decir, ***LIKE***hace la búsqueda de todas las cadenas de texto que tengan consigo una palabra en concreto; dicha ***palabra*** es pasada por medio de la propiedad ***LIKE***, entre comillas *“”*,e internamente en las comillas se asiste de símbolos de porcentajes antes y después de la escritura de la palabra en cuestión ***(“% ... %”)***. La *palabra* vendría siendo realmente el filtro para poder visualizar o no los registros, registros relacionados a un campo o atributo de tipo ***string***; en concreto, con la palabra pasada es que se filtran los registros de cadenas de texto. Lo dicho, los registros con cadenas de texto que contengan dicha palabra, pasada con ***LIKE,*** serán arrojados como resultado dentro del ***query.*** Veamos esto mejor con un ejemplo.

## ***Ejemplo 3. Propiedad LIKE***

Queremos ver los registros de la tabla ***posts*** donde el campo ***titulo,*** dentro de sus cadenas de texto, tenga la palabra ***“escandalo”***(o una extremadamente parecida a ella), tal así:





Ahora, si por ejemplo queremos que la palabra ***“escandalo”*** sea necesariamente la primera palabra escrita dentro de todos los registros, del campo ***titulo*** de la tabla ***posts,*** y queremos visualizar cuáles ***registros*** cumplen con dicha condición, ... ***El primer símbolo de porcentaje (%) se omite,*** lo demás se deja igual**;** tal que así:





Si quisiéramos que la palabra ***“escandalo”*** esté al final, ***es el segundo símbolo de “%” el que se omitiría.***



Como ***ninguna cadena de texto*** del campo ***titulo*** (de la tabla ***posts***) finaliza con la palabra ***“escandalo”,*** entonces, no se logra visualizar ***ningún registro*** en el ***Query.***



## ***Ejemplo 4. Filtrando Fechas***

Queremos ver los ***registros*** de la tabla ***posts*** donde el campo ***fecha\_publicacion*** sean ***superior*** a la fecha ***“2025-01-01”,*** tal así:  




**Una acotación:**Los valores para los tipos de datos “fecha” *(date, datetime, time, timestamp, year)* se escriben entre comillas.

Ahora, si quisiéramos ver los ***registros*** de la tabla ***posts*** que respeten un intervalo de tiempo; es decir, que estén entre una fecha y otra, nos debemos asistir de la propiedad ***Between*** dentro de la sentencia ***Where.*** Inmediatamente después de declarar la propiedad ***Between*** se pasan las fechas en cuestión por medio de la palabra clave ***AND.***

## ***Ejemplo 5. Filtrando con BETWEEN & AND***

Queremos ver todos los ***registros de posts*** que hayan sido guardados entre el ***1 de enero del 2023 y el 31 de diciembre del 2025***, tal que así:  




La propiedad ***Between*** no sólo funciona sobre tipos de datos ***Data;***sino, también, sobre ***cualquier tipo de dato numérico***; por ejemplo, veamos ahora los ***registros*** de la tabla ***posts*** que vayan desde el ***id 50*** hasta el ***id 60***, tal que así:





**Formas más exóticas de proyectar los datos:** Con ***Where,*** en ocasiones, no hace falta ser siempre genéricos a la hora de hacer una ***consulta*** sobre un ***campo*** de cualquier ***tabla***; por ejemplo, en el caso del ***atributo*** o campo ***fecha\_publicacion***, no hace falta pasar toda la *fecha* en concreto para que el ***query*** haga la consulta sobre dicha columna; podríamos incluso basar, y hacer más puntual, nuestra intención de búsqueda según un parámetro propio, más especifico, extraído del campo en cuestión; en este caso, de ***fecha\_publicacion.* Veamos varios ejemplos.**



En la anterior consulta se especificó algo más puntual que toda la fecha de publicación, el año ***(year);*** por lo tanto, al ***Query*** sólo le interesa que se cumpla la condición de que los ***posts*** hayan sido publicados ***entre*** los años ***2023*** & ***2024*** (no le interesa ni en qué mes ni que día); del resto le importa poco; incluso, repito, no le importa los meses ni los días; sólo que se cumpla la condición de que hayan sido escritos, los ***posts***, entre el año 2023 y el año 2024.

***Un ejemplo más exótico todavía sería el siguiente:***

***, ... Adivina?* Acá sólo importa que el mes corresponda al número “04” (sin importar el año ni la fecha).**

## ***Where para Nulo o no Nulos***

Hasta ahora no hemos dejado muy claro cómo usar la sentencia ***Where*** para hacer consultas en aquellas celdas o valores de tipo ***Null (or Not null)***. Cuando nos referimos a celdas tipo ***null,*** o también ***default,*** hablamos de celdas *vacías, ...* celdas que no han recibido ningún tipo de valor (no han sido definidas). Este tipo de celdas ***null*** reciben un tratamiento particular si quisiéramos consultarlas, se asisten del parámetro: ***IS NULL.*** Vamos directo al *grano* con un ejemplo.

***Ejemplo 1.*** Queremos consultar, dentro de la tabla ***posts,*** aquellos ***registros*** donde los ***posts*** no tengan un ***usuario asociado.***

Recuérdese, como estamos viendo sólo la información de una tabla, la tabla ***posts;*** entonces para saber puntualmente qué ***columna,*** ***campo*** o ***atributo*** de la tabla ***posts*** tiene la información de los ***usuarios,*** teniendo claro que ***usuarios*** vendría siendo realmente otra entidad o tabla,... A qué recurso, de la misma tabla ***posts,*** tendría que recurrir para ver los ***usuarios*** ***asociados*** a ***los*** ***posts? Pues bien, a la llave foránea de usuarios dentro de la tabla posts;*** es decir, al campo o columna: ***“usuario\_id”.*** Lo dicho. La tarea consiste en consultar qué registros, de la tabla ***posts,*** tienen un ***usuario\_id*** equivalente a ***NULL***;es decir, sin usuario asociado. Tal así:





*La base de datos entiende que le estás preguntando sobre todos los registros dónde el usuario es “nulo”.*

Sin embargo, hay que precisar que la mayoría de las veces no nos interesa traer los ***nulos***; sino, mas bien, los ***no nulos***; siguiendo el mismo ejemplo, estaríamos hablando entonces de los posts que sí tienen un usuario asociado. Cuando queremos consultar registros, de cualquier campo, que ***NO*** tengan celdas ***null*** nos debemos asistir del parámetro ***IS NOT NULL (que no están vacías las celdas).*** Tal así:





***Otros ejemplos de Query con Where***

## ***Ejemplo, AND & OR.***

******

**Hay que saber distinguir los dos casos, porque los resultados de ambas Query son distintos.** El ***primer*** ***Select*** consiste en traer los registros, de la tabla ***users****,* donde ***name*** sea igual a “Israel” y se apellide (***lastname***) como “Vázquez” **o** “López”.

Ahora, el ***segundo*** ***Select*** consiste en traer los registros, de la tabla ***users****,* donde ***name*** sea igual a “Israel” **y** se apellide (***lastname***) como “Vázquez”. Luego, también nos interesa traer los registros donde ***lastname*** sea “López”, sin importar que nombre tenga el usuario en sí (no es necesario que ***name*** sea “Israel”, por ejemplo).

## ***Otro ejemplo, LIKE & NOT LIKE en Comodín***

Queremos ver los registros de la tabla ***users*** donde el campo ***name,*** dentro de sus cadenas de texto, tenga la palabra ***“Is\_ael”****.* El guion bajo que ve después de **s** y antes de **a**, \_, realmente vendría siendo un comodín; es decir, ahí podría ir cualquier tipo de carácter; por ejemplo, “Is**m**ael” o “Is**r**ael”, … cualquiera de los dos casos es válido y se mostrarían en pantalla, los registros que les contienen, en caso que existiesen en la tabla ***users***:



Ahora, el caso contrario al anterior sería mostrar en el Query todos los registros, de la tabla ***users,*** que NO (uso de la sentencia ***NOT LIKE***) contengan la palabra “***Is\_ael***” (teniendo en cuenta lo que el comodín, \_, se refiere). Quedando algo así:

******

## ***Otro ejemplo, IN & NOT IN***

Queremos ver todos los registros de la tabla ***users*** donde el campo ***name*** sea igual a ***“Israel”, “Laura” o “Luis”****.*

******

Ahora, el caso contrario al anterior sería querer ver todos los registros de la tabla ***users*** donde el campo ***name*** NOsea igual (uso de la sentencia ***NOT IN***) a ***“Israel”, “Laura” o “Luis”****.*

****** *Suponga que hay un “****NOT****” antes de “****IN****” y listo!*

## ***AND a profundidad***

Recordemos, se pueden combinar varias condiciones en ***WHERE*** asistiéndose de la palabra clave ***AND;*** es decir, se pueden unir varios criterios a nuestra búsqueda o ***Query.***

Por ejemplo, qué tal si además de ver todos los posts asociados a un usuario se cumpla también la condición de que el post esté activo; es decir, un ***registro*** para ser mostrado por el ***Query*** en mención debe cumplir con ambas condiciones: *que el* *post tenga un usuario asociado y que, dicho post, se encuentre también activo,* tal así:





Entonces, aquí se están filtrando los registros por la validez, no sólo de una condición; sino, de dos condiciones (pueden ser más incluso): que el post necesariamente tenga un ***usuario asociado*** y que, además, esté ***“activo”.***

Ahora un ***Query*** más exigente (donde se cumplan, ya no dos; sino, tres condiciones al tiempo). A las dos criterios de búsqueda pasados, anteriormente, agreguemos que: el ***ID sea menor que 50.*** Tal así:





Agreguemos una cuarta condición más, que la ***categoria\_id sea 2.*** Tal así:





Con ***Where,*** si lo nota, podemos filtrar tanto como queramos!

Si por ejemplo, además de las otras condiciones, le agregamos una última que nos arroje en el ***query*** solo los ***posts*** que hayan sido publicados en el año ***2025,*** sería así:





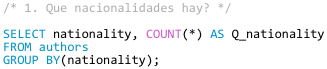
Nos arroja sólo un ***registro*** porque, precisamente, es ese el único registro que cumple, en su conjunto, con todas las condiciones pasadas.

# ***Group by***

***Group By***, como sentencia, trabaja de la mano de funciones de agregación tipo: **Sum(), Count(), Avg(), Max(), Min(), Stddev(), …** Entonces recordemos, ***Group By*** lo que hará es, según el campo que se le asigne, mantener la ejecución de una función de agregación asociada al *Query*; pero no como un todo, no se ejecutará la función de forma completamente *agregada* sin discriminar los registros;sino, sobre cada una de las *categorías* que hayan dentro del campo que ha sido pasado dentro de la sentencia ***Group By,*** distinguiendo la *Agregación* por categorías: por los datos que hayan para cada una de las categorías que hay para el campo pasado en ***Group By***.

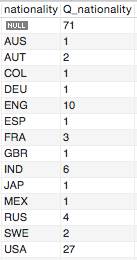
En un principio habíamos tratado de explicar este concepto por medio de un ejemplo con la columna ***city***, de la tabla ***people***, haciendo una consulta con la función de agregación **COUNT**. Habíamos planteado que ***Group By*** nos permite ***agrupar*** datos según un criterio pasado (cuando hablamos de *criterio* nos referimos realmente a una columna o campo); entonces, si retomamos el ejemplo, fue nuestra columna ***city*** nuestro criterio asignado para *agrupar* los datos, en contexto, el ***Query*** con ***Group By city*** lo que hará es *contar* todos los registros de ***people***haciendo el conteo por ciudad (**city**).La función de agregación, en este caso ***Count()***, no se calcula como un todo sobre todos los registros, sino que se calcula por categorías según campo asignado; en este caso el campo asignado es ***city***, entonces, **se van a contar todos los registros por ciudad y el conteo se agrupa para cada una de esas ciudades, por separado.**

Veamos el siguiente ejemplo, nos interesa saber cuántos registros hay para cada una de las nacionalidades (***nationality***) almacenadas en la tabla ***authors***; es decir, cuántos autores hay para cada nacionalidad evaluada.



También era posible mostrar estrictamente las nacionalidades, es decir, una sola columna (ya que no piden conteo de ellas); en ese caso, sólo sería necesario *suprimir* el código *COUNT(\*) AS Q\_nationality* del query ejecutado.

## *Resultado/Salida:*



**Importante:** cuando se trabaja con cualquier función de Agregación,que se pasa por ***SELECT,*** lo que se crea es una nueva columna que recibe por etiqueta la misma escritura de la función agregada. Por ejemplo, si la función de Agregación fuera ***COUNT()***, el nombre de la columna sería ***“COUNT()”*** también;que, justamente, nos arrojará el resultado de un conteo (para este caso).

El conteo, o cualquier operación matemática derivada de una función de Agregación, se hace para un campo o columna pasada a ***COUNT()*** entre sus paréntesis (esto es ideal si precisamos que nuestra función de Agregación se ejecute sobre los datos específicos de una columna). Sin embargo, cuando la función de Agregación es sobre el **total** de los registros, de una tabla seleccionada, hablamos de ***COUNT(\*).*** Este último criterio, ***COUNT(\*)***, sería: ***selecciona y cuenta todos los registros de la tabla en cuestión*.**

Ahora, si quizás a usted no le interesa que el campo creado con ***COUNT(\*)*** se llame “**COUNT(\*)”**; entonces, usted podría sustituir el nombre del campo, antes de crearlo, asistiéndose de la palabra reservada ***AS*** y pasándole un nuevo nombre identificativo al campo en cuestión***.*** Veamos todo lo abordado en un ejemplo.

## **Ejemplo 1.**

Queremos ***contar,*** con Count(\*), la totalidad de registros (posts publicados) que hay para cada uno de los valores, ***active*** e ***inactive,*** de la columna seleccionada ***estatus;*** todo esto por medio de ***agrupar,*** con ***Group by***.

Cambie el nombre de la columna ***Count(\*)***, posiblemente resultante, por el nombre ***“post\_quantity”.*** Tal así:





Hemos contado la cantidad de registros que hay para cada una de las categorías que asume la columna **estatus.**

## **Veamos otros ejemplos. Ejemplo 2.**

Queremos ***contar,*** con Count(\*), la totalidad de posts publicados que hay para cada año por separado, los cuales fueron registrados en los datos de la columna ***fecha\_publicacion***; es decir, toca ***seleccionar*** dicha columna *(fecha\_publicacion)* para evaluar cuántos posts fueron publicados por cada año en cuestión; en consecuencia, adicionalmente, toca ***agrupar*** con ***Group*** ***by*** la información de la misma columna ***fecha\_publicacion***.

Ésta última, para efectos prácticos, la identificaremos con un nuevo campo dinámico llamado ***post\_year***; y ahí, en ***post\_year***,sólo nos interesa ocupar la información relacionada a los años, no las fechas completas de la columna **fecha\_publicacion**; esto para decir puntualmente la cantidad de posts por año, OJO, no la cantidad de posts publicados en cada una de las fechas registradas en **fecha\_publicacion**; si ese fuera el caso, no funcionaría nuestra función por años **YEAR()**. En resumen, nos toca expresar la información de **fecha\_publicacion,** que está con un tipo de dato de ***TIMESTAMP***, en términos del tipo de dato ***YEAR.*** Veamos:





*Agrupemos* ahora la cantidad de posts publicados pero en términos de los 12 meses del año ***(MONTHNAME);*** es decir, la cantidad de posts por mes sin importar el año. Toca entonces ahora expresar la información de **fecha\_publicacion**, que está con un tipo de dato ***YEAR***, en términos del tipo de dato ***MONTHNAME.*** Tal así:





Si se da cuenta *Group by* lo que hace hasta ahora es tratar, de manera particular, cada una de las categorías que tenga el campo seleccionado; es decir, refleja toda la información *agregada* de los registros del campo *seleccionado*, tal como lo hace un *Select,* pero con *Group by* se distingue la *agregación* sobre cada categoría del campo en cuestión. **Ej.:** No se cuentan la totalidad de los posts sin discriminar, sino que se cuentan por mes.

## ***Varios criterios en Group by***

Usted recuerda la siguiente afirmación: … “***Group By*** nos permite ***agrupar*** datos según un criterio pasado (cuando hablamos de *criterio* nos referimos realmente a una columna o campo) …”

Pues bien, *pueden ser más de un criterio los pasados* tanto para el ***Select*** como para el ***Group By;*** es decir, se puede organizar o *agrupar* la información de su tabla (o sus tablas unidas) según las categorías de más de una columna. Si se supone que aún estamos trabajando con la función de Agregación **Count(\*)**, **lo que sucederá es que se hará un conteo de todos los registros según las categorías de cada una de las columnas seleccionadas, analizándolas conjuntamente, y conservando la prioridad sobre la 1ra columna pasada**.

Entonces cada criterio (o campo) pasado en el ***Select*** y en el ***Group*** ***By,*** cada nueva columna, es separada por comas en cada una de las líneas de código del ***Select*** y del ***Group By***. Esta vez los criterios pasados, para hacer la ***agrupación***, serán las columnas del ***estatus*** y la nueva columna dinámica ***post\_month***, tal que así:





Si se da cuenta, me hizo el conteo de la totalidad del posts publicados (registros), no sólo para cada mes por separado; sino, para cada mes por separado y también para cada *estatus* dentro de cada uno de esos meses, ya sea *activo* o *inactivo* (el *estatus*); es decir, el filtro ahora está más elaborado y es más específico.

**Las categorías de ‘estatus’ son: ‘activo’ & ‘inactivo’; mientras que, las categorías de ‘post\_month’ son los 12 meses de un año natural. Entonces, las categorías de ambos campos han trabajado conjuntamente para determinar qué registros (cálculos) se extraen según la función de Agregación aplicada sobre el campo ‘posts\_per\_month’.**

# ***Order by***

***Order by*** es una sentencia que tiene que ver con el ordenamiento de los datos. ***Order by*** quiere decir, justamente, ***ordenar por***; es decir, le vas a decir a la base de datos**de qué manera quieres ordenar los datos, descendente o ascendentemente, según x o y criterio (campo) pasado.** Es decir, con ***Order by*** no se hacen filtros; no, sino que se pretende mostrar una misma cantidad de información según un tipo de *orden* que le pasemos. Veamos esto con un ejemplo.

***Ejemplo 1.*** Si quisiera ver ordenado todos los registros de la tabla ***posts***, de tal manera que se me ordene u organice la lista de todos los posts publicados según su fecha de publicación, podríamos hacerlo con ***Order by;*** de hecho, podríamos establecer si queremos que el ordenamiento se haga desde la fecha de publicación más reciente a la más antigua (con la propiedad ***DESC***) o desde la fecha de publicación más antigua a la más reciente (con la propiedad ***ASC***); sin embargo, si no se le pasa ninguna de las dos propiedades, el ordenamiento por defecto se hará desde la fecha más antigua a la más reciente; es decir, como si fuera definido con ***ASC.***

*Probemos ambas propiedades: ASC & DESC*

## **Ordenar por tipo de dato “data” o fecha;**

desde la fecha más antigua a la más reciente o, bien, de forma ascendente: propiedad ***ASC***.





*Ahora,* **Ordenamiento de registros por tipo de dato “*data”*** o fecha; desde la fecha más reciente a la más antigua o, bien, de forma descendente: propiedad ***DESC***.





*Ahora,***nos interesaría probar un Order by por tipo de dato *“string” o*** cadena de texto; acá el ordenamiento también se asiste de las palabras reservadas: ***ASC & DESC.*** Trabaja de la siguiente manera: El ***Order by***, en cadenas de texto, ordena u organiza los registros de forma alfabética; de tal manera que, la propiedad ASC organiza los registros desde la A a la Z; y, por el contrario, la propiedad DESC organiza los registros desde la Z a la A. En ambas se parte o se toma como referencia, como criterio de ordenamiento, la inicial de cada registro. Sepa que, por defecto, si no se le pasa ninguna palabra clave a Order by, éste se ordenará de forma ascendente. Tal así:

## **Ordenar por tipo de dato “*string”* o cadenas de texto;**

desde los registros que empiezan con la vocal A hasta los que empiezan con la letra Z o, bien, de forma ascendente: propiedad ***ASC***. Veamos esto en la columna o atributo ***titulo*** (que me ordene de forma *ascendente* los registros de los posts según sus títulos).





**Ahora, revisemos el caso contrario,** ordenemos los registros desde las cadenas de texto que empiecen con la letra Z hasta las que empiecen con la letra A o, bien, de forma descendente: propiedad ***DESC***. Veamos esto en la misma columna o atributo ***titulo*** (que me ordene de forma *descendente* los registros de los posts según sus títulos).





## **Ordenar por tipo de dato “*number”* o numérico.**

Vamos al grano salvajemente: usted decidirá si quiere organizar los registros de una columna, que es de tipo numérica, de menor a mayor (con la propiedad ***ASC***) o de mayor a menor (con la propiedad ***DESC***). Probemos esto con el atributo o columna ***usuario\_id.***

*Ordenamiento de registros, de menor a mayor número. según la columna* ***usuario\_id****.*





*Ordenamiento de registros, de mayor a menor número. según la columna* ***usuario\_id****.*





## ***Order by* puede recibir varios parámetros, no sólo uno.**

Es decir, puede ordenar según varias columnas (separadas por coma).

Debe saber que ORDER BY solo funciona con las columnas incluidas en el SELECT. **Ahora, él ordena según la prioridad de la columna**; entonces, en la medida que la columna sea llamada de primero, la prioridad será mayor. Primero ordena según el criterio de la primera columna pasada y, luego de que termine con ella, ordenará tomando en cuenta el criterio de la segunda llamada, y así sucesivamente…

Aparte, hay una sentencia de complemento a la sentencia ***Order by*** que se llama ***Limit;*** en ***Limit*** se determina la cantidad de resultados que queremos que arroje el manejador de bases de datos; esto, claro, luego de determinar en qué orden, y con qué columna de referencia, serán presentados los datos al asistirse con ***Order by.*** Veamos.

# ***Limit & Offset***

Por ejemplo, si queremos ver ***los primeros 5 posts que se publicaron*** (es decir, los primeros 5 registros)tomando como referencia la columna de ***fecha\_publicacion***, sería así:





Como estamos hablando de ***las 5 primeras fechas (LIMIT 5)***, entonces, tendríamos que asistirnos también de la sentencia auxiliar ***ASC;*** pues, los registros los queremos ver de forma ascendente según, o tomando como referencia, la columna ***fecha\_publicacion;*** es decir, que vayan desde la fecha más antigua ***(los primeros)*** a la fecha más reciente.

Con la herramienta ***Limit*** podemos hacer ***Top10,*** o ***Top100,*** de cualquier cosa que usted se imagine y sea medible; es decir, de cualquier tipo de dato en últimas: ya sea fecha, número o cadena de texto.

Muy contrario a ***Limit,*** existe un comando que, en vez de registrar un listado máximo de 10 o 100 registros, como lo hace ***Limit,*** los salta! Ese comando es: ***OFFSET.*** Es decir, existe un comando que, según el número que le pases, dicho número, sería el número correspondiente al total de registros iniciales que NO se tomarían en cuenta NI, por tanto, se mostrarían por pantalla (se los vuela). Idealmente este recurso se asiste también de la sentencia ***ORDER BY & LIMIT***.

***Ejemplo:***

******

Lo que plantea la ejecución de la última sentencia, , es que se seleccionen *todas* las columnas de la tabla ***tabla\_diaria*** y se muestren por pantalla todos sus registros; *pero*, **con dos condiciones**, que los primeros 1500 registros que serían reflejados de la tabla ***tabla\_diaria*** no se tengan en cuenta (no se muestren por pantalla); y que, **luego de “volarse” esos primeros 1500 registros, sólo se nos muestre los 1500 registros inmediatamente posteriores, … después de esos primeros 1500 omitidos.**

Más adelante estaremos abordando un nuevo tema que se conoce como los ***índices,*** este nuevo recurso es para búsquedas y ordenamientos más complejos.

# ***Función RAND()***

Suele usarse como complemento de la sentencia ***ORDER BY***; pues, si se operan juntas, el *Query* ordenaría de forma **aleatoria** la totalidad de los registros mostrados por pantalla (como salida). En código esta cooperación se ve así: ***ORDER BY RAND().*** Por ejemplo, ordenemos de forma aleatoria los 10 primeros registros de la tabla *books*. Al incluir la función *RAND()* el código se vería así:

../../../../../../Captura%20de%20pantalla%202022-02-26%20a%20las%209.48.51%20p.m..png

Cada vez que usted ejecute este código, su *Query* será enlistado con un orden diferente; es decir, los registros de la tabla *books* serán ordenados de forma aleatoria con cada ejecución del código que se haga.

**Aclaración:** Evítese en lo posible el uso de esta función, no es recomendable su uso programáticamente; pues, no es muy eficiente. Especialmente evite su uso en los manejadores de bases de datos, ya que ocupan más memoria dinámica que una terminal para poder funcionar y correr *scripts*.

# ***Having***

***Having*** tiene una gran similitud con ***Where;*** pero, justamente, de lo que se trata acá es de ver cuáles son sus posibles diferencias. Usted se recuerda de esto:

***... Where*** es ahora la sentencia que nos ayudará a filtrar, no columnas; sino, filas o ***registros.*** Esta vez nos va a permitir filtrar datos en específicos que, según unos parámetros pasados, desearíamos mostrar; no en el sentido totalizador de “*qué campos desearíamos visualizar en su conjunto”*, no; más bien, qué registros puntuales ver según una condición de *filtro* pasada que deban cumplir. Vamos al grano con ejemplos.

En concreto, la sentencia ***Where*** lo que nos va a permitir hacer es filtrar y decir cuáles ***registros,*** de una o varias tablas combinadas con ***Join,*** nos van a servir en nuestro reporte; esto según un parámetro o condición especificada.

## **Ejemplo 1.**

Queremos ver los ***registros*** de la tabla ***posts*** donde el campo ***id sea igual o menor*** que ***50,*** tal así:





*Tenga en cuenta que los ID para los posts comienzan desde el 43*

... Pues bien, la sentencia ***Having*** hace exactamente lo mismo; sólo que, en vez de ***Where,*** la sentencia ***Having*** es la que se usa en los casos puntuales donde se pretende hacer filtrossobre **registros agrupados y/o de columnas dinámicas.**

Es decir, ***Where*** nos ayuda hacer filtros sobre los registros, sí, pero los filtra antes de correr la sentencia ***Group by***; dicho eso, ***Where*** no puede trabajar de la mano de ***Group*** ***by,*** porque Where *desconoce* dicha sentencia y, en consecuencia, no puede hacer una selección o filtro de filas o registros agrupados ni de columnas dinámicas; en cambio, ***Having*** sí. La sentencia ***Having*** se coloca justo después de la sentencia ***Group by.***

## ***Veamos el siguiente ejemplo de uso de Having:***

Se pretende ***agrupar,*** y ***contar,*** la cantidad de posts publicados según cada una de las categorías que contengan las columnas ***fecha\_publicacion*** & ***estatus***; esto, con la condición de que se muestren solamente los registros donde se publicó ***más de un posts***. Lo dicho, se tiene que tener en cuenta cada una de las categorías de las columnas ***fecha\_publicacion*** & ***estatus*** donde se coincida, conjuntamente y en un mismo ***registro,*** con la condición evaluada: *más de un post publicado*. Además, se desea tomar como referencia la columna ***fecha\_publicacion*** para ordenar los datos seleccionados. Tal que así:





Como no se le pasó un ***asc*** o un ***desc*** a ***order by;*** él, por defecto, organiza los datos de ***post\_month*** de forma***ascendente.*** Quizás usted de entrada no lo note, pero si se fija bien, los ***meses*** están organizados de tal manera que la vocal ***A***, como inicial,tenga la prioridad sobre el primer registro enlistado hasta ***ascender*** a la letra ***Z;*** es decir, tomando como referencia la inicial de cada ***post\_month,*** los registros se ordenan desde la ***A*** a la ***Z.***

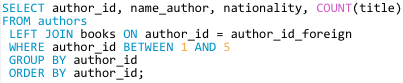
*Ahora, analizando este conjunto de sentencias, la idea es que primero el manejador de datos agrupe y cuente, posteriormente que filtre (sobre los datos contados); y, luego, que se ordenen dichos datos según unos criterios deseados.*

# ***Lo más importante...***

Probablemente lo más importante del manejo de una base de datos relacional consiste en ***saber convertir una pregunta en un Query SQL;*** en la medida, con práctica, se va perfeccionando este *arte*.

# ***Prueba.***

Según todo lo que ya ha aprendido, ¿podría entender el siguiente Query y su consecuente salida?



*Resultado/Salida:*



Si sabe interpretar el significado de “**COUNT**(**title**)” & “**GROUP** **BY** **author**\_**id**” del anterior Query?

R/ En contexto, esto significaría saber cuántos registros de “***title***” (no importa si se repiten) corresponden a cada uno de los valores, por separado, de “***author***\_***id***”; es decir a cada *ID* de autor, en últimas, a cada autor. Si lo anterior no sucediera, si no nos asistiéramos de los recursos de “**COUNT**(**title**)” & “**GROUP** **BY** **author**\_**id**”, NO tendríamos los registros agrupados por cada valor o *categoría* de **author**\_**id** (sin repetirse éstas) ni el conteo de la totalidad de los registros de **title** para cada valor o *categoría* que asume **author**\_**id. Se vería así:**



# ***Cómo estructurar una pregunta a un Query***

Vamos a reconocer, de una manera directa y concisa, cómo interpretar y relacionar las consultas que queramos realizar en nuestra base de datos relacional con cada una de las sentencias más importantes de SQL para hacer, justamente, las Queries de nuestro interés. Es decir, nos interesa darle un significado más práctico y en contexto a cada una de las partes (sentencias) de un Query; para que, cuando se te presente una pregunta en el mundo laboral real, tú sepas relacionar su equivalente en relación a, precisamente, el lenguaje SQL y a las sentencias derivadas o más usadas en un Query.



En ***Select*** por lo general se selecciona todo o una(s) Columna(s), dinámica o no, en especifica (Atributo alguno).

En ***From*** sesuele seleccionar a la tabla (o tablas) de interés; es decir, de dónde será seleccionada su(s) Columna(s).

En ***Where,*** luego de que ya has seleccionadolo que quieres mostrar ***(Select)*** y de dónde lo vas a seleccionar ***(From)***; se especifica con filtros, puntualmente, lo que te interesa extraer de ahí ***(qué*** ***datos puntuales).***

En ***Group by*** por lo general se seleccionan las mismas columnas que en ***Select.***

En ***Order by*** se establece, según una columna seleccionada, en qué orden será presentada la información: de mayor o menor con ***desc*** o de menor a mayor con ***asc.***

En ***Having,*** prácticamente,se hace lo mismo que en ***Where*** sólo que se aplica solamente para *Queries* con datos agrupados (***Having*** se usa cuando utilizamos ***Group by***).

# ***Queries Anidados***

**Las *Queries Anidadas* consisten en ejecutar una sentencia, una principal, al ejecutarse primero una *query* contenida dentro de dicha query principal**. Ahora, la *sentencia anidada;* es decir, la sentencia interna que se ejecuta, se ejecuta dentro del principal Query.

Luego, cuando se ejecuta una sentencia dentro de otra, cuando se llama, se llama entre paréntesis… Empecemos por ahí. Más adelante podríamos hablar de los beneficios de este recurso de *asistirse de un Query para crear otro*; mas, por lo pronto, evidenciemos en un ejemplo cómo se ve la estructura de un Query Anidado en la consola. Más precisamente nos interesa saber cómo es visualmente un Query anidado al compararlo, o al hacer un símil, con uno que no lo es pero que arroja el mismo resultado; es decir, al compararlo con su homologo que no se asiste de un Query anidado para mostrar por salida lo mismo. *Ejemplo*:



Céntrese en el segundo caso, en “El mismo Query solo que anidado”. Si usted analiza solamente el Query que está anidado dentro de *VALUES*, dejando por fuera un momento el análisis de ese Query en su conjunto, se dará cuenta que realmente se está ejecutando lo mismo al comparar los dos *VALUES*, los que *están en el medio,* de los dos casos: tanto del Query anterior (que no usa Queries anidados) como del Query que sí, nuestro último caso;

es decir, al comparar “**6**” con

“**(SELECT author\_id**

**FROM authors**

**WHERE name\_author = 'Octavio Paz'**

**LIMIT 1)**”.

Observe que simplemente se está llamando al *id* o *Primary key* (author\_id) de la tabla ***authors*** (From authors) que corresponde (Where) al autor Octavio Paz; es decir, se está llamando lo mismo: el numero ***6.***

**El número 6, en este caso, sería nuestro número entero que podría relacionar ambas tablas, tanto “books” como “authors”… Es 6 debido a que fue el *author* con *id* igual a 6 quién escribió dicho libro, el libro: “Vuelta al laberinto de la Soledad”.** Como consejo de buena práctica, en el *SELECT* de Queries Anidadas, siempre corra al final el siguiente recurso: **LIMIT 1** para asegurarnos de que efectivamente sólo se traerá un solo valor, ese valor para este caso es **6.**

**El resultado de ambos casos de Queries es el mismo:**



Finalmente, use esta herramienta con cuidado. Es ideal y se requiere que tenga completo dominio del tema para ser usado este recurso; como si fuera poco, le consume a la consola el doble de tiempo para ejecutar la sentencia en su conjunto; es decir, para correr la sentencia que anida y la que está siendo anidada (son dos).

# ***Práctica: Casos de negocio y función DISTINCT.***

Evalúe los siguientes ***Querys*** en su manejador de base de datos y saque conclusiones. La base de datos relacionada a este ejercicio es: ***platzi\_operation.***



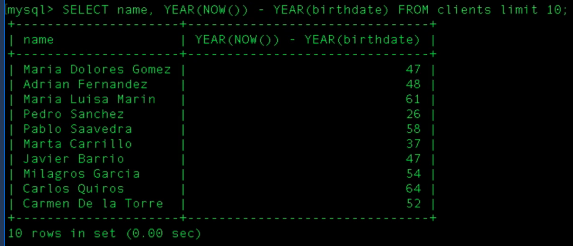
# ***Función NOW con SELECT***

**SELECT(NOW);** nos regresa la fecha más reciente, a tiempo real, que registra la computadora, incluyendo segundos. Tal que así: 



**Caso práctico.**

Relacionando nombres de clientes con sus respectivas edades por medio de un ***query****.*

**

*Observe el* ***Query*** *ejecutado:* **SELECT** name**, YEAR(NOW()) – YEAR(**birthdate**)**

**FROM** clients **LIMIT** 10.

Como la expresión: “**YEAR(NOW()) – YEAR(**birthdate**)**” realmente representa la ***edad*** del cliente al correr dicho cálculo, entonces, por razones meramente estéticas podríamos pasarle un ***alias*** más descriptivo a la expresión evaluada; de tal manera que, la columna que se arroje como salida del *query* identifique de mejor manera la información que se pretende mostrar; es decir, la edad del cliente… ya no verá en el encabezado de la columna “**YEAR(NOW()) – YEAR(**birthdate**)**”, sino, “***edad***”.

Su sintaxis se vería así:

**SELECT** name**, YEAR(NOW()) – YEAR(**birthdate**) AS** edad

**FROM** clients **LIMIT** 10.

# ***Función TO\_DAYS()***

Esta función permite recibir como parámetros los valores con tipos de dato de tipo *fecha*, más precisamente los tipos de dato **TIMESTAMP & DATATIME.** Dentro de los paréntesis se pasa el atributo de tipo *TIMESTAMP* o *DATATIME* (o la fecha en bruto, por ejemplo, **0000-01-00**). **Es preciso decir que TO\_DAYS() recibe fechas desde, justamente, 0000-01-00.** Ahora, cuál es la importancia de esta función? **TO\_DAYS() te permite saber la cantidad de días que han pasado, para una fecha pasada dentro de los paréntesis de TO\_DAYS(), desde la fecha originaria 0000-01-00.** Por ej.: si corremos por consola el siguiente código: **TO\_DAYS(0000-01-01),** tendríamos por pantalla la salida equivalente a: **1.** Pues, los días pasados desde **0000-01-00 a 0000-01-01** son equivalentes a **1. Veamos esto mismo dentro del manejador de base de datos** (asístase con un *SELECT*).

****

*Resultado/Salida:*

****

Lo interesante de este recurso es que nos permite hacer operaciones matemáticas, entre fechas, al ser llamadas dentro de esta función; es decir, podemos tomar dos fechas asistidas por la función ***TO\_DAYS()***y operarlas matemáticamente. Usualmente esta técnica, de operar dos fechas matemáticamente, es común en los casos en los que nos interesa saber qué días han pasado desde que se ejecutó una tarea hasta el día de hoy. Por ejemplo, un caso práctico sería, ***¿Cuántos días han pasado desde qué Pedro rentó el libro ABC del autor Juan?*** Generalmente, esto se respondería en código de la siguiente forma:

**SELECT TO\_DAYS(NOW()) – TO\_DAYS (tablename.created\_at)**

donde “tablename” es el nombre de la tabla que se pretende afectar y que contiene el atributo “created\_at”.

y se le pasa idealmente un alias, podría ser: ***ago*** (para referirse al tiempo que ha pasado). **Quedando así:**

**SELECT TO\_DAYS(NOW()) – TO\_DAYS (tablename.created\_at) AS ago**

Entonces, estamos creando otra columna (con el alias ***ago)*** que nos dirá cuántos días han pasado desde la fecha en la que se registró la renta **TO\_DAYS (tablename.created\_at)**, suponiendo que estamos tratando el mismo ejemplo de *Pedro,* hasta el día de hoy **TO\_DAYS(NOW())**.

Recuerde que por convención *created\_at* es el atributo que se crea para almacenar la fecha, a tiempo real, que corresponde al momento exacto en el que se introdujo un nuevo registro para cierta tabla. Es decir, registra y guarda el tiempo exacto en qué se insertó un nuevo registro (y esta información la guarda sobre el mismo registro en sí, bajo su columna o atributo *created\_at*); en este caso, siguiendo el mismo ejemplo, *created\_at* sería *la fecha equivalente al momento exacto en que Pedro rentó el libro ABC del autor Juan.*

# ***EXTRA: Comando TRUNCATE***

Este es un **comando de mucho cuidado**, pues, es uno de esos pocos comandos que pueden eliminar datos de forma masiva sin ningún tipo de restricción. ***TRUNCATE,*** al ser ejecutado sobre una tabla, se encarga de eliminar la totalidad de los registros que ésta contenga sin eliminar la estructura de la tabla ni, mucho menos, la tabla misma. Por ejemplo, si quisiéramos suprimir o eliminar todos los datos registrados en la tabla *operations,* sin eliminar su estructura; es decir, conservando la tabla en sí con sus atributos (incluida la información de sus tipos de datos, conformación de llaves primarias y foráneas y, adicionalmente, sus constricciones), podemos ejecutar el siguiente código:



Este simple, pero muy poderoso, código eliminaría todos los datos registrados de su tabla *operations.*

**Nota aclaratoria:** No es recomendable su uso. En caso que considere que no necesitará nunca más los registros de una tabla, incluso aunque sea sólo uno, lo ideal es “desactivar”el registro (o registros) en cuestión. Puede hacer uso del recurso enseñado previamente; donde creamos atributos, nombrados por convención, de tipo *active.* Con el atributo *active* podemos desactivar, por medio de la sentencia *UPDATE,* cualquier registro que esté “activo”*…* Si sabemos que un registro deja de estar “activo” no se hace necesario borrar los datos del mismo.

# ***Super Queries (no es un término oficial)***

Sobre cómo utilizar condicionales *(filtros)*, que generalmente van en la sentencia *Where,* con las columnas.

Si queremos pasar, *iterar,* por todos los registros de una tabla (para evaluar registro por registro, uno por uno), podemos asistirnos de la función de agregación ***Count.*** Ahora, *¿Qué motivos tendríamos, además de contar los registros de una tabla, en iterar y evaluar cada uno de los registros de la tabla en cuestión?* Pues bien, esto sirve para hacer uso de estructuras de control o *condicionales*, tipo *sentencia IF*, para evaluar justamente la validez lógica de una condición pasada sobre cada uno de los registros de una columna seleccionada. Lo dicho, la estructura de control ***IF*** se asiste de la función ***COUNT*** para poder pasar por todos los registros y evaluarlos. Luego, las condicionales se pasarán ahora vía columnas y no por medio de un *Where*.

Es decir, se busca ejecutar, de acuerdo con una condición lógica a evaluar, un grupo u otro de sentencias *(If-Then-Else)* sobre la totalidad de los registros de una columna (al iterar sobre cada una de sus filas) y saber cuáles registros cumplen con la condición evaluada. Posteriormente se determina, según los valores booleanos, con que acción proceder sobre aquellos registros que sí cumplen la condición evaluada (true) y con que acción proceder sobre aquellos que no (false).

Generalmente en *Queries* se trabaja solamente con sentencias de tipo *If*.

## ***Sintaxis***

Debemos tener claro primero qué columna, de qué tabla, vamos a seleccionar para evaluar cada uno de los registros de la tabla de interés por medio de iteraciones; esto, para saber si se cumple o no, sobre la totalidad de los registros de una tabla seleccionada, una condición pasada con la sentencia *If*;y, posteriormente, qué acción tomar sobre aquellos registros que cumplen con la condición evaluada (true) y sobre aquellos que no (false)*.*

Generalmente la columna seleccionada, para hacer las iteraciones sobre todos los registros de una tabla, es la columna correspondiente a la *primary key* porque nos garantiza que se evaluarán todos los registros de la tabla en cuestión. Esta es, usualmente, la primera columna seleccionada. Luego, la segunda columna pasada ya tiene que ver propiamente con la columna donde se evalúa una condición por medio de la estructura de control *IF*. Generalmente lo que se pretende correr con la sentencia *IF*, ante los registros que logren validar la condición pasada, es una función de agregación; es decir, ejecutar una operación matemática sobre aquellos registros que cumplen con la condición lógica evaluada (true); si no lo cumplen, simplemente no se corre nada (false). La **sintaxis** de todo esto se ve más o menos así:

**1ra columna: SELECT COUNT(*primary key\_columna*) AS … ,**

**2da columna: FUNCION\_AGREGACIÓN(IF(Condición\_lógica\_evaluada, True, False)) AS … ,**

**FROM *tabla que contiene a las columnas.***

*Nota:* La primera columna no necesariamente corresponde con la primera. **Veamos un ejemplo.**

Si **queremos contar sólo los libros que fueron escritos antes de 1950, cómo lo haría?** Bien sabe usted que esto puede hacerlo por medio de la sentencia *Where,* sin embargo, queremos hacer lo propio asistiéndonos de sentencias de tipo *If* sobre columnas (**dándole inteligencia a éstas, con lo cual se podrían hacer matrices**). Veamos la resolución de esta *query* vía sentencia *If* y luego con *Where*. **Ejercicio resuelto** **con sentencias *IF.***



Lo que nos dice la sentencia IF es que se sume una unidad **(SUM),** con cada iteración sobre cada registro, si (**IF)** se encuentra un registro donde el libro haya sido escrito antes de 1950 **(age < 1950).** Ahora, se sabe que solo se sumará una unidad por cada registro valido, que cumple con dicha condición, porque así ha sido determinado en nuestro valor booleano de tipo "true": **(1)**. Ahora, en aquellos registros que no cumplan con la condición lógica no se sumará nada, así fue determinado en nuestro valor booleano de tipo "false": **(0).** *Puede tomar como guía la estructura de sintaxis previamente escrita.*

*Resultado/Salida:*

../../../../../../Captura%20de%20pantalla%202022-02-28%20a%20las%206.30.01%20p.m..png

**Dejemos ahora el código resuelto por medio de la sentencia *Where.***

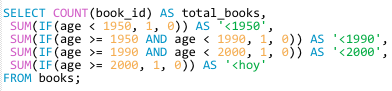
**

*Resultado/Salida:*

**

Ahora, lo que destaca especialmente a las “Super Query”sobre las Queries con filtros hechos con la sentencia *Where…* es que, en una misma *Query*, podemos mostrar y evidenciar por pantalla cuántos filtros deseemos por medio de *sentencias If y columnas* ilimitadas (**constituyendo así, incluso sin tener la intención, *matrices…* esto último es más evidente en los casos donde *agrupamos* la información con otra columna o atributo porque se podrían evidenciar divisiones o filtros tanto por columnas como por filas**); en cambio, **con una sentencia *Where,* no se puede hacer eso.** Con una sentencia *Where,* en caso que se pretendiera aplicar los mismos filtros que se pasan por columnas en una *Super Query*,no se evidenciaría la especificación de cada filtro aplicado en la visualización del *Query* cuando se muestrapor pantallay, además, nos limitamos a correr todos los filtros bajo una misma columna por *Query*; con lo cual, y es por ello que, no se evidenciaría la proporcionalidad que le corresponde a cada filtro ni tampoco se dejan señalados (si bien los filtros si los corre, la información se da de forma generalizada sin aclarar qué unidades corresponden a cada filtro pasado). Evaluemos lo anterior bajo el mismo ejemplo abordado para que se entienda mejor.

*Con Super Queries:*

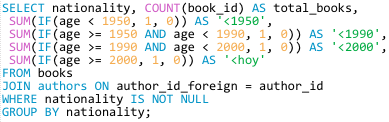
, *1 Query, 4 filtros en total.*

*Resultado/Salida:*

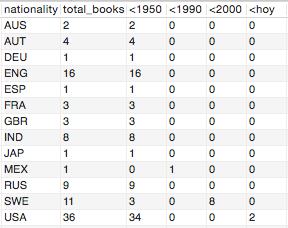


**nota**: Si quieres hacer uso de funciones de agregación + sentencias de control, las funciones de agregación siempre van por delante de las sentencias de control, así los opera SQL; de lo contrario, se marcaría error.

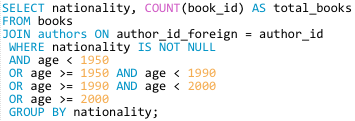
Tenemos 4 filtros, sobre 4 columnas, aplicados sobre una misma Query. Se evidencia la correspondencia, en términos de unidades, de cada filtro aplicado; es decir, la cantidad de unidades que corresponden a cada intervalo de tiempo evaluado (esto no se podrá notar con *Where* y más adelante lo evidenciaremos). **Ahora agrupemos estas unidades de libros, en estos 4 intervalos de tiempo, según nacionalidad del autor.**



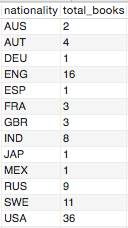
*Resultado/Salida:*



*Ahora ejecutemos el mismo Query haciendo uso de la sentencia Where, en vez de usar una "Super Query".*



*Resultado/Salida:*



**Conclusión:** Con una “Super Query” realmente sí se evidenciaría que podríamos filtrar/dividir tanto por nacionalidad como por intervalos de tiempo (en este caso puntual); en cambio, con un *Query* normal, sólo se evidenciarían las divisiones o filtros por nacionalidad.