**SQL**

**Breve historia de SQL**

**SQL** (*Structured Query Language* - Lenguaje de consulta estructurada) es un lenguaje que se basó en 2 principios fundamentales:

1. Teoría de conjuntos.
2. Álgebra relacional de Edgar Codd (científico informático inglés que evoluciono la teoría de conjuntos).

⠀  
**SQL** fue creada en 1974 por IBM. Originalmente fue llamado **SEQUEL**, pero posteriormente se cambió el nombre por problemas de derechos de autor con otra compañía. Mas adelante **SQL** se convertiría en un lenguaje estándar que unificaría todo dentro de las bases de datos relacionales, se convierte en una norma ANSI o ISO; por lo que, la gran mayoría de sub-lenguajes o manejadores de bases de datos relacionales tratan de la misma manera la sintaxis del lenguaje SQL (Oracle, MySQL, PostgreSQL,…).

**Álgebra relacional**

El **álgebra relacional** estudia básicamente las operaciones que se pueden realizar entre diversos conjuntos de datos.

⠀  
No confundir las **relaciones del álgebra relacional** con las **relaciones de una base de datos relacional**.

* Las relaciones en una base de datos se refieren a cuando, básicamente, unes dos tablas.
* Las relaciones en álgebra relacional se refieren a una tabla; o, para ser más precisos, a la interacción que haya con los conjuntos de datos dentro de una misma tabla.

La diferencia es conceptual: Las tablas pueden tener filas o tuplas repetidas pero en el álgebra relacional cada relación no tiene un cuerpo, no tiene un primer ni último *row* (esto debe abordarse de manera más detallada, posteriormente, para su mejor entendimiento).

**Tipos de operadores en Álgebra relacional:**

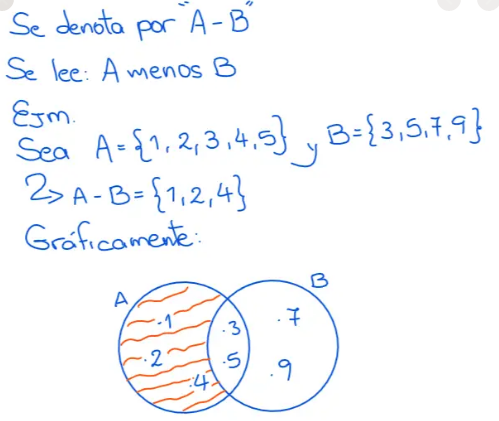
* **Operadores unarios.-** Requieren una tabla o relación para funcionar o correr. *Ejemplos*:

- **Proyección (π):** Haciendo un símil con SQL equivale al comando ***Select***. Saca o extrae, *proyecta,* toda la información de un número de columnas o atributos. Lo dicho, esto sin necesidad de hacer una unión con una segunda tabla o relación. *Simbología de la proyección:* ***π***.  
  
**π**<Nombre, Apellido, Email>(Tabla\_Alumno).  
⠀  
- **Selección (σ):** Haciendo un símil con SQL equivale al comando ***Where***. Consiste en el filtrado de tuplas o filas que cumplan con cierta condición o valor buscado. *Simbología de la la selección:* **σ.**  
  
**σ**<Suscripción=Expert>(Tabla\_Alumno).  
⠀

* **Operadores binarios.-** Requieren más de una tabla para operar (específicamente dos). *Ejemplos*:

- **Producto cartesiano (x):** Toma todos los elementos de una primera tabla, uno por uno, y los combina con todos los elementos de una segunda tabla.   
  
Docentes\_Quinto\_A **x** Alumnos\_Quinto\_A  
⠀  
- **Unión (∪):** Obtiene los elementos que existen, así sea una vez, en cualquiera de las dos tablas; es decir, todo elemento existente, ya sea en tabla A o en tabla B, los une. Une los elementos de tabla A con los de tabla B.  
  
Alumnos\_Quinto\_A **x** Alumnos\_Quinto\_B  
⠀  
- **Diferencia (-):** Obtiene los elementos que existen en una primera tabla seleccionada, pero que no existen en la segunda tabla evaluada en cuestión.  
  
Alumnos\_planExpertPlus **-** Alumnos\_planFree.

***Diferencia al detalle***La diferencia entre dos conjuntos A y B es el conjunto A \ B que contiene todos los elementos de A que no pertenecen a B.



Como ni 7 ni 9 están en el conjunto A, no podemos restar en A esos equivalentes que tenemos en B, pues no los tenemos (7, 9); es decir, se omiten 7 y 9 simplemente en dicha resta. En el conjunto de la “diferencia” se colocan solamente, entonces, los números que no se pudieron restar en A porque no tuvieron su equivalente en B; es decir: 1, 2 y 4.

***Repaso - Propiedad AS***

Usted recordará que 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 tablas); es decir, ***AS*** le pasaría un nombre temporal, cambiando temporalmente el nombre original, a una o varias columnas seleccionadas (o tablas) a la hora de querer hacer una consulta de ellas, como se esperaría, con ***Select***. 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 por 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,* y 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 instrumento, **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,***de la carpeta *Platzi,* explicado entrelas páginas 6 & 7 (por si desea saber más sobre el Objeto ***View***).

***Repaso - Propiedad Count***

***Select,*** también, nos permiten crear datos de manera temporal que nos pueden agregar información adicional; un buen ejemplo de eso sería combinar el ***Select*** con el uso de la sentencia ***Count(),*** ¿y qué hace ***Count()***?

La sentencia ***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 sentencia ***Count(),*** en connivencia con la sentencia ***Select,...*** es que en ***Count()*** se le pase, como parámetro, que cuente absolutamente todos (sin filtro alguno) los registros existentes para una tabla seleccionada con la sentencia ***From***; entonces, en este caso, ese “nuevo dato que se crearía 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 que se ha creado, el dato numérico, está sustentado por un nuevo atributo, campo o columna que también ha sido 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.***

***Un abreboca…***

***Funciones 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 a 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 o condicional definida 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.

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 las expresiones valiéndose de la propiedad WHEN; y, en caso de encontrar valida a alguna expresión lógica, dicha expresión devolverá por pantalla la salida especificada como verdadera para su propia condición evaluada (lo que se refleja por pantalla, como verdadero, es justo lo que se coloca después de la propiedad THEN).

La clausula ELSE, que complementaria a CASE, es opcional. ELSE devolverá su salida o valor definido como verdadero en caso que todas las condiciones WHEN, anteriores, hayan sido falsas.

Ahora, si todas las condiciones son falsas, pero tampoco existe la clausula ELSE, se devolverá NULL.

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

***Modelación***



***Repaso - 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 traer información que sea valiosa, conviene unir dichas tablas independientes; pues, nunca dejan de relacionarse y nos ayudaría, justamente, a *extraer más conclusiones e información adicional.*

En consecuencia, para lograr esa cooperación*,* 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.

***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 o se repitan en 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 uno de sus datos se relacione o se repita en 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 los datos que tenemos en común dentro de cada tabla, los datos que se repiten, que están están en ambas tablas; es decir, más puntualmente, estamos hablando de seleccionar los datos de la intersección entre los conjuntos. Si retomamos el ejemplo anterior estaríamos hablando de seleccionar los 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 en español ***Join interno.***



**3. Unión o Full Outer Join:** Cuando hablamos de “Unión”, estamos definitivamente hablando de unir ambas tablas, tanto ***A*** como ***B,*** por ejemplo; es decir, acá se seleccionarían y se unirían la totalidad de los datos, sin excepción alguna, de ambas tablas. *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 repitan en cada una de ellas; es decir, los datos que tengan en común ambas tablas, tanto la tabla ***A*** como la tabla ***B,*** por ejemplo. 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 esto, si lo conectamos 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 existen las dos entidades dentro de un mismo registro… sin ignorar que esto será evaluado registro por registro)***. Esto último es lo más importante. Y acuérdese siempre de la relación que tengan ambas entidades; pues, serán fundamentales para las conclusiones que podríamos tener para las Queries que hagamos.

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:





***Ahora, evidenciemos el mismo caso anterior pero 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; pero, hay muchas veces que es necesario utilizar ***Left*** o ***Right*** para que no te importe uno de los lados y siempre traigas todos los datos de un lado de la ecuación.

***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:





***Por último, 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***: 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*** 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*

**Ejemplo 2.** 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*** en los casos en los que se desee visualizar cadenas de texto (*strings*), dentro de todos los ***registros*** para un ***campo (o varios),*** que lleven consigo alguna palabra en especifica (que las cadenas de texto tengan); 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***, posteriormente se abre comillas *“”*,y se asiste de símbolos de porcentajes antes y después de la escritura, dentro de las propias comillas ***(“% ... %”)***, 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. Conclusión: 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.** 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 la palabra ***“escandalo”*** queremos que 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.** 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 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.***

Puntualmente, **ejemplo 5,** 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 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 detallada) 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:***

***,... lo saca?* Acá sólo importa que el mes sea “04” (sin importar año ni 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, o pasado, ningún tipo de valor (no han sido definidas). Este tipo de celdas ***null*** reciben un tratamiento particular, digamos, para hacer consultas de ellas,… se asisten de la sentencia: ***NULL (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, donde ***usuarios\_id sea null.*** Tal así:





*La base de datos entiende que le estás preguntando donde el usuario es “nulo”; es decir, trae todos los registros donde el usuario es nulo (is null).*

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 de la sentencia ***IS NOT NULL (que no están vacías).*** 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 el nombre que 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.** Queremos ver todos los registros de la tabla ***users*** donde el campo ***name*** sea igual a ***“Israel”, “Laura” o “Luis”****.*

******

***AND***

Ahora, 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 otra 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***

Esta nueva sentencia, si bien no es tan común, también es muy usada. Tiene que ver, como ya se imagina, con ***agrupar.*** La sentencia ***Group*** ***by*** lo que hará es indicarle a la base de datos qué criterios o valores (datos), por separado, nos va a arrojar según un campo o columna seleccionada,… para agrupar cierto número de registros en ella; como se dijo, esto a partir de seleccionar desde qué campo o columna se verán los registros, ahí agrupados, según un criterio pasado en ***Select*** (que, entre otras cosas, suele coincidir con la columna o campo seleccionado en ***Group by***). Por lo general, ***Group by*** trabaja de la mano de la sentencia ***COUNT();*** pues, muchas veces la intención de *agrupar* registros iguales, por separado, consiste en saber una totalidad numérica de ellos (los cuenta) para cierta columna pasada en ***Select***, según *x* o *y* parámetro, criterio. El criterio o parámetro suelen ser los distintos valores o datos con los que ha sido definido el campo o columna seleccionada con ***Select;*** es decir, **se me hará un conteo de los registros que hay para un determinado campo o columna seleccionada, sí; pero, haciendo el conteo de registros, por separado, para cada uno de los posibles valores que asume el campo o la columna en cuestión.**

Lo siguiente: cuando se trabaja con ***COUNT(),*** lo que se crea es una nueva columna, con ese mismo nombre ***“COUNT()”;*** que, justamente, nos arrojará el resultado de un conteo; en este caso, el conteo de registros para un campo o columna seleccionada con ***Select***. Ahora, cuando el conteo es sobre el **total** de los registros, de dicho campo seleccionado, hablamos de ***COUNT(\*);*** el criterio sería: ***selecciona y cuenta todos los registros del campo 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 “***COUNT(\*)”*** en cuestión***.*** Veamos todo esto, en su conjunto, con 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***, dicha información en la misma columna existente ***estatus.***

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





Hemos contado la cantidad de registros que hay en cada uno de los valores 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 la totalidad de los datos o valores que tiene 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\_publicación***. Ésta última, para efectos prácticos, la identificaremos ahora 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 nuestro filtro por años (YEAR); por lo cual, entonces, nos tocaría 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 uno de los valores que tenga el campo seleccionado; es decir, refleja toda la información de los registros del campo seleccionado, tal como lo hace un *Select,* pero con *Group by* se distingue de manera particular a cada valor del campo en cuestión. Ej.: No se cuentan todos los posts bajo un numero total, sino, se cuentan por mes.**

***Varios criterios en Group by***

Usted recuerda lo anterior: *“La sentencia* ***Group******by*** *lo que hará es indicarle a la base de datos qué criterios o valores (datos) nos va a arrojar, de un campo o columna seleccionada, para agrupar cierto número de registros en ellos; esto a partir de seleccionar desde qué campo o columna se verán los registros, ahí agrupados, según un criterio pasado en* ***Select*** *(que, entre otras cosas, suele coincidir con la columna o campo seleccionado en* ***Group by****)*”... Pues bien, *pueden ser más de un criterio los pasados* tanto para el ***Select*** como para el ***Group By;*** es decir, para que se organice o se agrupe la información según los datos o valores de más de una columna; como estamos trabajando con Count(\*),... para que se cuenten los registros; más precisamente, **haciendo el conteo de los registros por cada valor, separadamente, de cada una de las columnas seleccionadas**; en este caso, el conteo de la cantidad de posts publicados para cada uno de los valores que contemplan los campos o columnas pasadas como criterios en el ***Select*** y en el ***Group By***.

Entonces, cada criterio 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;* es decir, el filtro ahora está más elaborado.**

En resumen, cuando queremos que el conteo sea dividido entre los valores de sus registros (los registros de los campos agrupados, es decir, los que llamas en *GROUP BY*) y no sea un conteo totalizador que no discrimine o no agrupe, no cuente por separado, los valores de dichos registro… Nos asistimos de Group By **(esta es quizás la mejor explicación del recurso Group By en este contexto, asistiéndose de *COUNT()…* Téngalo en cuenta).** Tenga en cuenta además que ***COUNT()*** pasaría a ser un nuevo campo o columna dentro de la entidad o tabla con la que esté trabajando; la cual, al principio, hace un conteo de todos los registros de la tabla (no de un campo como tal, no).

**Ahora, algo importante:** Lo dicho, *COUNT()* no cuenta los registros de un campo en particular, no, realmente cuenta los registros que haya en la totalidad de la tabla que se genere en el Query. El conteo que hace *COUNT(),* en principio,se refleja como un total sobre una sola celda (el conteo de todos los registros que arroje la tabla consultada); sin embargo, como estamos *agrupando,* este conteo se refleja de forma dividida y se dividen los conteos por cada valor que tengan los registros de las columnas *agrupadas;* en este último caso, los campos agrupados son: “estatus” (sus valores: activo e inactivo) & “post\_month” (sus valores: cada mes del año).

**Ahora, lo que hicimos con la función *Count(),* en relación a *Group by*,lo podemos hacer con cualquier función matemática; ya sea para *sumar, promediar, maximizar, minimizar, etc.* Podemos seguir agrupando los resultados de las funciones matemáticas por cada valor, los que hayan en el total de registros, de los campos seleccionados para ser agrupados.**

***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 *que te ordene los datos por x o y criterio y/o campo*. 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 todos los registros de la tabla ***posts*** (sin hacer ningún tipo de filtro en ella, aclaremos); 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 ***ASC. Probemos ambas propiedades de Order By: ASC & DESC.***

**Ordenamiento de registros 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:

**Ordenamiento de registros 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).





*También* existe el **Ordenamiento de registros 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****.*





Ahora, 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.

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.

***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, o sin la interferencia, de la sentencia ***Group by***; dicho eso, ***Where*** no puede trabajar de la mano con ***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 uno de los valores que asumen las columnas de referencia ***fecha\_publicacion*** & ***estatus,*** con la condición de que se muestren solamente los registros donde se publicó ***más de un posts*** (para cada uno de los valores de las columnas ***fecha\_publicacion*** & ***estatus*** donde se coincida, conjuntamente y en un mismo ***registro,*** con 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 maravilloso *arte* de saber convertir una pregunta en un Query.

***La estructuración de cómo convertir una pregunta en 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 Columna, 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 Columna.

En ***Where,*** luego de que ya has seleccionado ***(Select)*** lo que quieres mostrar y de dónde ***(From)*** lo vas a seleccionar; 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);*** y, en conjunto con ***Limit,*** se pueden establecer ***Tops (Top 10, por ej.),*** ya sea de los *peores* (últimos) registros o de los *mejores* (primeros) registros.

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***).