**Clase 05 - Consultas y Subconsultas**

Union

Definición

El operador UNION **combina los resultados de dos o más consultas en un único resultado** que incluye todas las filas que pertenecen a todas las consultas que aparecen.

Las consultas se ejecutan por separado , concatenando luego el resultado de cada una.

Para que la Union sea compatible la cantidad de columnas en cada consulta o tablas debe ser la misma y del mismo tipo de dato.

Ejemplo:

SELECT id\_game, name, description, id\_level, id\_class

FROM game

WHERE id\_level = 1

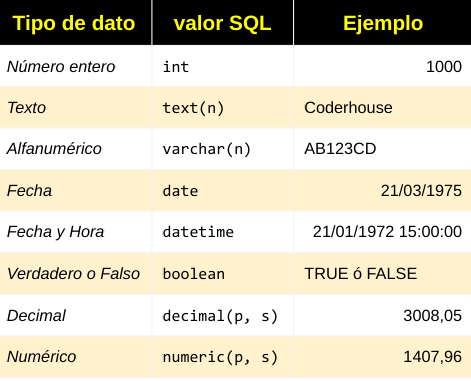
UNION

SELECT id\_game, name, description, id\_level, id\_class

FROM game

WHERE id\_level = 2;

Tipos de Datos



El tipo de dato a definir en un campo es un punto clave; al gestionar nuestra DB debemos establecer reglas de contenidos claras para cada uno de los campos en las tablas.

## Uso de textos

En los datos de cadenas de caracteres discriminamos aquellos relacionados a solo texto de los alfanuméricos.

Tanto **text** como **varchar** cuentan con una **(n)** que los precede. Dicha **n** equivale a que nosotros podemos especificar un valor numérico, el cual oficiará de limitante a la cantidad de caracteres que dichos campos deben almacenar. Si bien podemos obviar poner un número alto, lo más efectivo es siempre poner un limitante.

La limitación de datos nos ayudará a que las bases de datos no solo estén normalizadas, sino a que no se saturen con contenido librado al azar.

## Fechas

Originalmente existía el tipo de dato **datetime**, y no **date** y **time** por separado. Estos últimos dos llegaron al universo de las bases de datos hace menos de 2 décadas.

Datetime es muy útil para aquellas tablas que guardan información del tipo log. Configurando el campo datetime de forma predeterminada cuando se genere un registro en éste; dicho campo no necesitará recibir un valor ya que podrá generarlo automáticamente desde el servidor de base de datos.

## Datos numéricos

Por el lado de los datos numéricos, la oferta es bastante amplia en SQL.

Tenemos los datos enteros(número sin decimales) que podemos definirlo utilizando **int.**

Pero también, tenemos la posibilidad de seguir con números enteros de mayor o menor tamaño, utilizando **smallInt** y **bigInt**. Además de **integer**, **numeric** y **decimal**. En el caso de estos dos, si ponemos un 0 (cero) en su segundo parámetro, técnicamente tendríamos un valor entero también.

Operador LIKE

**LIKE** es denominado un **Operador Lógico**, usado para determinar si una determinada cadena de caracteres coincide con un patrón específico.

En el mundo de la programación en general, existe un paradigma denominado caracteres regulares o expresiones regulares, además de los famosos caracteres comodín. Estos nos permiten definir algún dato faltante que no conozcamos o que pueda coincidir con determinado patrón. Allí es donde se aplica el comodín **%**.

También contamos con otros caracteres comodines, los cuales pueden ser “**\_**” o “**\***”, variando estos de acuerdo “al sabor” de la base de datos que utilicemos.

Ejemplo:

SELECT \*

FROM game

WHERE name LIKE ‘FIFA%’;

## Like con corchetes[ ]

Dentro de las expresiones regulares, el uso de corchetes **nos permite que el resultado de la búsqueda se limite a un rango inicial determinado de caracteres.**

Ejemplo:

SELECT \*

FROM game

WHERE name LIKE ‘[A-B]%’;

En este ejemplo, el resultado de la búsqueda estará basado en aquellos registros del campo **name**, que coincidan en su inicio con las letras contenidas entre **A** y **B**: **[*A, B*]**. Prestemos atención que en este caso, el caracter % se usa al final y por fuera de los corchetes en cuestión para indicar que incluya cualquier cantidad de caracteres.

## Exclusión de caracteres

Aprovechando el uso de corchetes, también nos podemos ocupar de excluir de los resultados determinados caracteres. Para ello debemos usar **[^]**

SELECT \*

FROM game

WHERE name LIKE ‘[**^DV**]%’;

Subconsultas SQL

Las subconsultas son, básicamente, una segunda consulta completa que debe ser definida a continuación de las cláusulas **WHERE** o **HAVING** dentro de una sentencia general de SQL.

Reglas a tener en cuenta

* La subconsulta debe ir entre paréntesis.
* La subconsulta debe tener una sola columna o expresión.
* No podemos utilizar **BETWEEN** o **LIKE** en la subconsulta.
* No debemos colocar la cláusula **ORDER BY** en la subconsulta.
* Otras cuestiones más con **UPDATE** y **DELETE**, que veremos oportunamente cuando abordemos dichos temas.

## Forma de ejecución

Cada subconsulta que escribamos de forma anidada a otra consulta o subconsulta, será la primera que se ejecute. El orden de ejecución de una consulta que contenga una o más subconsultas será desde la subconsulta de mayor profundidad, hacia afuera. Por lo tanto, aquí, encontraremos una similitud con las operaciones matemáticas de complejidad media o alta, donde siempre debemos resolver primero las operaciones matemáticas encerradas en los paréntesis o corchetes internos donde, una vez conseguido dicho resultado, iremos resolviendo el resto de subconsultas externas.

### Ejemplo:

SELECT id\_system\_user, last\_name

FROM system\_user

WHERE id\_user\_type = (SELECT max(id\_user\_type) FROM user\_type);

El resultado que obtenemos en esta consulta será solamente el **id\_system\_user y last\_name**, pero solamente de aquellos usuarios que son del tipo cuyo id\_user\_type es el máximo. Así, facilitamos la consulta que debe vincular a una tabla alternativa, sin que debamos tener presente cuál es el **ID** que distingue al **máximo tipo** que estamos buscando y que se relaciona con la tabla de **system\_user**.

Pensemos con esto que, si tenemos elaborado un sistema de gestión que permite consultas de este tipo, facilitamos a que el usuario operador de dicho sistema pueda buscar la información utilizando un parámetro acorde a su saber y no relacionado con un identificador interno propio.

Otro ejemplo podría haber sido:

SELECT id\_system\_user, last\_name

FROM system\_user

WHERE id\_user\_type = (SELECT id\_user\_type FROM user\_type

WHERE description like 'tempo%');

Seleccionando todos los usuarios cuyo tipo incluye en su descripción “tempo”, con los datos que actualmente poseemos en la DB GAMER no tenemos inconvenientes porque contamos con un solo registro en user\_type con esas características pero si contáramos con más registros la consulta daría error.

Tenemos que estar seguros que al utilizar los operadores de comparación (=, <, >, etc) el resultado de la subconsulta es un único valor.

# Subconsultas internas

También podemos abordar subconsultas dentro de una misma tabla.

Ejemplo:

SELECT id\_system\_user

FROM vote WHERE value = (SELECT FLOOR(AVG(value)) FROM vote);

Podemos también sumar otros operadores combinados con **WHERE**, como ser **<, > =>, <=**, para recuperar un conjunto de valores más específico todavía.

# Ordenamiento de subconsultas

## Order by

La **sentencia ORDER BY** puede ser utilizada dentro de consultas con subconsultas, teniendo en cuenta que dicho ordenamiento debe realizarse en la consulta principal.

Ejemplo:

SELECT id\_system\_user, last\_name

FROM system\_user

WHERE id\_user\_type = (SELECT max(id\_user\_type) FROM user\_type)

ORDER BY last\_name ASC;

Importante:La flexibilidad de SQL nos permite realizar cualquier consulta, siempre que los diseños estén correctamente normalizados.

## 

**SQL PERFORMANCE TUNING**

La optimización de consultas es el proceso de escribir la consulta de manera que pueda **ejecutarse rápidamente.**

Por ejemplo, si nuestra **tabla game** tiene 100000000000 filas de datos y necesitamos averiguar la fila donde el **id\_game** es 77 , existen tres formas diferentes de encontrar el resultado:

* **O(1):** Índice hash en id, o resultado almacenado en caché en id\_game= 77 de una consulta anterior.
* **O(n) :** Sihacemos un escaneo completo de la **tabla game** y miramos todas y cada una de las filas, entonces tomaría mucho tiempo.
* **O(log(n)) :** Si ordenamos la tabla y hacemos una búsqueda binaria en esta tabla, entonces necesitamos alrededor de 36 búsquedas (la base de registro 2 de 100000000000 es ~ 36) para averiguar nuestro valor, pero ordenar la tabla llevaría un tiempo (depende totalmente del optimizador del motor sql).

La opción con menor tiempo de respuesta sería **O(1).**

**BUENAS PRÁCTICAS PERFORMANCE TUNING**

Algunas buenas prácticas para mejorar el rendimiento de nuestras queries son la siguientes:

1. **Usar SELECT \* FROM solo si es necesario**

No utilicemos ciegamente SELECT \* en el código. Si hay muchas columnas en la tabla, se devolverán todas, lo que ralentizará el tiempo de respuesta, especialmente si envía el resultado a una aplicación front-end.

Es recomendable escribir explícitamente los nombres de las columnas que realmente se necesitan.

1. **Usar la cláusula ORDER BY sólo si es necesario**

Si deseamos mostrar el resultado en la aplicación frontal, dejar que ORDENE el conjunto de resultados. Hacer esto en SQL puede ralentizar el tiempo de respuesta en el entorno multiusuario.

1. **Usar UNION ALL en vez de UNION**

La cláusula UNION ALL responde más rápido que UNION.

UNION ALL busca filas duplicadas, mientras que la declaración UNION lo hace independientemente de si existen o no.

1. **Usar la cláusula EXISTS donde sea necesario**

Si deseamos comprobar la existencia de datos, no es recomendable utilizar:

**IF** (**SELECT** **COUNT**(\*) **FROM**  TABLA **WHERE** Columna=**’Algun Valor’**)>0

En su lugar, podemos usar la cláusula EXISTS:

**IF EXISTS** (**SELECT** **COUNT**(\*) **FROM**  TABLA **WHERE** Columna=**’Algun Valor’**)

**Es más rápido en tiempo de respuesta.**

**MYSQL ANALYZER**

MySQL Query Analyzer permite a los desarrolladores y analistas mejorar rápidamente el rendimiento de sus aplicaciones de base de datos, mediante la supervisión del rendimiento de las queries.

A lo largo de la cursada, prestaremos atención a buenas prácticas que ayudan para mejorar el rendimiento de las queries.

## 