STRUCTURED QUERY LANGUAGE

(SQL)

Una **base de datos** es cualquier cosa que agrupe información y que esta **tenga un sentido**.

Una base de datos puede ser:

* Una lista de supermercado
* Una lista de contactos
* Una lista de tareas
* Un listado de alumnos
* El listado de usuarios de Instagram o Twitter

Lo importante de una base de datos es que, obviamente, tenga datos y puedan ser relacionados entre sí, por ejemplo: un usuario con sus tweets o su post de Instagram.

La base de datos que usamos recurrentemente es nuestra **memoria**. Nosotros tenemos una forma de interactuar con nuestra memoria, pero también tenemos formas de interactuar con la memoria de otra persona. Y el que finalmente estaría encargado de poder **gestionar** todos los datos que se encuentran dentro de nuestra base de datos (nuestra memoria) sería uno mismo.

Otra forma que tenemos para almacenar una base de datos es una **hoja y un lápiz**.

Y, obviamente la mejor alternativa, son los **computadores;** ya que se puede acceder a la información más rápido y pueden almacenar grandes cantidades de datos. Para que nosotros podamos acceder a datos, dentro de una base de datos, necesitamos un Sistema Gestor de Base de Datos (Relational Database Management System), un software para poder administrar dichas bases de datos. **MySql, Postgres, MariaDB, Oracle, Informix.** El objetivo de todos estos softwares que vemos acá es que estos nos entreguen un acceso fácil a la base de datos; **se encargan de la seguridad y la integridad de nuestros datos.**

También tienen otras funcionalidades como poder efectuar **respaldos** de nuestros datos, **importar y exportar** datos, manejar la **concurrencia**, etc. Con **concurrencia** queremos decir cuando más de una conexión está intentando acceder a la base de datos, por ejemplo:

Usuario: chanchitofeliz; en este momento hay un usuario que quiere acceder a este usuario de chanchitofeliz. Pero también hay otro usuario (conocido como admin de nuestra plataforma), el cual quiere efectuar un DELETE (ósea quiere eliminar a chanchitofeliz). Si es que nosotros intentamos hacer esta operación al mismo tiempo, de LEER el usuario de chanchitofeliz y ELIMINARLO **al mismo tiempo**, **se generan errores de consistencia en los datos**, ya sea para quien los está rayando y para quien está escribiendo.

Existen muchas formas de controlar/manejar estas inconsistencias en los diferentes SGBD, pero una muy común es que cuando alguien está efectuando una operación de LEER, el SGBD se va a encargar de que esta operación se efectúe primero (mientras tanto se va a encargar de bloquear los registros, en particular este, para que no se puedan realizar otras operaciones). En el caso de que se haya terminado la operación de LEER va a pasar a la cola la siguiente operación, en este caso, la de DELETE.

Otro problema sería cuando dos usuarios, al mismo tiempo, están tratando de modificar datos del mismo registro.

De todos estos problemas y más, se va a encargar de solucionarlo nuestro SGBD.

**Otra tarea sumamente importante de la que se encarga nuestro SGBD es de PODER CONECTARSE con los distintos lenguajes de programación** (PHP, PYTHON, JAVASCRIPT). Las operaciones más importantes que vamos a ver en SGBD son:

* Create
* Read
* Update
* Delete

También conocido como **CRUD.** Las aplicaciones que vamos a crear **giran en torno a estas operaciones**. Para que nosotros podamos acceder a los registros y también realizar estas operaciones que tiene que hacer un SGBD, vamos a tener que escribir lo que se conoce como **query (consulta).**

Cuando nosotros estamos buscando algo en Google, estamos realizando una **consulta,** o cuando miramos la foto de la persona que nos gusta en Instagram, estamos haciendo una **consulta.**

**FACEBOOK→ SGBD → BASE DE DATOS**

**← ↑ ↓**

**↑ ← ←**

A través de un SGBD realizan consultan a la base de datos, para obtener la información, que se la devuelve al SGBD, que finalmente es devuelta a la app de Facebook y la podemos ver.

**AMAZON→ ver un producto**

**↑ ↓**

**←←←←← SGBD → BASE DE DATOS**

**↑ ←←←←←←←**

Al buscar un producto en Amazon, lo que hace la app es, mediante un SGBD acceder a la base de datos para obtener de la misma la información que estamos buscando (precio, cantidad, etc.) para luego retornar esta información al SGBD el cual nos mostrara dicha información mediante la app de Amazon.

Existen **dos grandes** mundos al momento de nosotros seleccionar un SGBD; tenemos los que implementan **SQL (bases de datos relacionales)** y los SGBD que no implementan SQL que se llaman **NoSQL.**

* Los softwares de gestión que implementan SQL suelen almacenar la información en forma de **TABLAS**, cuales tienen **FILAS** y también **COLUMNAS.**
* Las que no son SQL tienen múltiples formas de almacenar los datos; JSON, BSON, BLOB, key\_Value.

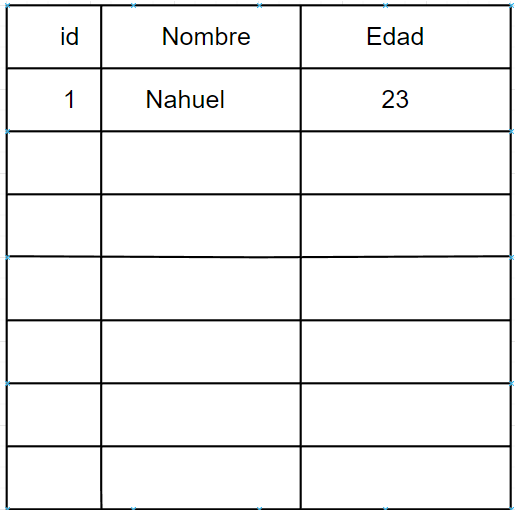
**TABLAS**

La forma de almacenar datos en SQL

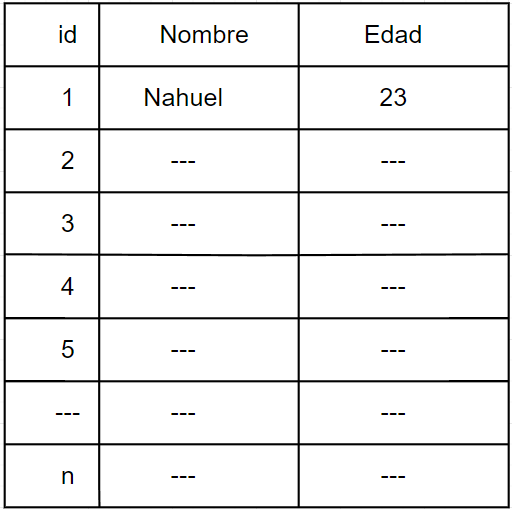
Estas van a tener columnas, en este caso tenemos la columna id, Nombre, edad. Las columnas **hacen referencia al dato mismo que se está guardando;** esto quiere decir que si mi nombre es Nahuel, guardare dicho dato en la columna Nombre, lo mismo con mi edad, si tengo 23 años, voy a guardar mi dato (edad) en la columna edad.



Y también tenemos la primera columna, que es la de id (Identificador Unico). Es un numero completamente único que se genera de manera automática cuando nosotros generamos un registro y también cumple con la característica de ser autoincremental.



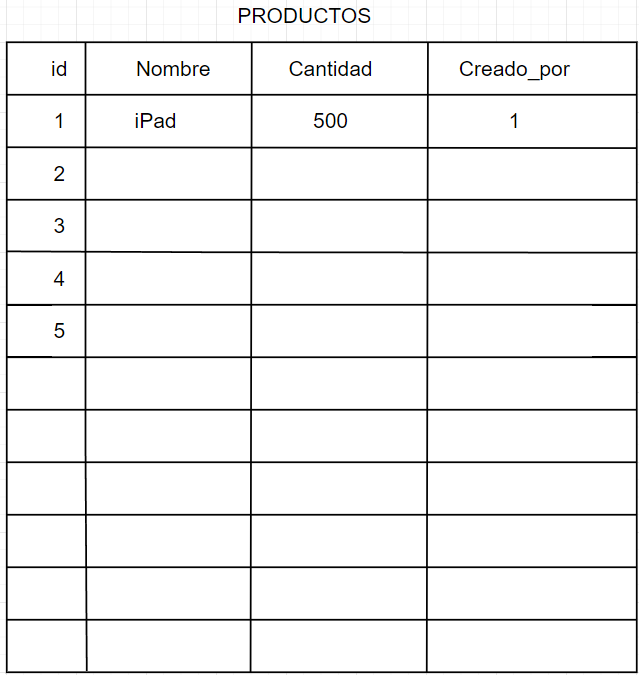
La siguiente vez que nosotros ingresemos datos dentro de esta tabla, el id que se le va asignar al nuevo registro es 2, con su nombre y edad, y así sucesivamente hasta llegar al último registro que hayamos ingresado el cual va a tener un identificador único de **n.**



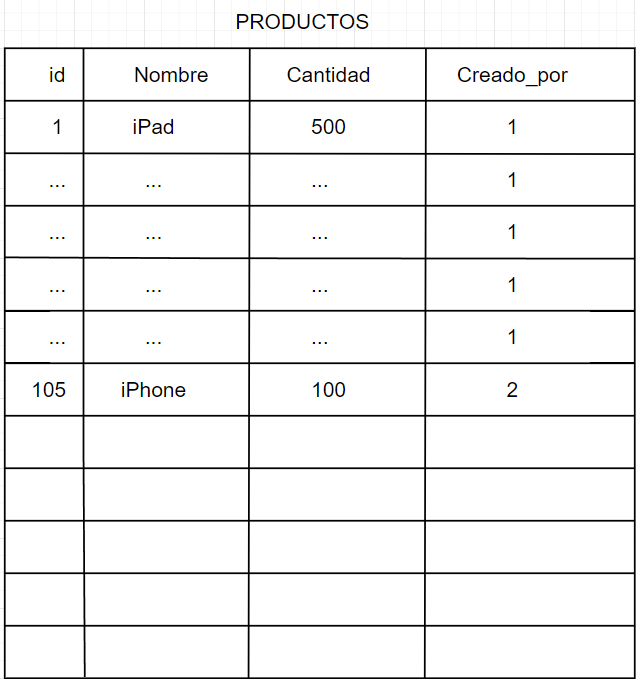
La parte vertical son las **columnas,** mientras que el conjunto de columnas horizantales (que representan la información de una persona) son los **registros**. Los valores que se van a encontrar en nuestras tablas, como, por ejemplo, Felipe (el cual se encuentra en la columna de nombre) se le conoce como **DATO**. 1, en la tabla id, **ES UN DATO**; 20, en la tabla Edad, **ES UN DATO**. **PERO** a toda esta tabla le asignamos el nombre de User, o tabla Usuario, es ahí cuando nosotros sabemos que los datos guardados en esta tabla hacen referencia a un usuario, y en ese entonces tenemos información. **Si nosotros no agruparamos estos datos de una manera que hicieran sentido solo tendríamos datos sueltos, pero no es hasta ahora cuando le asignamos un sentido, que pasan a ser información.**

El ejemplo que vimos antes era la tabla de usuarios, sin estar asociada a nada. Es la más importante que siempre se va a encontrar dentro de una base de datos, ya que mediante esta tabla vamos a poder asignar nuevos registros que se creen a alguien. Por ejemplo, en esta tabla “Productos”. Nosotros podemos tener varios usuarios que vayan a comprar, administradores y un montón de perfiles mas que nosotros podríamos asignar.

El que vamos a ver ahora es el tipo **administrador**, el cual puede crear productos dentro de un software.



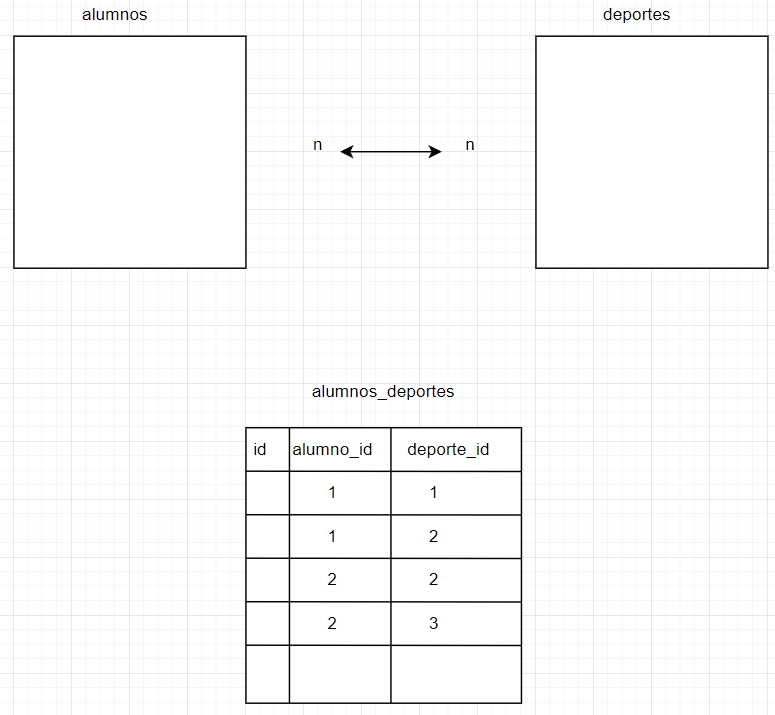
**El producto de nombre iPad, con cantidad de 500, fue creado por el usuario 1.** El usuario número 1 también podría crear un listado de productos que se van a encontrar también dentro de nuestras tablas. Pero eventualmente nosotros podríamos tener más de un administrador. Suponiendo que tengamos dos administradores, uno con el id de 1 y el otro con el id de 2, y en el caso que este último cree un producto, por ejemplo:



Nosotros para poder indicarle que este producto fue creado por el usuario id 2, nosotros, dentro de la columna “Creado\_por” vamos a pasarle el valor 2, indicando que este producto fue creado por el usuario id 2.

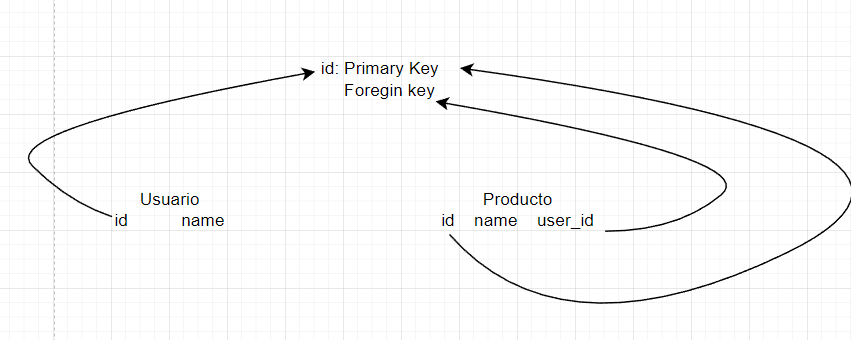
La razón por la cual le pasamos un id en lugar del nombre del usuario u otro dato del usuario, es porque los datos de este usuario eventualmente pueden cambiar, y también para **evitar duplicar información.** Ya que si utilizamos esta forma para poder asignarle un creador a un producto, lo que estamos haciendo es que en disco solamente se va a grabar el número 2 y no el no el usuario completo, de esta manera ahorramos espacio en disco duro. Y, finalmente, todas las bases de datos SQL se van a mover alrededor de modelos de tablas, guardar registros, registros en otra tabla y **asociarlos** de alguna manera. Esta forma que nosotros vimos es una relación **1🡪n (un usuario puede haber creado muchos productos, pero un producto solamente pudo haber sido creado por un usuario)**

Lo anterior visto fue la forma de relacionar una tabla con otra, pero solo con relación 1🡪n. En el caso de una relación, por ejemplo, con tablas de “**alumnos**” y “**deportes**”, donde un alumno puede estar inscripto en más de una rama deportiva. En este caso se crea una nueva tabla intermedia a la que le vamos a asignar el nombre de estas dos tablas juntas; “**alumnos\_deportes**”, y dentro de esta tabla, con su columna id, debemos agregarle 2 columnas mas: **alumno\_id** y **deporte\_id**



Esta relación se conoce como **n🡨🡪n** (muchos a muchos) donde un alumno, o varios alumnos pueden asistir a varios deportes, y un deporte puede tener muchos alumnos.

Las columnas que tienen el nombre de **id** se las conoce como **Primary Key (clave primaria).** Pero si este **id** (por ejemplo, en la tabla Usuario) se encontrará dentro de la tabla Producto para que nosotros podamos indicarle quien creo este producto en este caso se llama **Foreign Key (clave foránea)**

****

En la tabla **Usuario, id es clave primaria.** En la tabla de Producto tenemos **user\_id, que es nuestra clave foránea.** Pero, en la tabla producto también tenemos **id, que es la clave primaria de esta tabla**

En el caso de alumnos\_deportes, **id es PK (Primary Key), alumno\_id es FK (Foreign Key), como así también deporte\_id es FK (Foreign Key)**

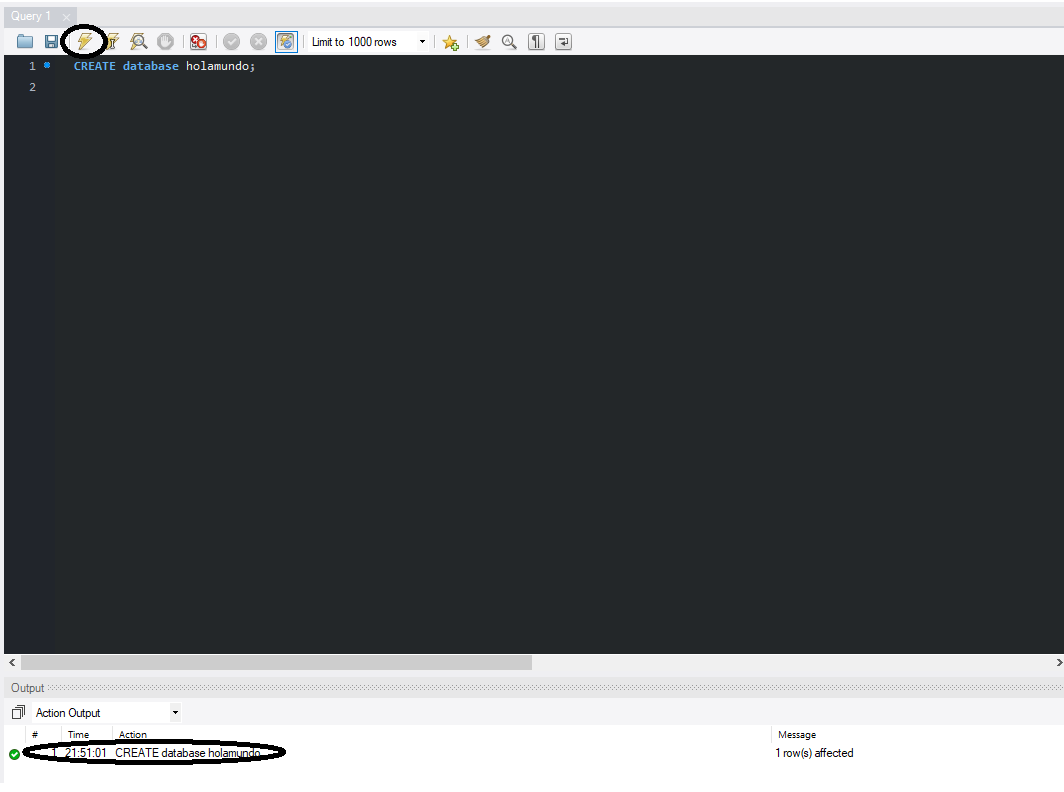
SISTEMA GESTOR DE BASE DE DATOS

MYSQL

Crear una base de datos:

**CREATE DATABASE holamundo;**

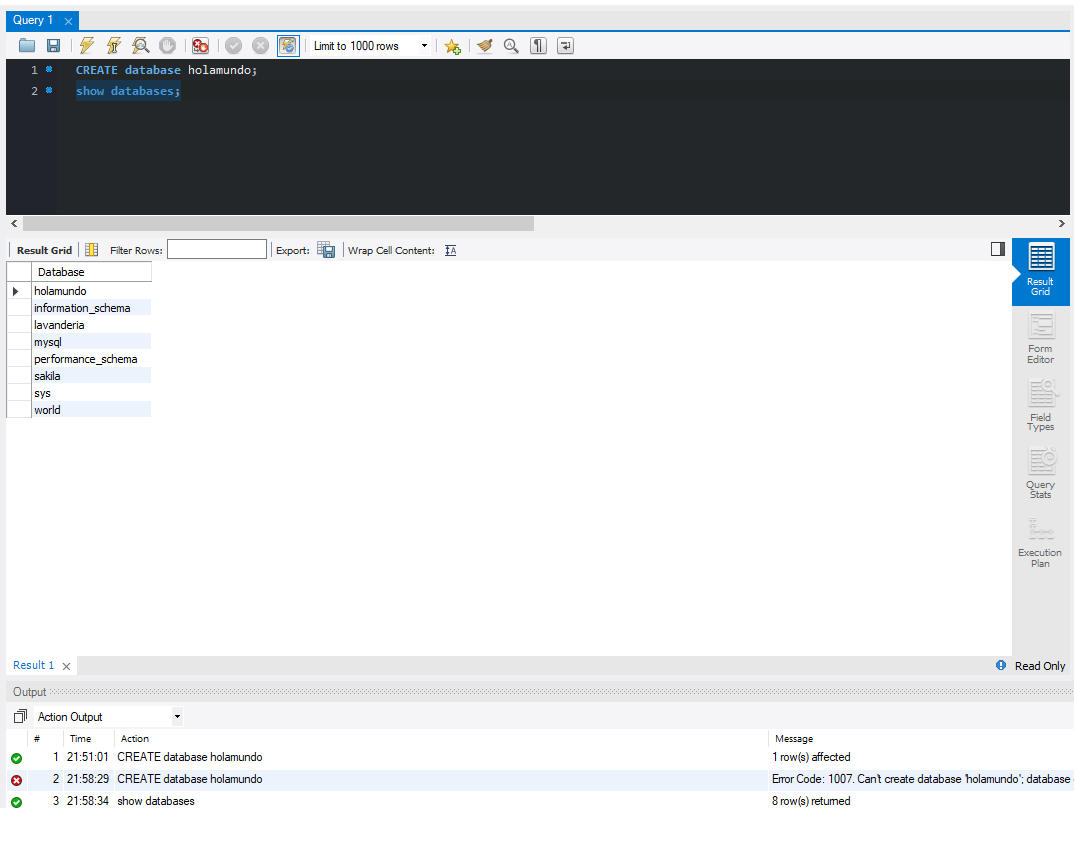
Luego pinchamos en el rayito para ejecutar; este se encarga de ejecutar todas las consultas que vayamos haciendo.



Para ver los registros de todas las bases de datos que existen y que MySQL está gestionando usamos el comando:

**show databases;**

Seleccionamos la sentencia (show databases;) y pinchamos en el rayito nuevamente para que abajo muestre todas las bases de datos que se encuentran creadas.



Crear Tabla:

**CREATE TABLE animales (**

**);**

Dentro de estos paréntesis pondremos las columnas que va a tener nuestra Tabla de animales. Y lo primero que necesitamos es un identificador (id) de manera que cuando nosotros agreguemos registros, sepamos que este tiene un valor único y de esta manera referenciar este mismo registro en otros lados.

**CREATE TABLE animales (**

**id**

**);**

Después debemos indicar el **tipo de dato** que este va a tener. Hay muchos tipos de datos, pero partiremos con:

**int = números enteros (0, 1, 2, 255, 400, 6000)**

**float = números con decimales (61.2, 40,32)**

**varchar = cadena de caracteres (‘hola mundo’, ‘chanchito’, ‘feliz’)**

**CREATE TABLE animales (**

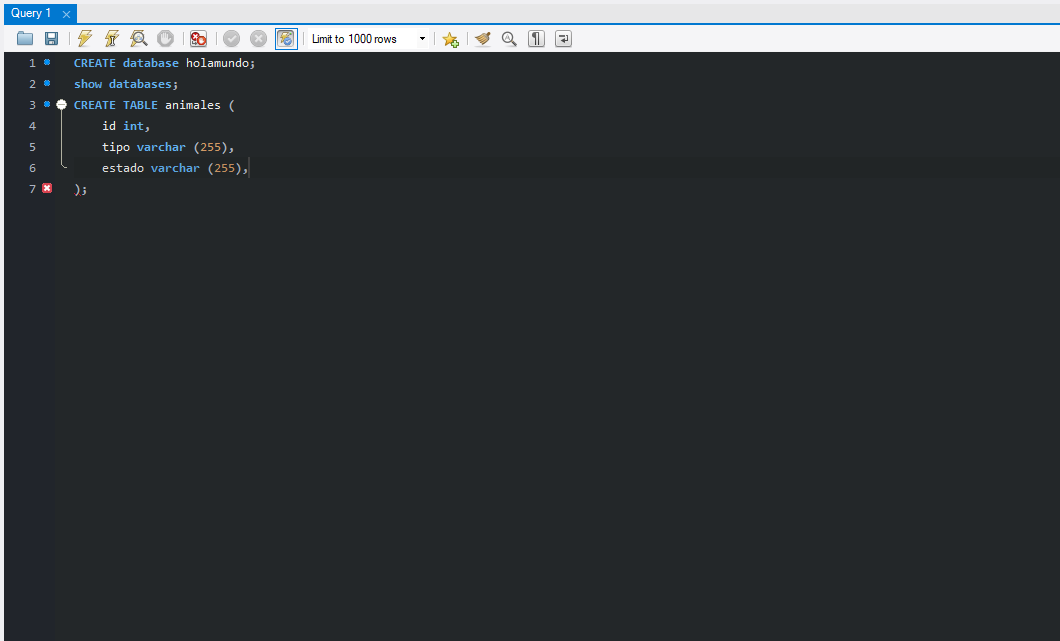
**id int,**

**tipo varchar (255),**

**estado varchar (255),** // al usar varchar, se debe poner entre paréntesis la cantidad de

caracteres a usar

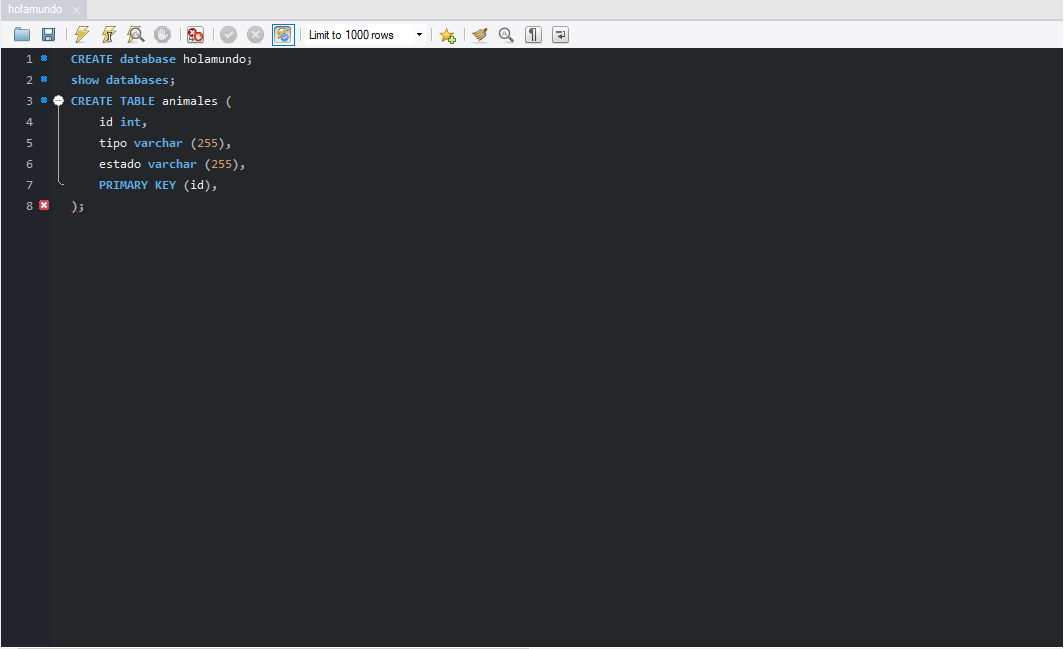
**);**

****

Habíamos hablado de que las tablas deben tener un **identificador único,** en este caso la columna **id de tipo entero,** pero debemos indicar que ES una PK.

Indicar Clave Primaria:

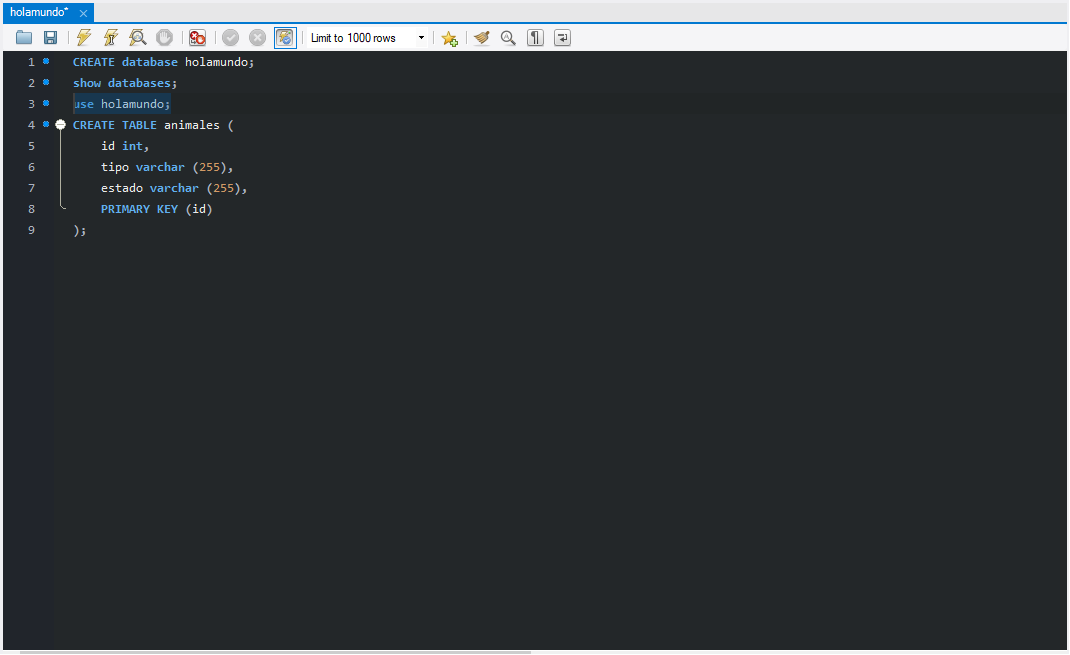
**PRIMARY KEY (id);**

****

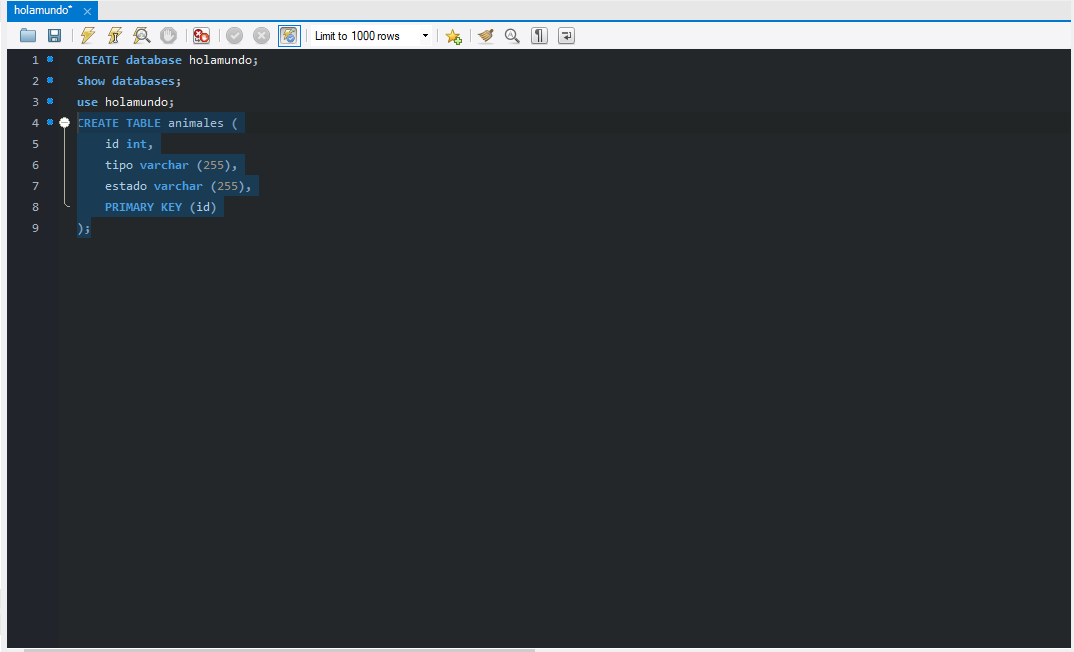
**IMPORTANTE: la , luego de identificar la Primary Key NO VA, por eso sale la cruz roja a la izquierda, bastara con solo borrar dicha coma luego de (id)**

**Pero antes de poder ejecutar nuestra consulta de CREATE TABLE, debemos indicar antes que Base de Datos tenemos que utilizar.** Si bien creamos la base de datos holamundo, la consulta no sabe qué base de datos utilizar.

**use holamundo;**

Seleccionamos esta línea con el mouse y ejecutamos para que asi el SGBD sepa que estamos usando la base de datos holamundo; 

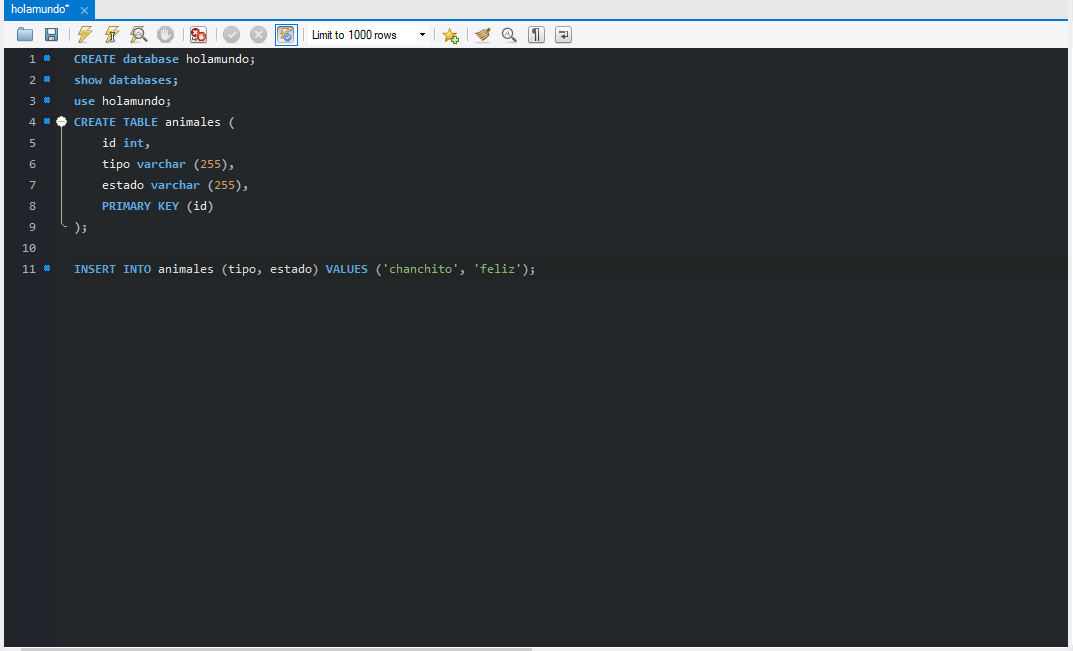
Y, ahora si, podemos ejecutar nuestra consulta de **CREATE TABLE.**



Insertar datos dentro de nuestra tabla:

**INSERT INTO animales (tipo, estado) VALUES ('chanchito', 'feliz');**

Insertamos dentro de la tabla **animales** y, dentro de las columnas **tipo** y **estado**, los valores de **chanchito** y **feliz**.



Sin embargo, nuestra consulta se encuentra incompleta. Habíamos mencionado en un comienzo que cuando insertamos datos y tenían un identificador único, estos cumplían con tener un valor autoincremental. Esto quiere decir que si llegáramos a insertar **INSERT INTO animales (tipo, estado) VALUES ('chanchito', 'feliz');** dentro de nuestra tabla, esto debiese generar inmediatamente un id único con el valor de 1. Pero esto no va a ocurrir, mostrara error:

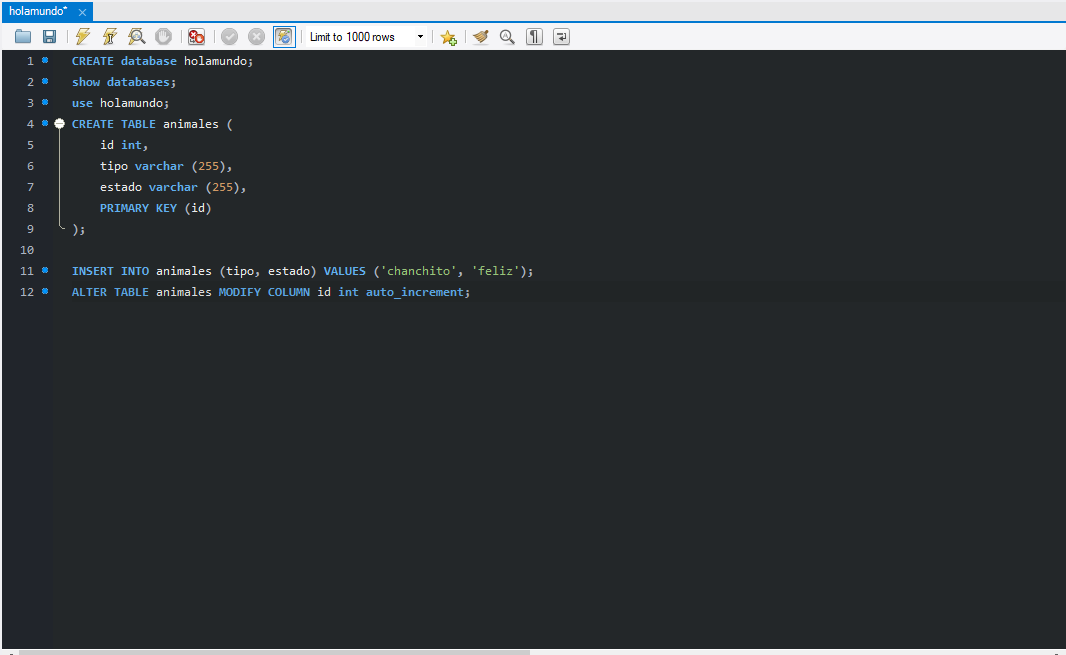


Y esto es porque nosotros le indicamos que el campo de **id** es una Clave Primaria y que es de tipo entero, pero nos faltó indicar una propiedad importantísima: la de **autoincrementar su valor.**

Ahora, **¿Cómo vamos a hacer esto? Porque nuestra tabla ya fue creada.** Aprovecho para mostrar otro comando para modificar una tabla que ya ha sido creada.



Indicamos la tabla que queremos “alterar”, la columna que debemos modificar, su tipo y hacerla auto\_increment (que autoincremente su valor)



Seleccionamos la consulta y ejecutamos:



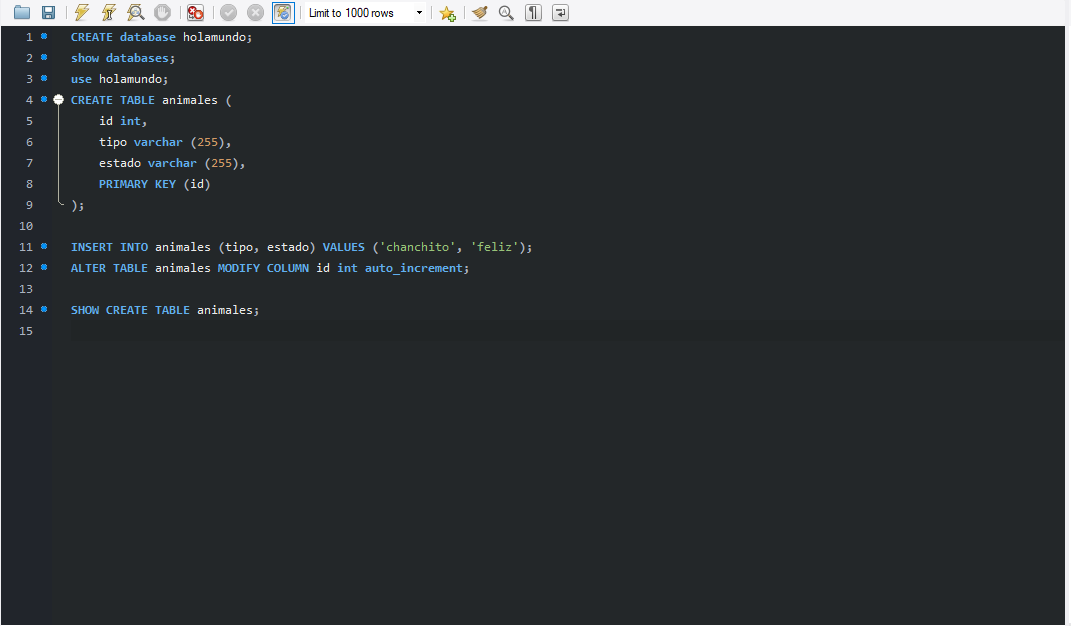
Seleccionamos la línea de INSERT INTO y la ejecutamos, y ahora si veremos que no mostrara error al agregar valores nuevos a las columnas TIPO, ESTADO:



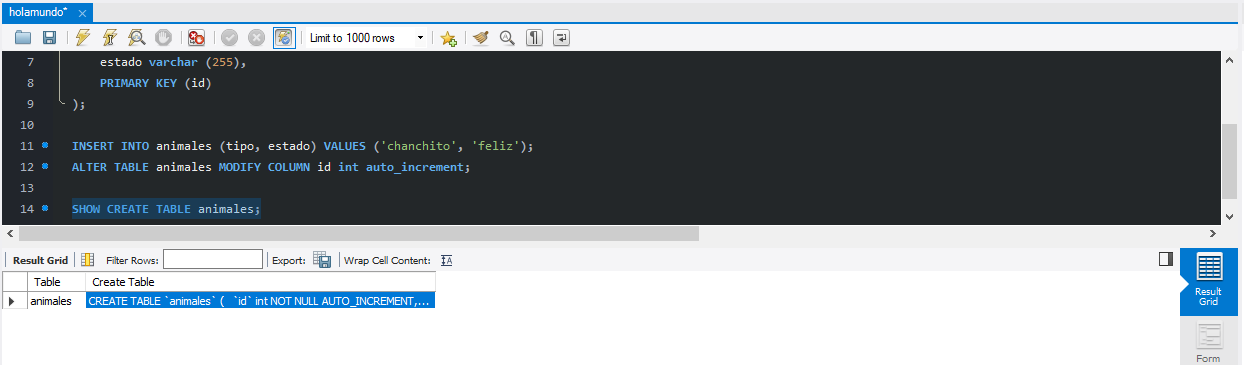
**¡PERO PROFESOR, USTED LO QUE HIZO ES MODIFICAR LA TABLA Y YO QUIERO SABER COMO CREAR UNA TABLA DESDE EL COMIENZO CON QUE ESTE YA VENGA CON EL VALOR DE AUTOINCREMENTAR!**

Tranquilo, enseguida te muestro como. Luego de nuestro **ALTER TABLE** ponemos:

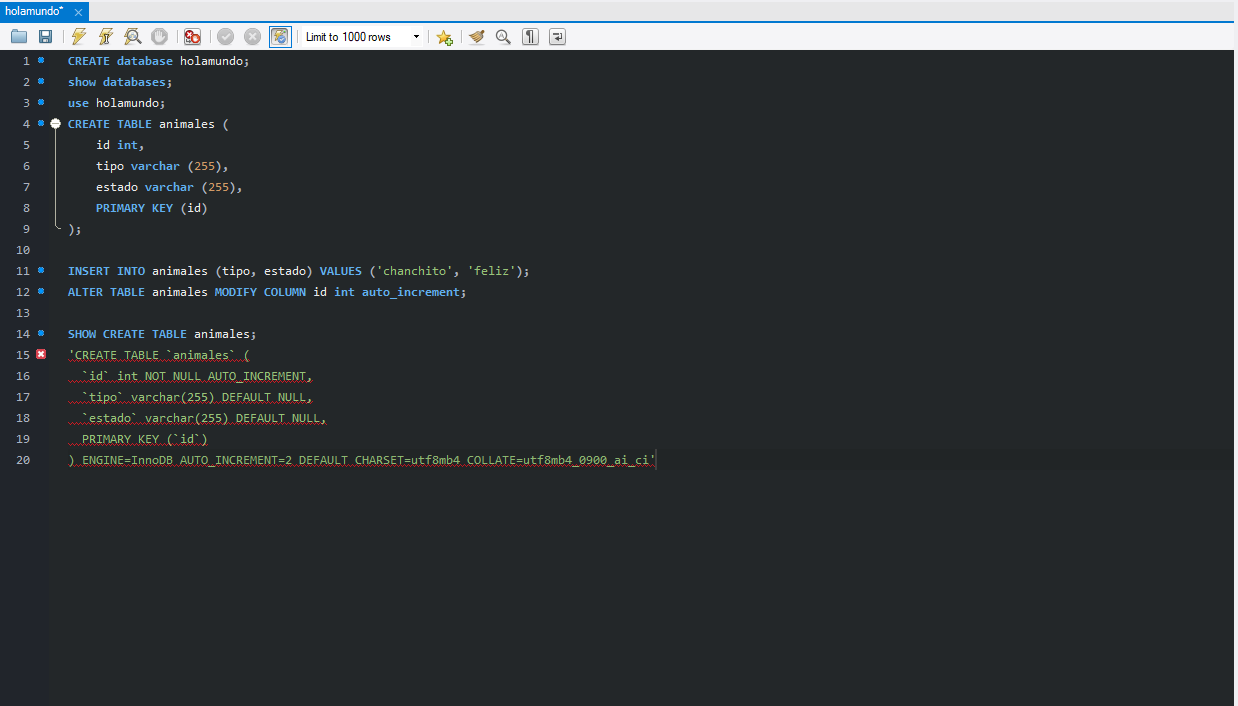
**SHOW CREATE TABLE animales;**



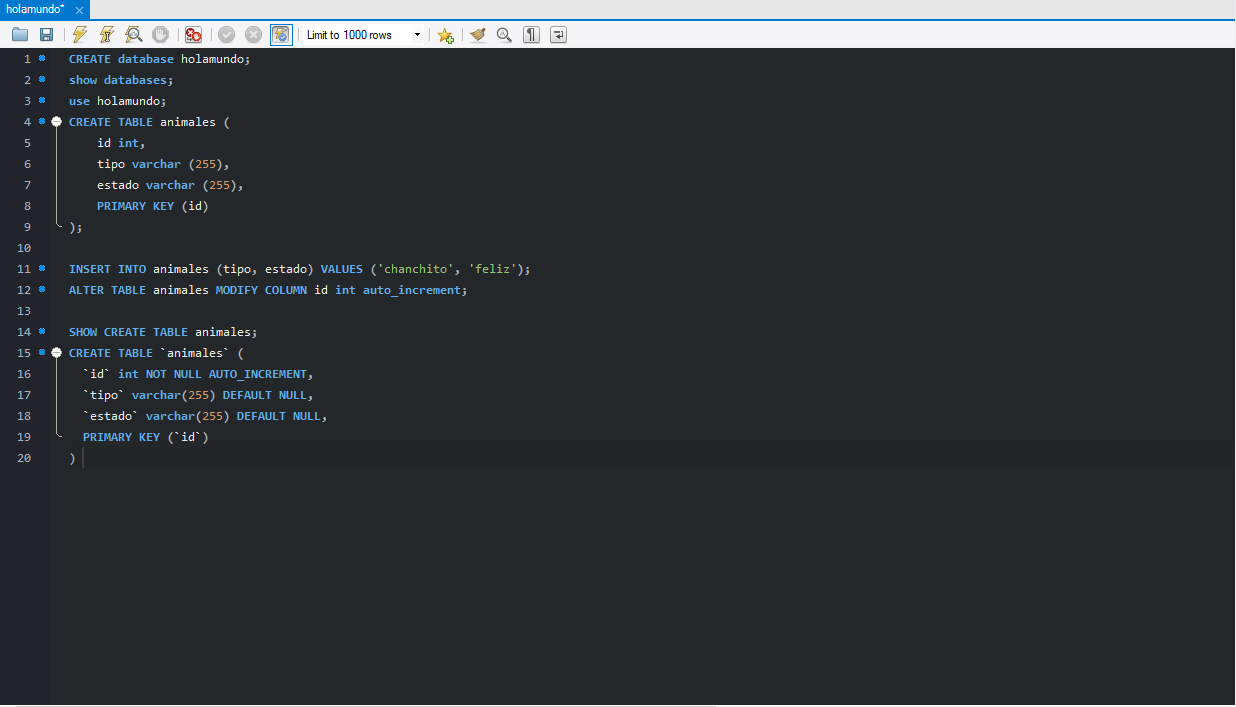
Y con esto, al seleccionar y ejecutar, obtenemos el comando que se utiliza para crear esta misma tabla, sin tener que redactar el comando por nuestra cuenta. **Obviamente, la tabla debe hacer sido creada antes:**

****

Lo resaltado en azul seria el comando que nos genera la sentencia anterior. Hacemos clic derecho en esto y apretamos “Copy Field”. Lo seleccionamos y lo pegamos en el editor luego del SHOW CREATE TABLE animales;



Nos mostrara en error ya que esta copiado como tipo carácter, por lo que debemos borrar la primera comilla antes de CREATE TABLE. **Borramos la primera comilla y todo lo que se encuentra luego del ultimo paréntesis (esto en este momento no es obligatorio incluirlo, y para no complicarnos la vida, lo dejamos hasta aca).**

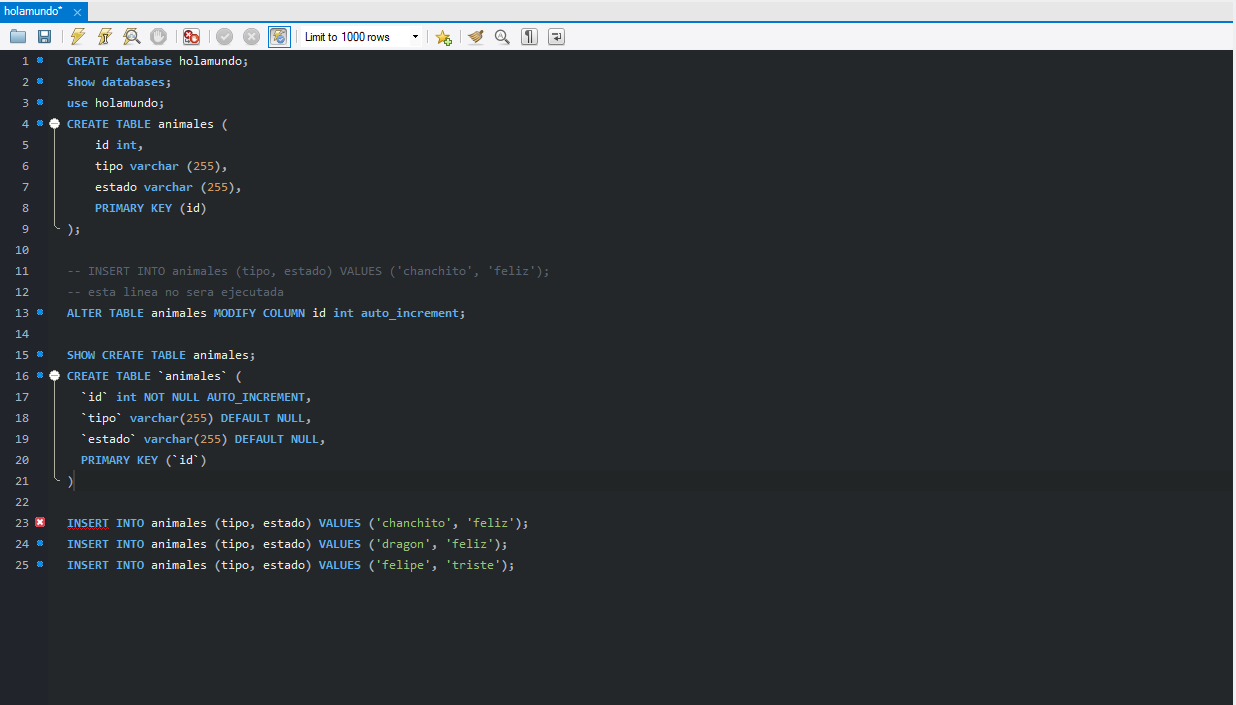
****

Aca vemos que creo la tabla animales puesta entre comillas simples indicando strings, y después tenemos nuestro id, seguido del tipo de dato y agregó **not null.** Cuando nosotros le agregamos la propiedad de autoincrement también lo que hace es indicar que esta columna **no puede ser nula.** Esto puede ser redundante porque cuando comencemos a ingresar datos estos van a cumplir con ser autoincreméntales, por ende, siempre se va a estar agregando un valor. Finalmente tenemos el tipo, estado y ambos cuentan con ser varchar de 255, y su valor por defecto es null. No es necesario que agreguemos **default null** cuando creemos una tabla, pero esto hace que sea un poco más explícito para que cuando después nosotros veamos el comando de CREATE TABLE SEPAMOS QUE ESTA VA A TENER UN VALOR POR DEFECTO DE NULO. Pero en MySQL no es necesario que agreguemos esto.

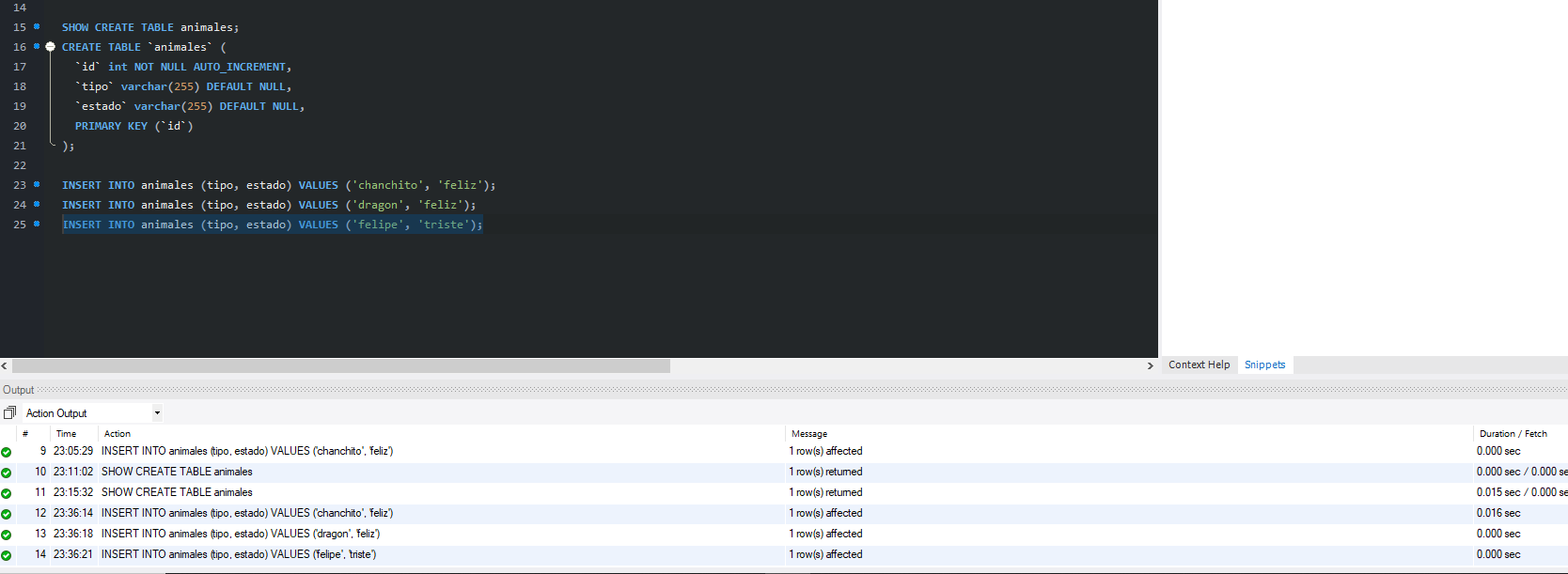
Comentarios en MYSQL:



**Con doble guion (--) se comenta una línea en Workbench.**

****

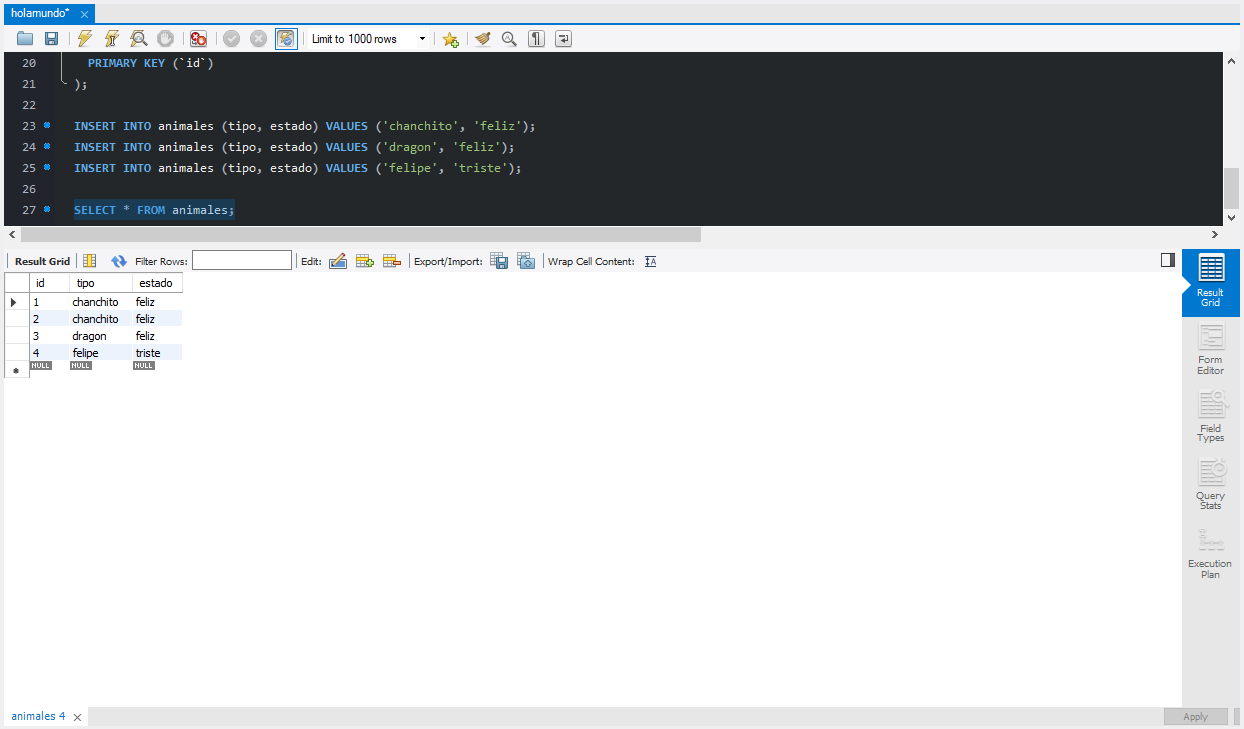
Agregamos otros valores como se muestra en la foto luego del paréntesis. Mostrará error ya que falta un ; en el paréntesis de arriba. Esto es un **error común**. Lo agregamos y ejecutamos las sentencias INSERT INTO una por una:



Ahora, ¿Cómo podemos listar nuestros registros que hemos ingresado?

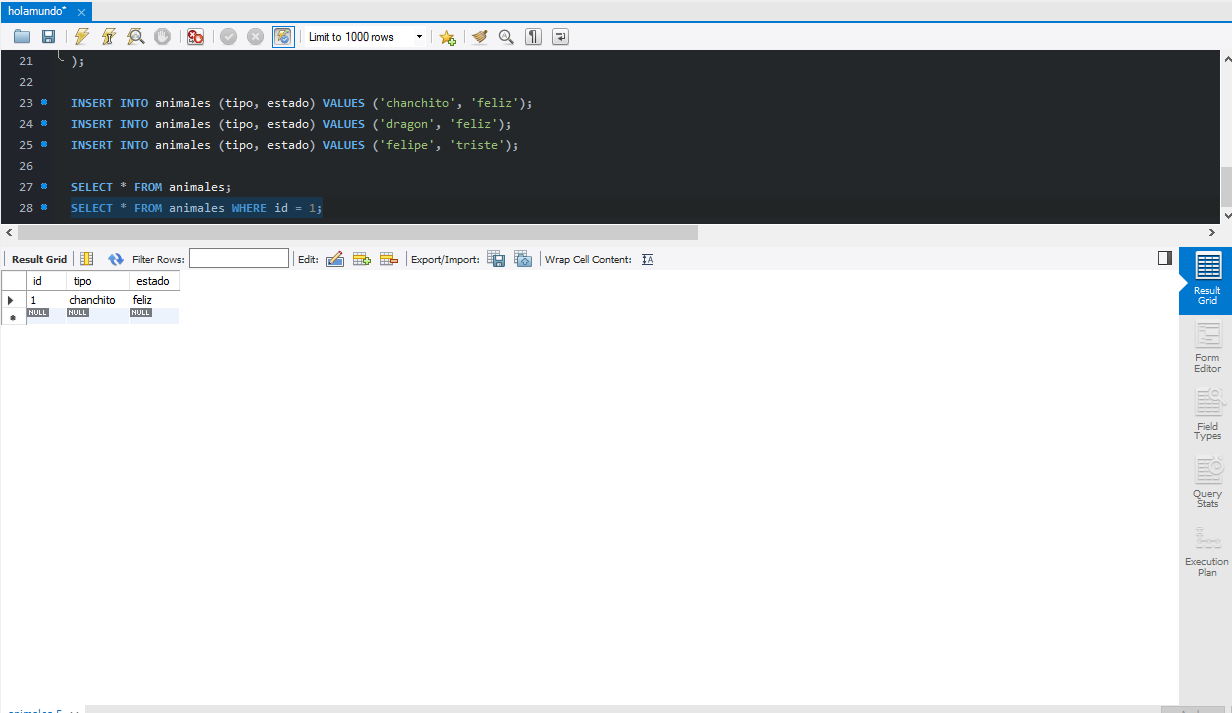
**SELECT** (seleccionar elementos de una tabla) \* (asterisco) **FROM animales** (indicamos el nombre de la tabla).

Seleccionamos la línea, ejecutamos y nos mostrara la tabla de la siguiente manera:

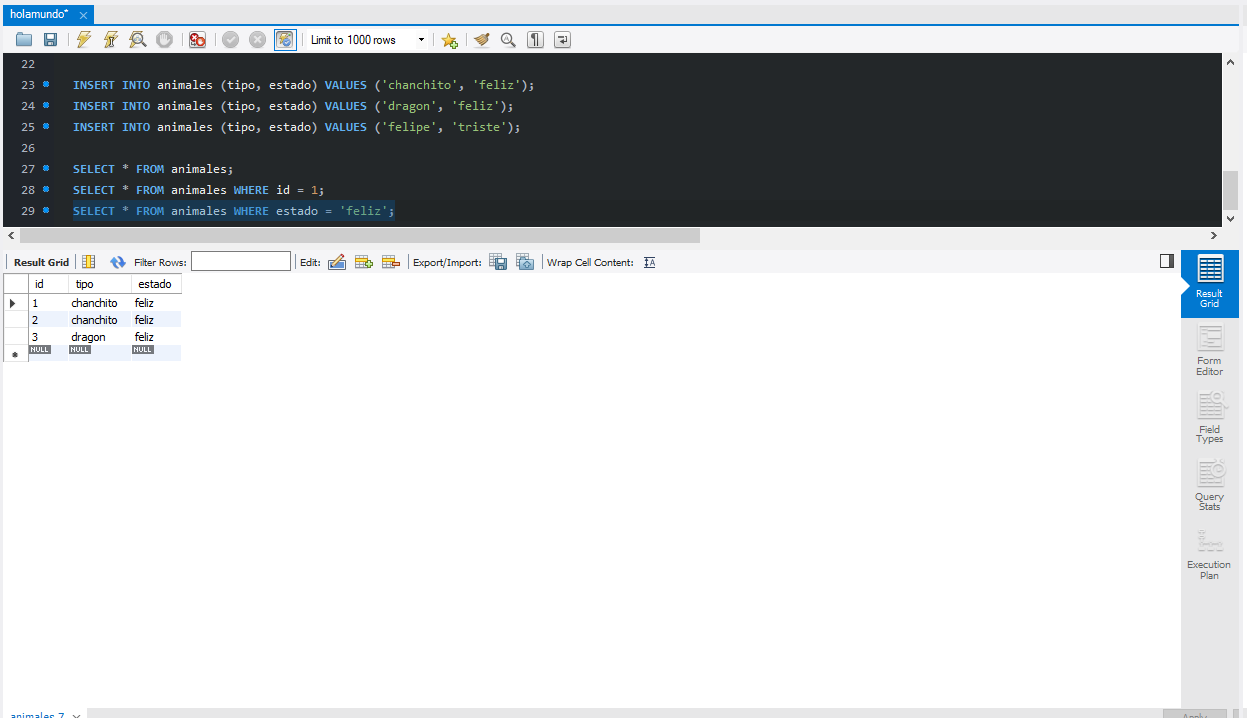


¿Y si solo quiero seleccionar un solo registro?

**SELECT** (seleccionar elementos de una tabla) \* (asterisco) **FROM animales** (indicamos el nombre de la tabla) **WHERE** (donde) **id = 1;**

****

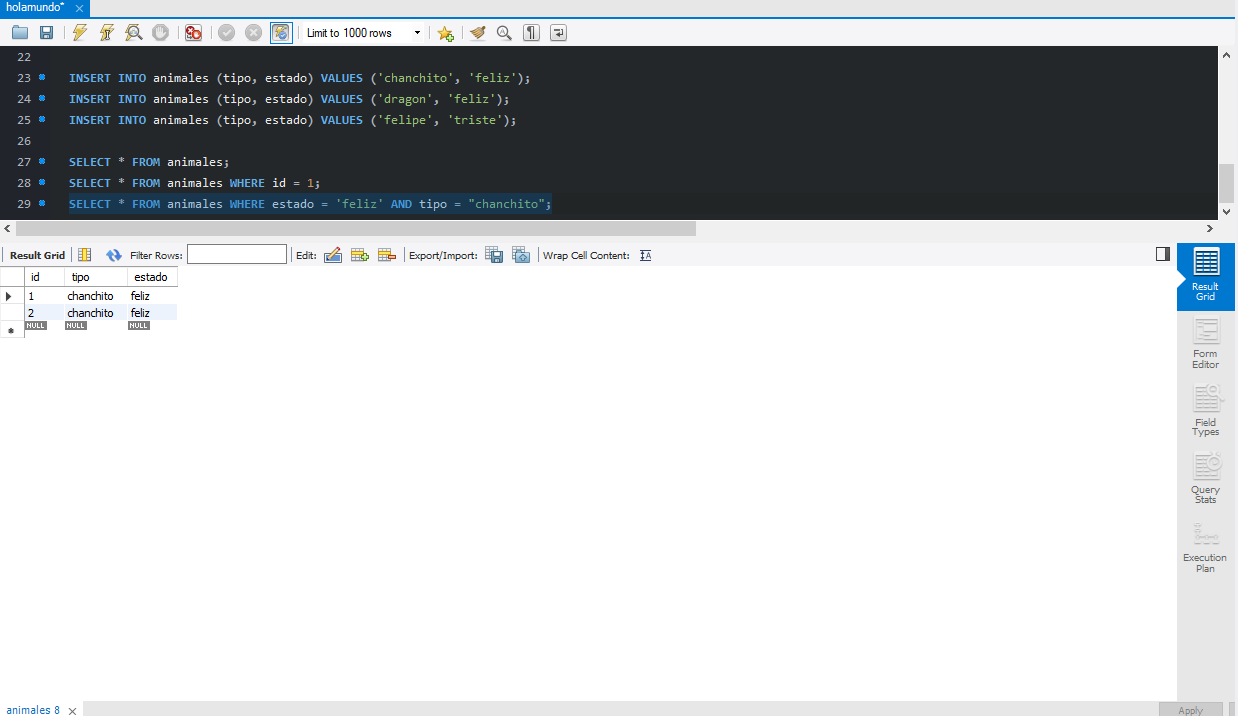
**SELECT** (seleccionar elementos de una tabla) \* (asterisco) **FROM animales** (indicamos el nombre de la tabla) **WHERE** (donde) **estado = ‘feliz’;**

****

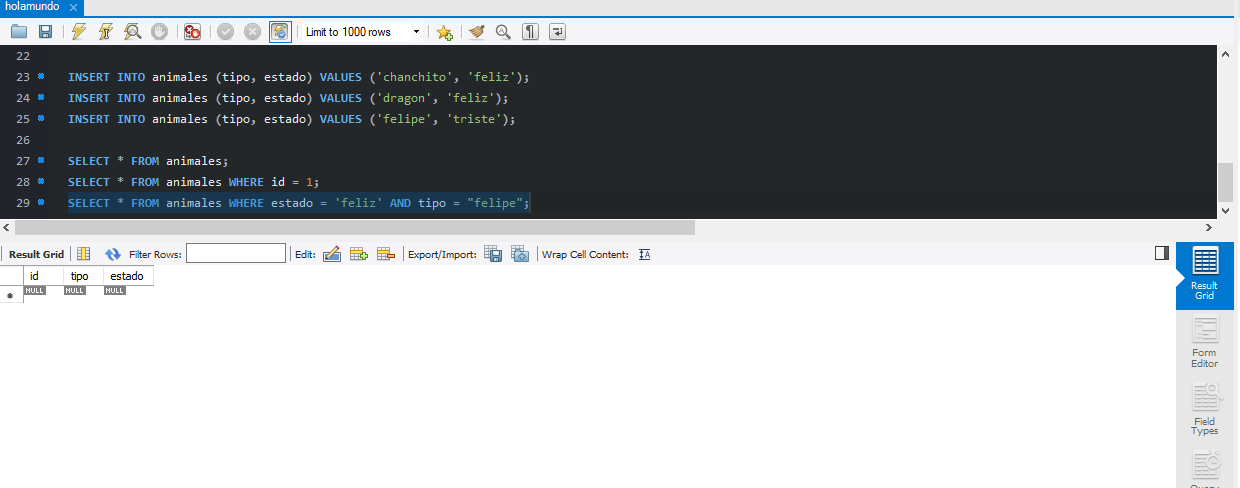
**En este caso, podemos consultar por un valor que exista en alguna de las columnas para que esta nos devuelva registros.**

Podemos usar el operador **AND** para que nos devuelvo un registro o registros con ambos valores consultados:

**SELECT** (seleccionar elementos de una tabla) \* (asterisco) **FROM animales** (indicamos el nombre de la tabla) **WHERE** (donde) **estado = ‘feliz’ AND tipo = ‘chanchito’;**

****

Si cambiamos chanchito por Felipe, no nos mostrara nada, debido a que ninguna condición se cumple, por ejemplo:

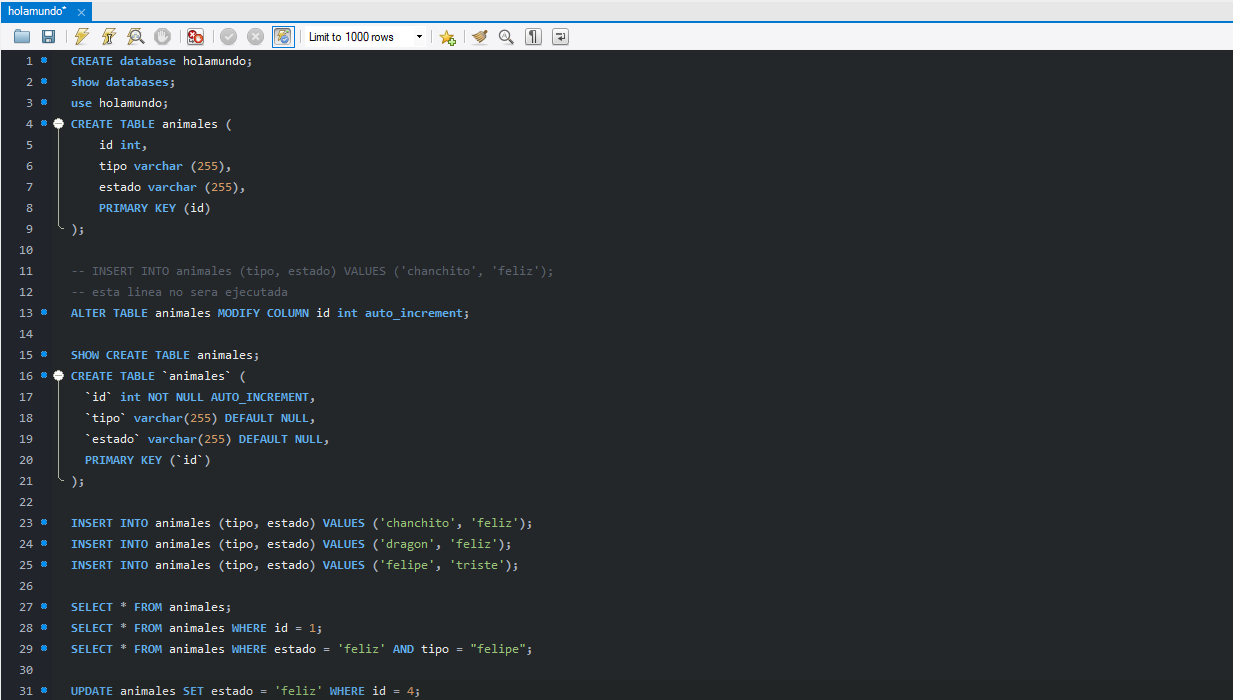


Ya que no existe felipe feliz, sino felipe triste.

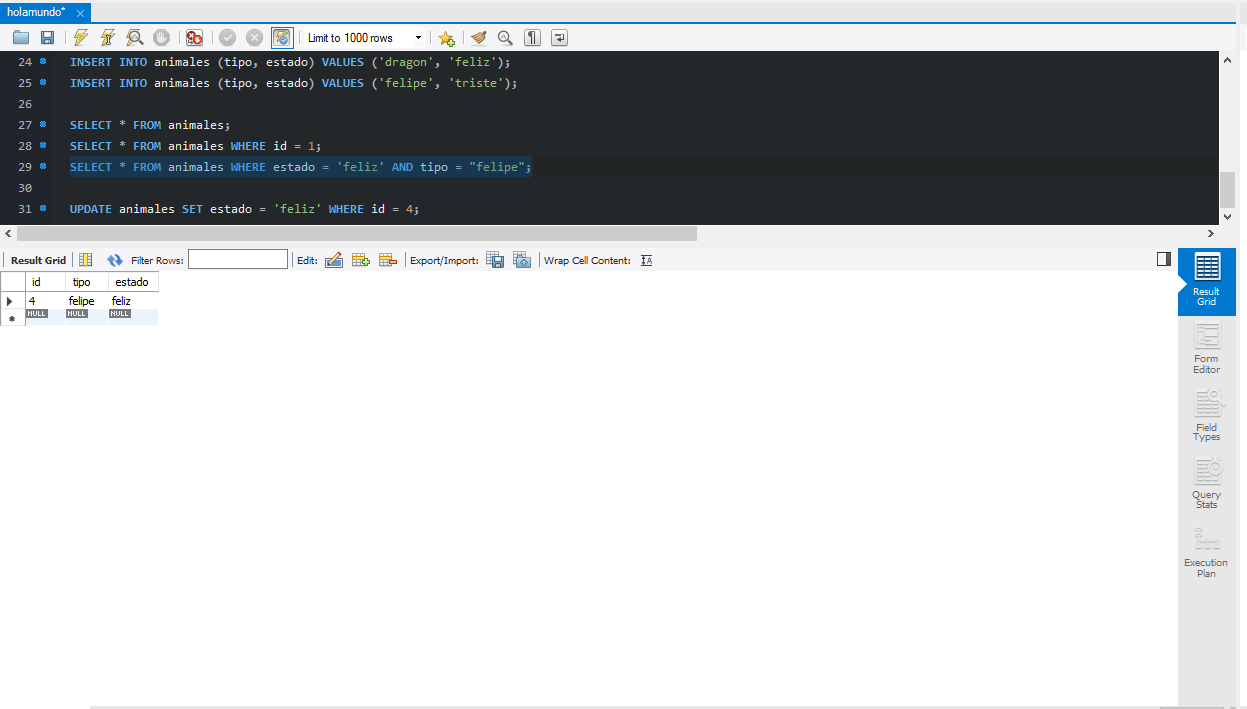
Actualizar datos:

**UPDATE animales SET estado = ‘feliz’ where id = 4;**

Aca lo que estamos haciendo es indicar mediante **UPDATE** que queremos actualizar la tabla ‘animales’**,** la columna **estado** y queremos que su valor pase a ser **‘feliz’** solo en el registro que contenga el id 4 (en este caso felipe)



Y como vimos antes, no se mostraba ningún registro que tuviera ‘felipe’ y ‘feliz’ pero si ahora ejecutamos esa misma línea:

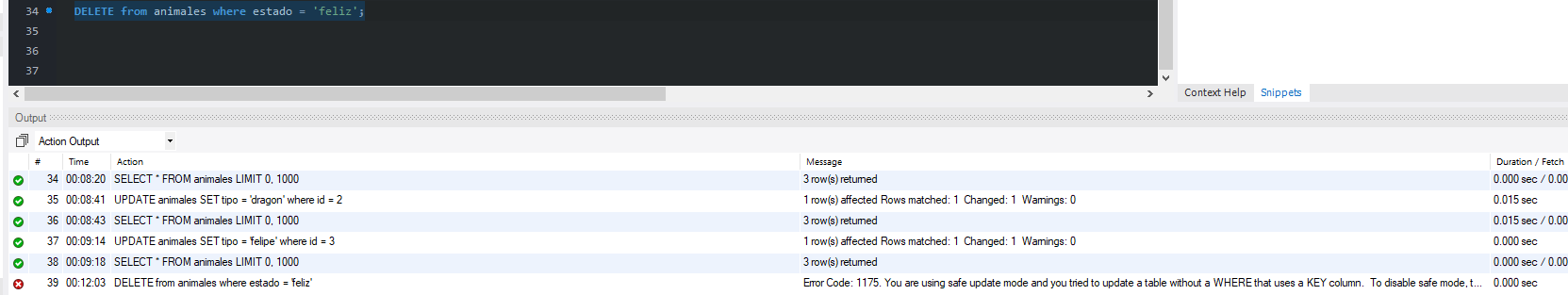


IMPORTANTE: si están atentos, debieron notar que hay una redundancia de datos con chanchito y feliz, los reto a actualizar la tabla mediante el comando UPDATE y eliminar esa redundancia de datos.

Eliminar registros de nuestra base de datos:

**DELETE from** animales **where** estado = ‘feliz’;

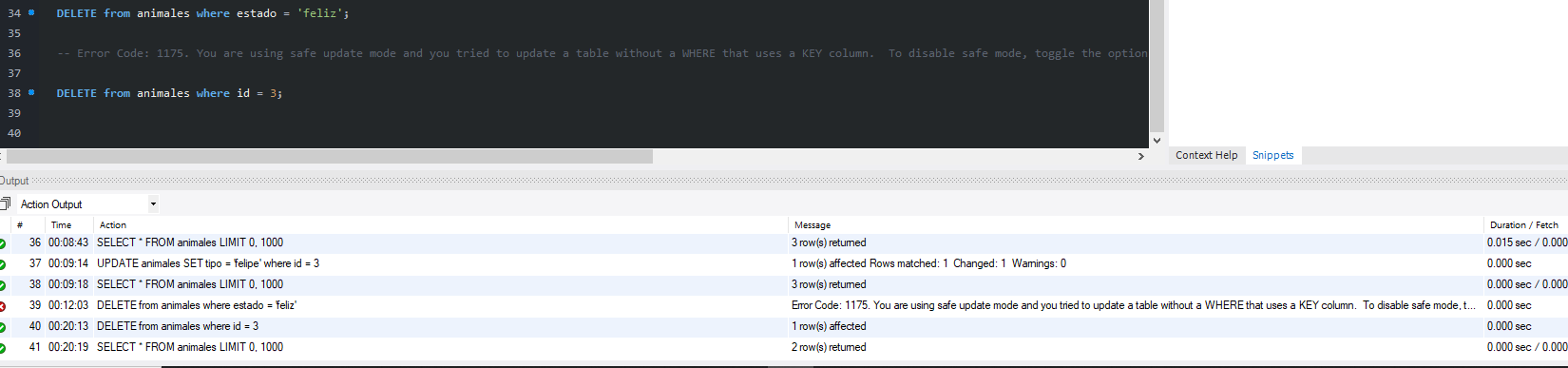
Aca indicamos que queremos eliminar de la tabla animales los registros que tengan de estado ‘feliz’, **sin embargo, esto mostrara un error al querer ejecutarse:**

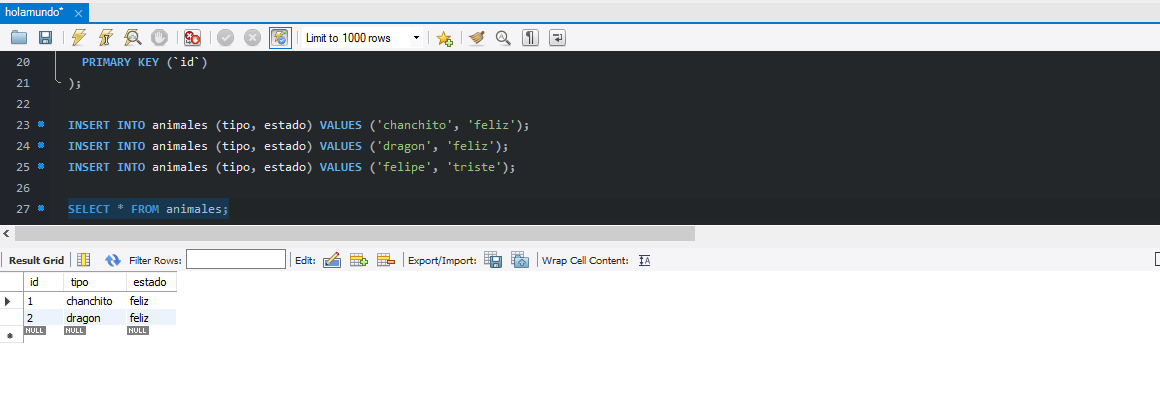
****

**Error Code: 1175. You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column. To disable safe mode, toggle the option in Preferences -> SQL Editor and reconnect.**

SE HIZO UN DELETE SIN AGREGAR UN WHERE A LA CONSULTA. Afortunadamente, MYSQL viene con un modo seguro que no nos permite realizar este tipo de consultas que probablemente terminen en desastre. Por ende, nos va a forzar a que cada vez que queramos realizar un DELETE y UPDATE a que tengamos que indicarle necesariamente un ID.

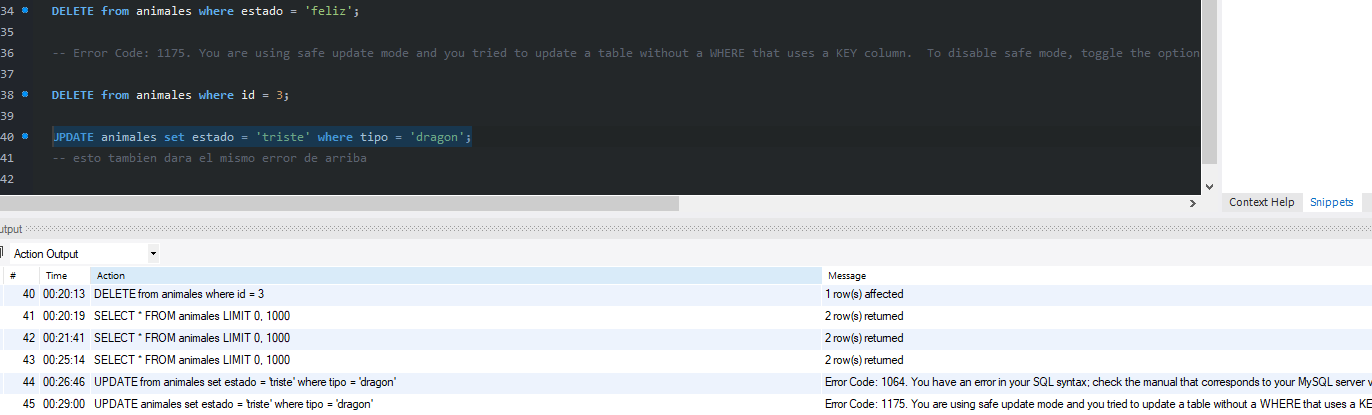
Entonces, indicamos, el id en vez de la columna que queremos eliminar;

Hacemos el SELECT \* FROM animales;



Y veremos cómo se eliminó el registro con el ID = 3

Como dije antes, **un UPDATE** también puede largar este mismo error visto más arriba:

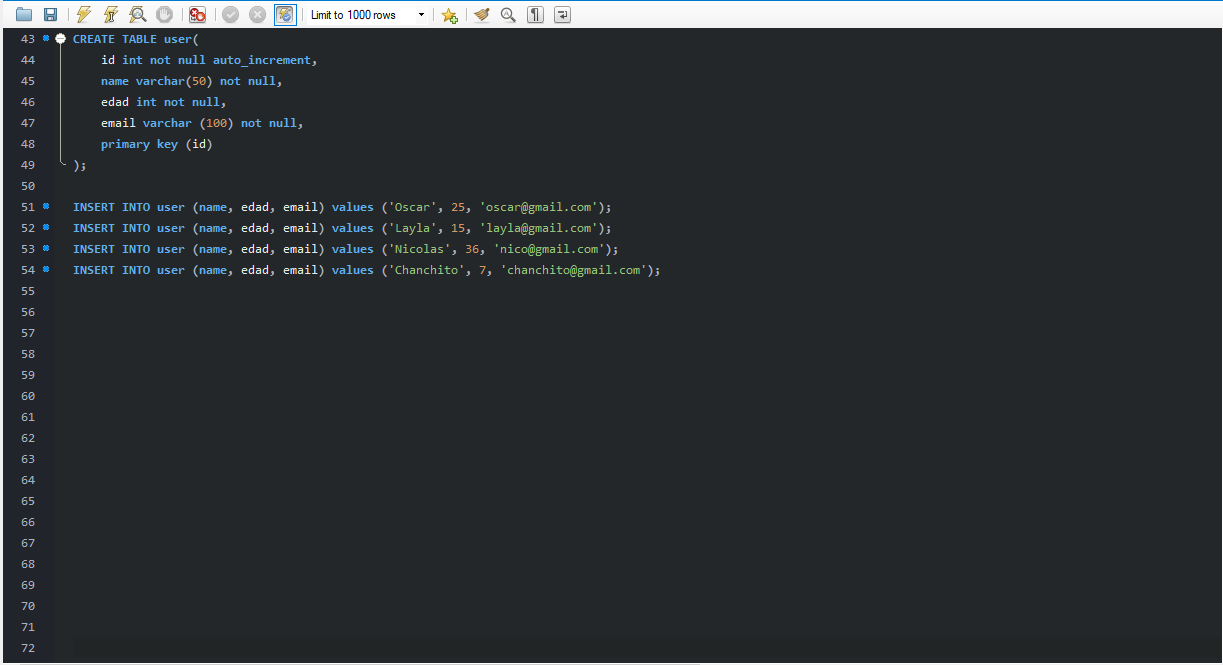
****

No estamos indicando el id, por lo que dará el mismo error que en el DELETE más arriba. **SI O SI DEBEMOS INDICAR EL id. ARRIBA TUVE UN ERROR MIO DE SINTAXIS, SORRY D:**

**YA APRENDIMOS A REALIZAR LAS 4 ACCIONES PRINCIPALES DE SQL: Create, Read, Update, Delete (CRUD)**

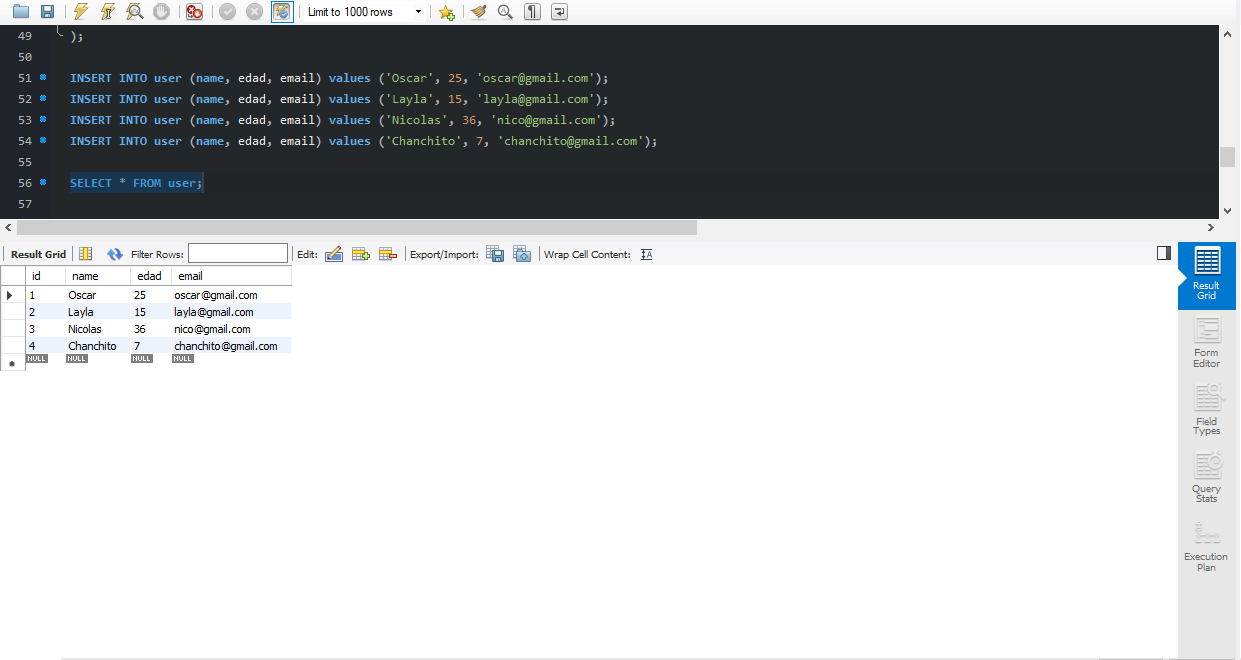
**AHORA, PROFUNDIZAREMOS EN EL “READ”: EMPEZAREMOS A VER OTRAS FORMAS DE SELECCIONAR (SELECT) LOS ELEMENTOS QUE SE VAN A ENCONTRAR DENTRO DE UNA TABLA.**

**CREAREMOS UNA NUEVA TABLA, LLAMADA user** y le asignamos todos los valores de la foto:



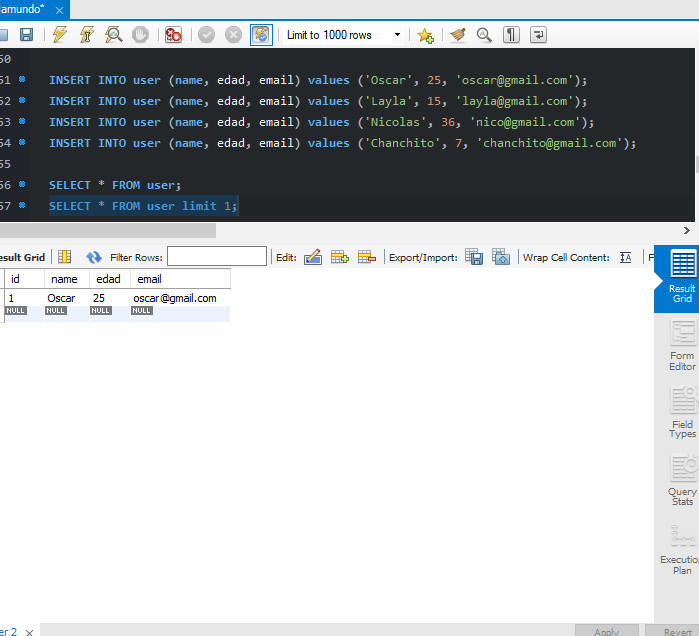
Ahora profundizaremos más en **SELECT,** que nos permite obtener datos de nuestra base de datos.

Mostramos nuestros registros usando **SELECT \* FROM user;**



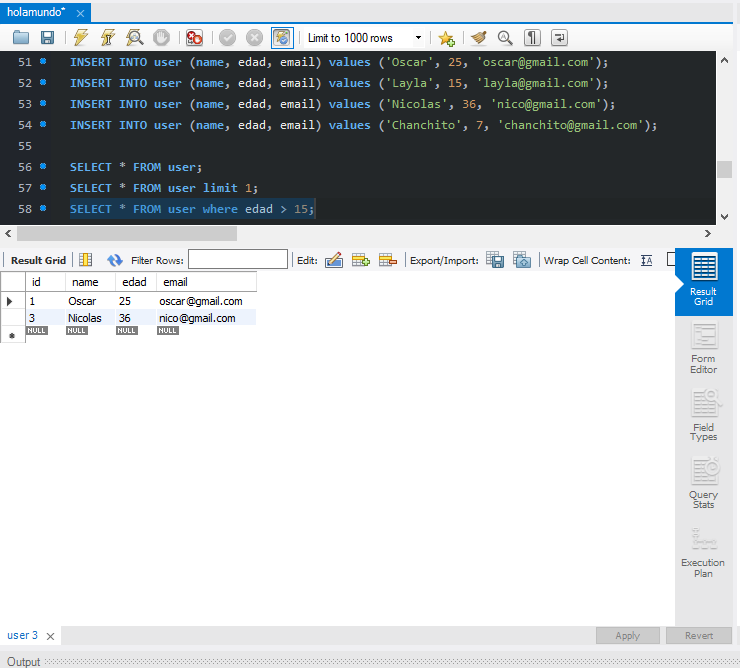
**SELECT \* FROM user limit 1;**

Como lo dice la palabra limit, limitara la cantidad de recursos que se van a retornar hacia nosotros, en este caso limit 1, nos devolverá el primer registro que este encuentre:

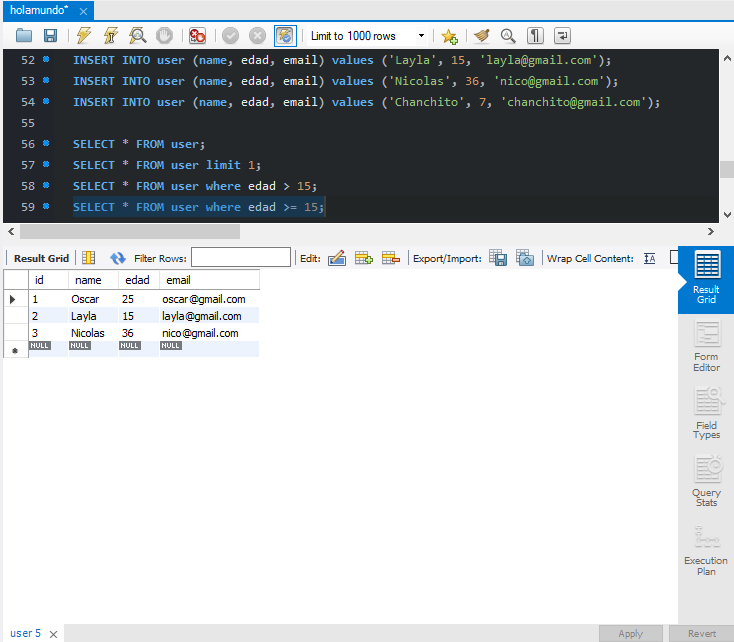


**SELECT \* FROM user where edad > 15;**

Esto nos mostrara los registros en los que se encuentre que la **EDAD SEA MAYOR (>) a 15:**

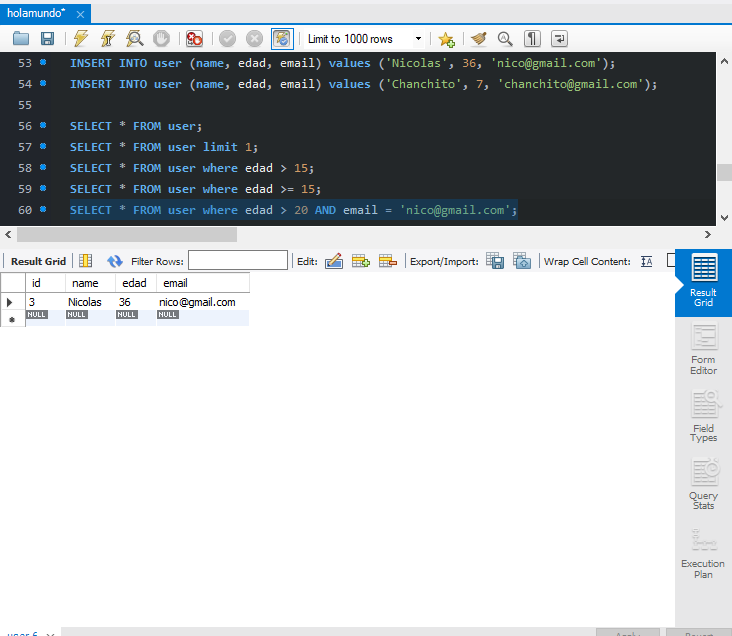
****

**SELECT \* FROM user where edad >= 15;**

****

En este caso, nos devolverá también el registro de Layla ya que su edad es = 15

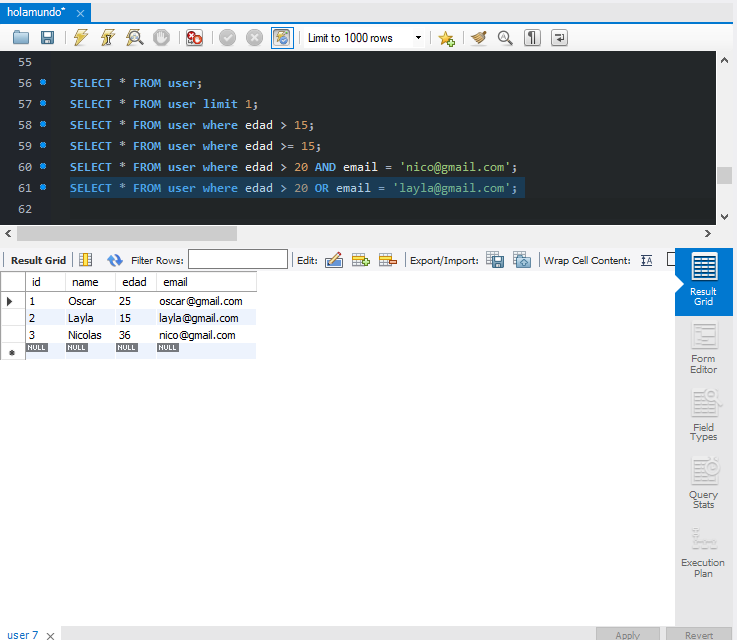
**SELECT \* FROM user where edad > 20 AND email = 'nico@gmail.com';**



Si bien en este caso pedimos datos de edad mayores a 20, estamos especificando el mail que queremos. Por ende, mostrara el registro con dicho email y que sea mayor a 20.

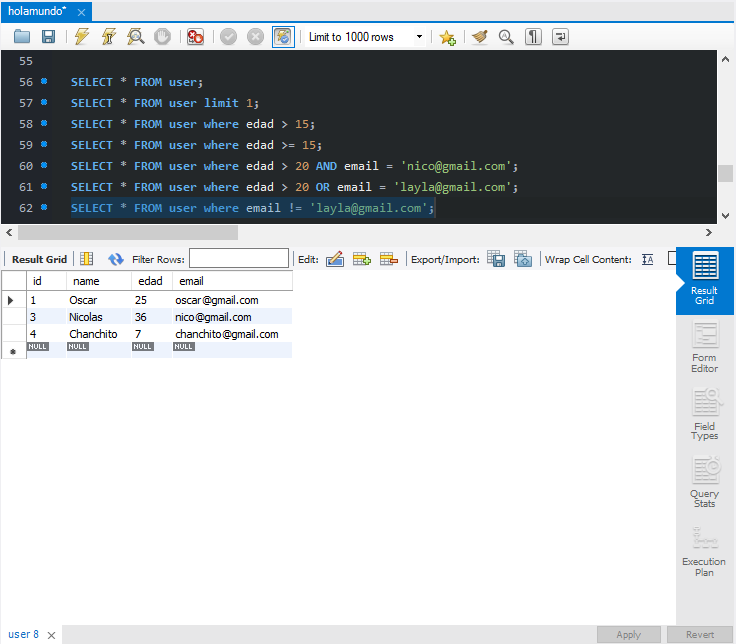
**SELECT \* FROM user where edad > 20 OR email = 'layla@gmail.com';**

Aquí viene la diferencia: primero se traerá todos los registros que cumplan con la condición de tener más de 20 años, pero independientemente que tenga mas de 20 años o no, se traerá también el registro que contenga el email descrito:



Aca vemos que se muestran los registros con mayoría de edad 20 y también el de Layla que no tiene 20 años, pero si tiene el mail escrito en la sentencia SELECT.

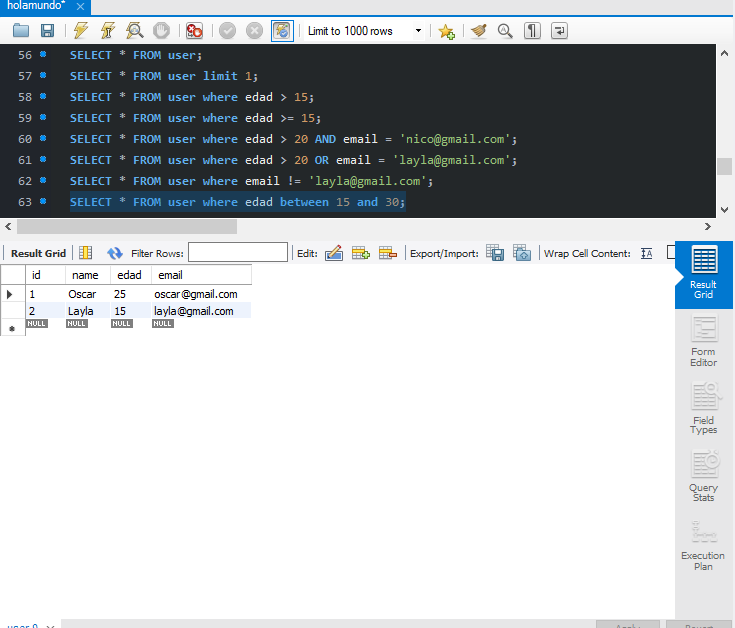
**SELECT \* FROM user where email != 'layla@gmail.com';**

****

Me mostrara registros que no contengan dicho mail.

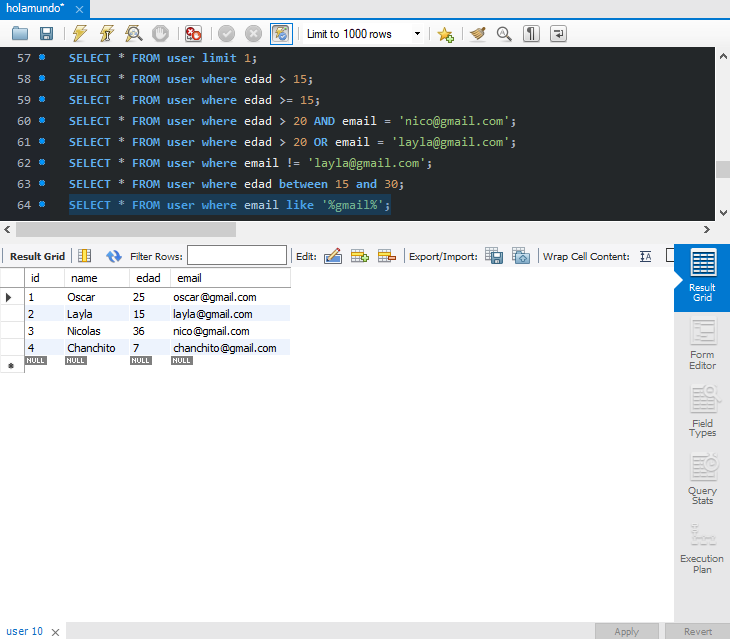
**SELECT \* FROM user where edad between 15 and 30;**

Seleccionar de user donde la edad este DENTRO (between) 15 y 30, mostrando así dichos registros que cumplan con estás condiciones:



Ahora veremos una instrucción con la cual podemos ir a buscar algo en la base de datos, pero cuando el usuario quiere buscar **algo en concreto**:

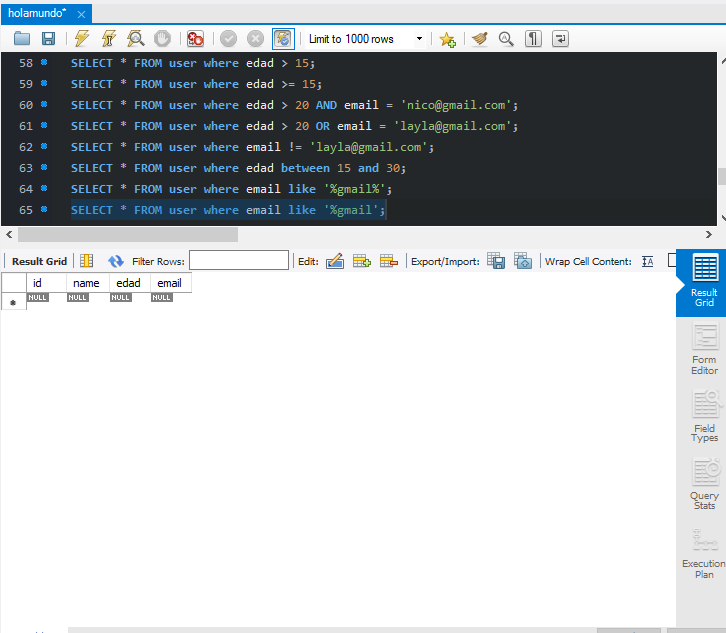
**SELECT \* FROM user where email like '%gmail%';**



De esta forma mostrara todos los registros que contengan la cadena **Gmail** en él, **pero me da lo mismo su inicio y su final**.

**SELECT \* FROM user where email like '%gmail';**

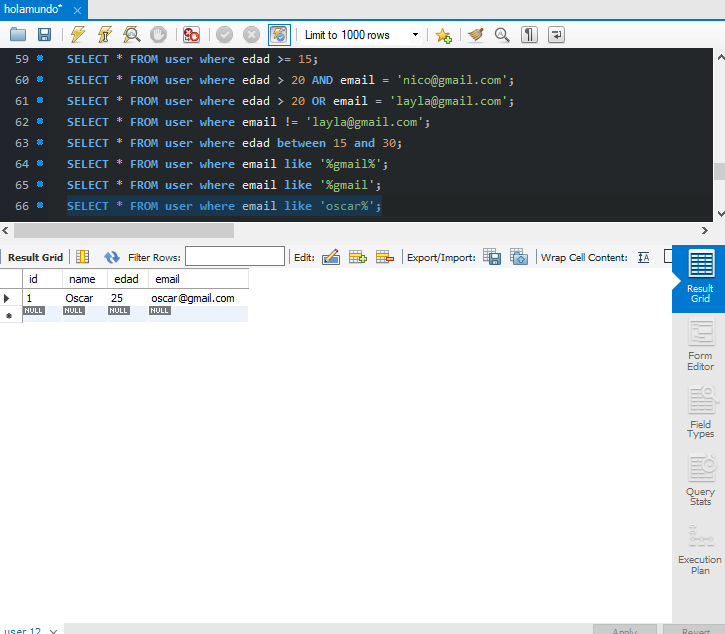
Como podrás ver, hicimos un pequeño cambio eliminando el porcentaje final dentro de las comillas simples. **Aca lo que estamos indicando es que la palabra gmail puede empezar con cualquier cosa, PERO DEBE TERMINAR CON GMAIL:**

****

Claramente, no va a mostrar nada, porque no hay nada que termine solo con Gmail (siempre llevan un .com al final).

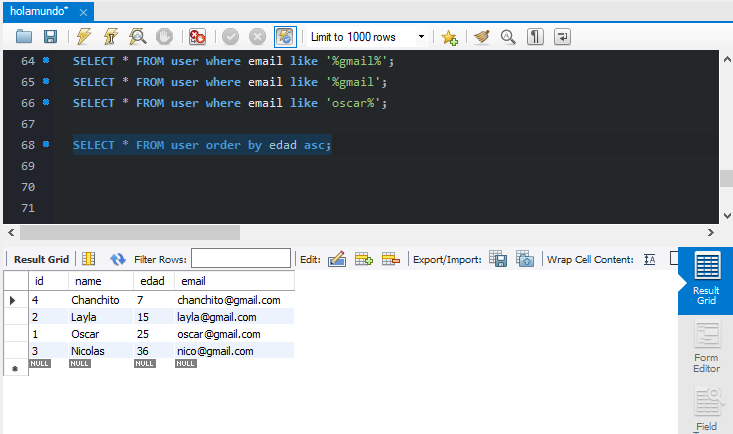
**SELECT \* FROM user where email like 'oscar%’;**

Esto sería el viceversa de lo anterior, buscamos el registro que **SI O SI** tiene que empezar con oscar, **pero no importa su final** por ende nos muestra el registro de oscar completo:



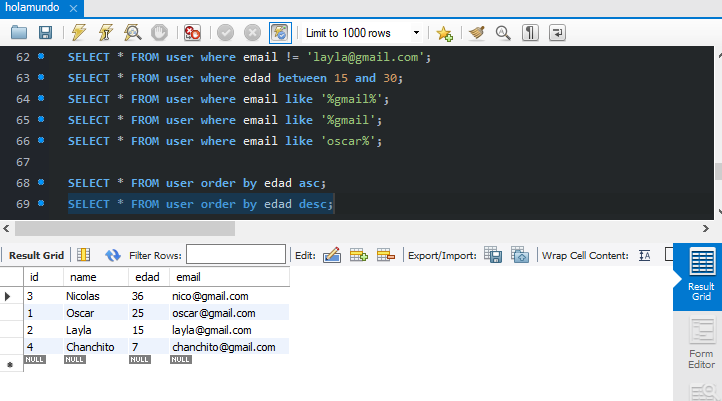
**SELECT \* FROM user order by edad asc;**

Indicamos que nos muestre los registros en orden ascendente mediante la edad:



**SELECT \* FROM user order by edad desc;**

Indicamos que nos muestre los registros en orden descendente mediante la edad:

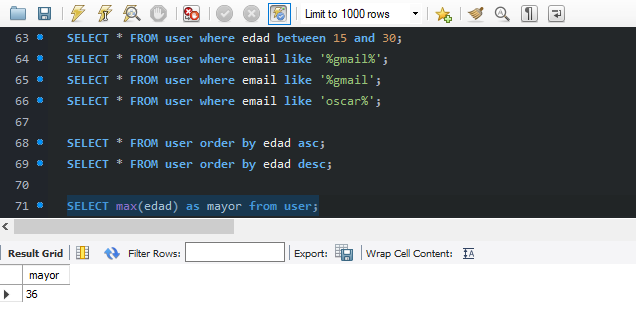


**SELECT max(edad) as mayor from user;**

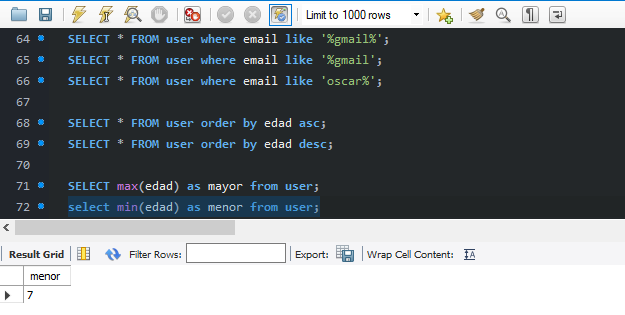
**select min(edad) as menor from user;**

Aqui nosotros estamos utilizando la palabra reservada SELECT seguida de **max** o **min** (máximo/minimo) las cuales son funciones por ende usan paréntesis (). Dentro de estos paréntesis se indica la **columna (edad),** después de eso indicamos como queremos que este se llame utilizando la palabra reservada **as,** seguida del nombre del alias (o sobrenombre: **mayor y menor,** en este caso), luego el FROM indicando la tabla:

**max**

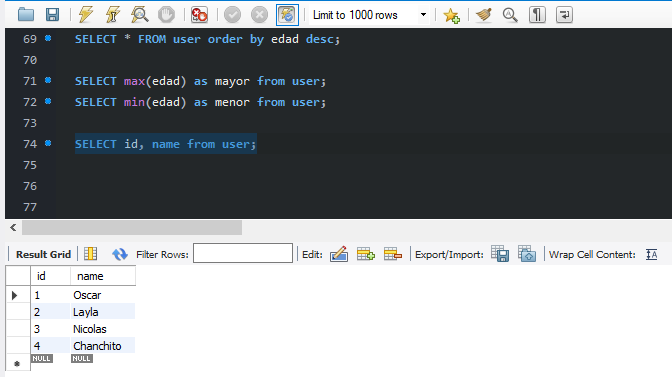
****

**min**

****

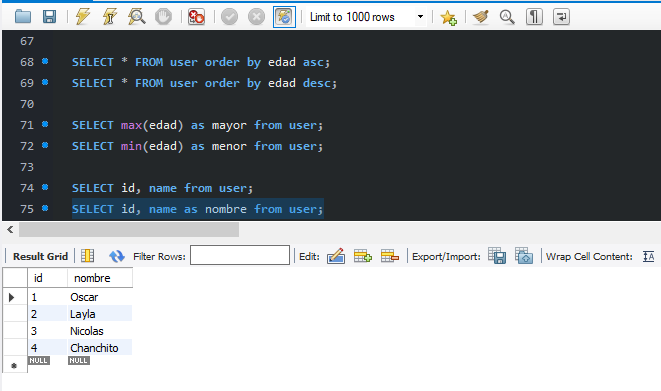
**COMO SELECCIONAR LAS COLUMNAS DE NUESTRO INTERES:**

**SELECT id, name from user;**

****

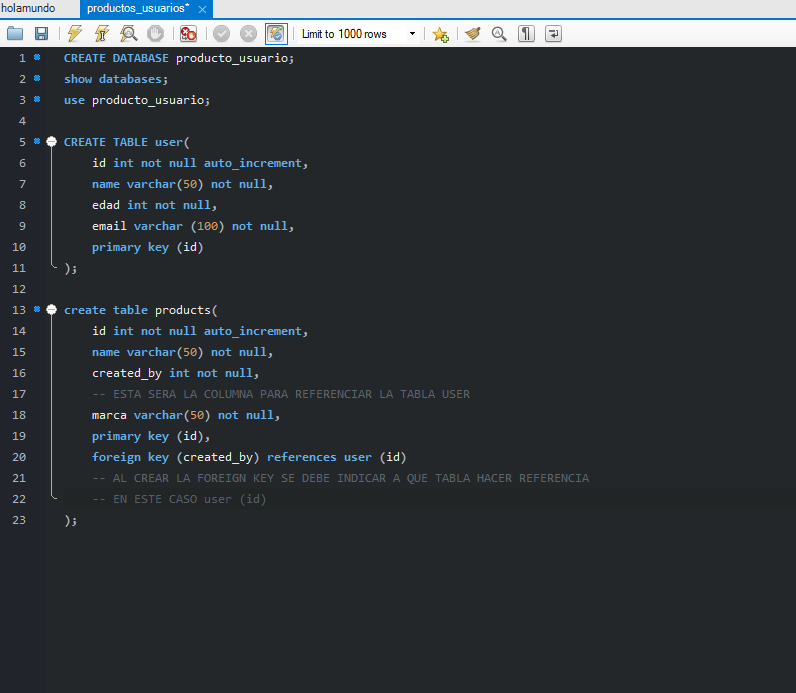
**CAMBIAR EL NOMBRE DE LA COLUMNA QUE SE ESTA MOSTRANDO:**

**SELECT id, name as nombre from user;**

****

**UNIR UNA TABLA CON OTRA TABLA IMPORTANTE**

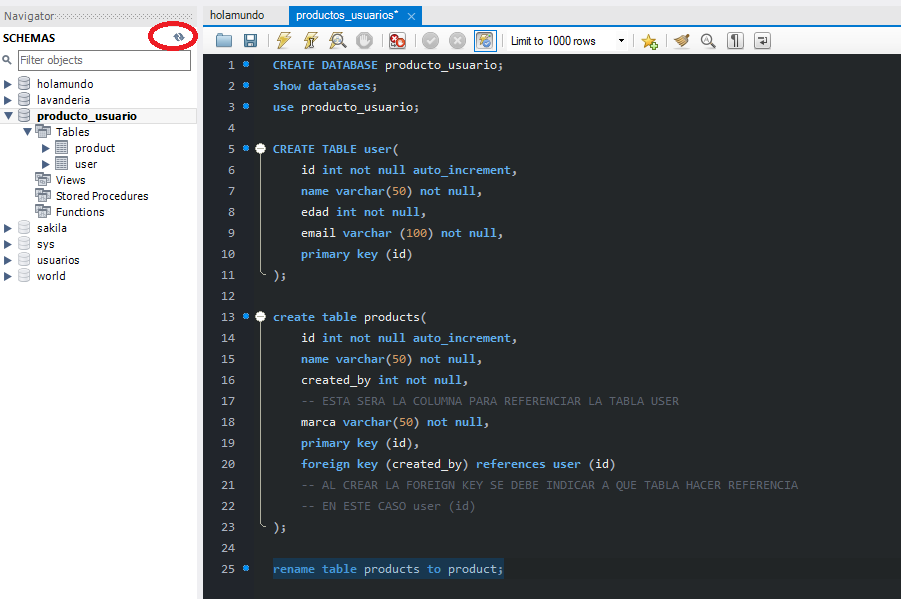
Lo que haremos ahora es crear una nueva tabla que podamos cruzar con la tabla User, y de esa forma empezar a usar **llaves foráneas,** para poder identificar al creador de este producto. Crearemos la tabla products y cuando alguien ingrese un dato vamos a asignarle el id del usuario que creo el producto.

****

**EN MI CASO CREE OTRA BASE DE USUARIOS CON LA MISMA TABLA USER DE LA ANTERIOR MAS LA NUEVA TABLA PRODUCTS AGREGANDO LOS VALORES A USER TAMBIEN.**

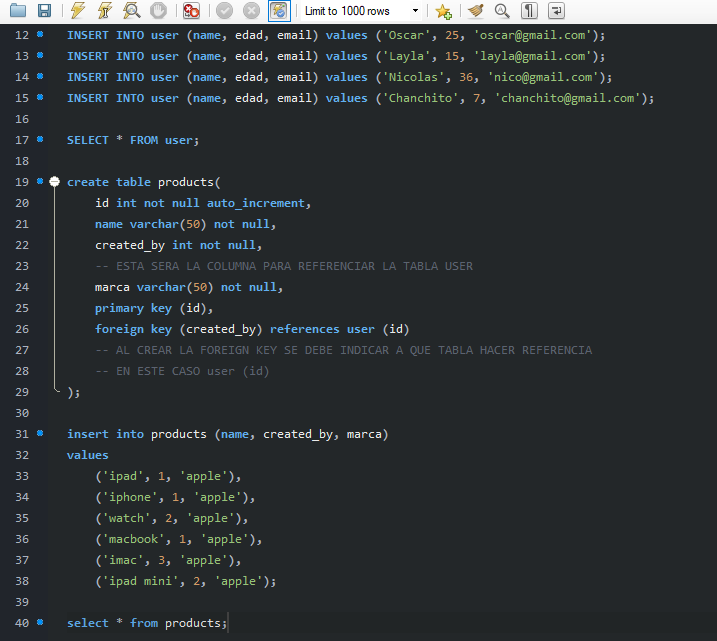
Si queremos cambiar el nombre de la tabla products a singular (product) usamos el siguiente comando

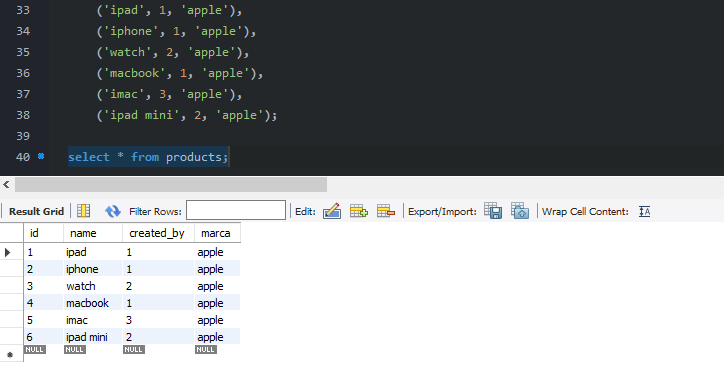
**rename table products to product**

****

**En el caso de que no se actualice el nombre, hacer clic en las flechitas de refresh indicadas en la foto de arriba. Personalmente, yo la deje igual, osea “products”.**

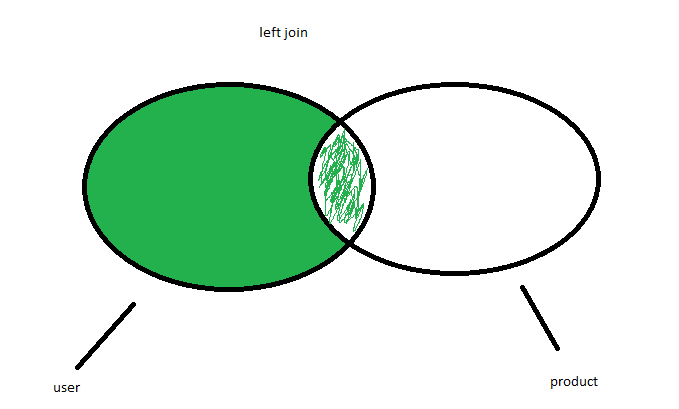
Ahora insertaremos unos valores para poder jugar con los datos, y aprovecho para mostrar una nueva alternativa de poder insertar datos. Antiguamente utilizamos un INSERT por encima del otro, ahora insertaremos todos los datos utilizando solo un INSERT.





**LEFT JOIN**

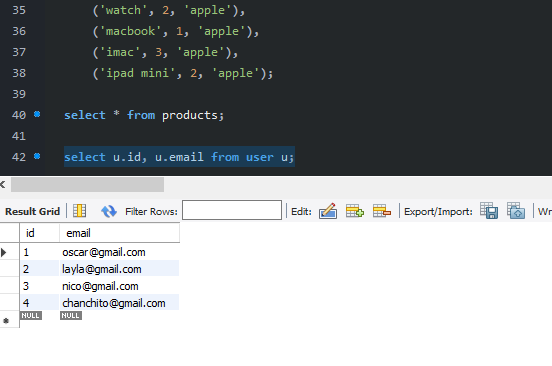
Ahora vamos a hacer un **left join** de la tabla user con la tabla products:



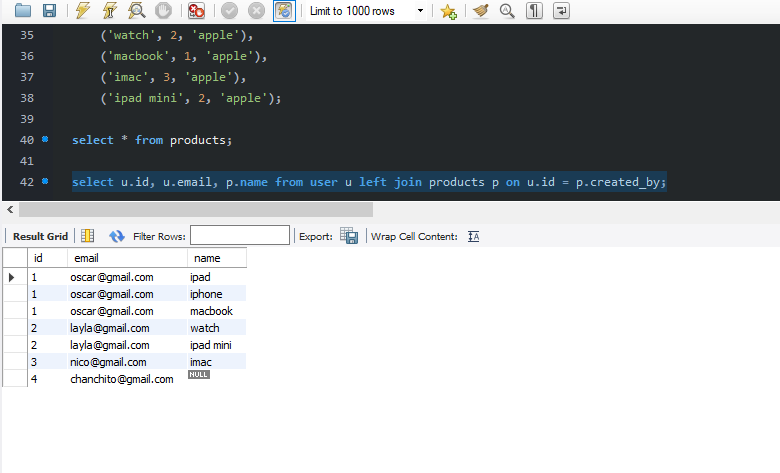
Representadas las tablas con círculos. Lo que vamos a hacer es indicar que tiene que traerme todos los registros que encuentre en la tabla user dependiendo de la consulta que hayamos creado. Y en el caso que se encuentren registros en la tabla product que hayan sido creados (created\_by) por el usuario que aparecen en la consulta de la izquierda, en ese caso, estos productos (los del medio) aparecerán en la consulta. Pero si es que se crearon productos, los cuales no aparecen como resultado sus creadores en la izquierda, en ese caso, esos productos NO VAN A APARECER. Esa es la razón por la que podemos representar un left join de esta manera. En este caso el que manda es la tabla user. Por la consulta que hagamos, vamos a traer todos los usuarios que esa consulta nos arroje y después, si es que estos tienen datos en la tabla product, entonces traelos, sino deja los datos sencillamente como null.

Nuestra consulta se va a escribir de la siguiente manera:

**SELECT u.id, u.email from user u;** / le entregamos un alias a la tabla de usuarios, en este caso **u**

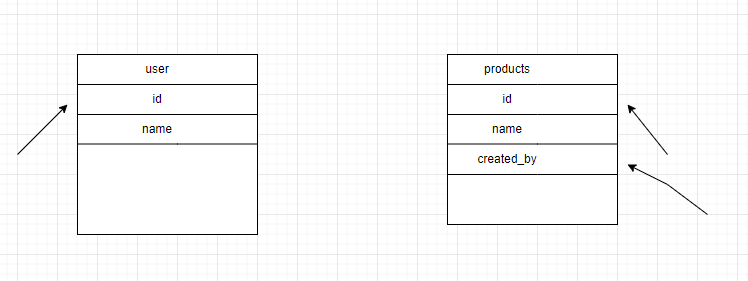


Tengo que indicar de que tabla es la columna que estoy consultando: en este caso se pone u.id ya que u es el alias de la tabla user, y al lado el nombre de la columna.



En esta parte hicimos el **left join.** Aquí indicamos que tabla queremos unir con la tabla de user (en este caso products). Y es la oportunidad para indicarle un alias (p).

Pero yo estoy haciendo un left join, entonces también me voy a traer datos de la tabla de products, voy a querer ver también cual es el producto que me estoy trayendo. Así que agregamos esa consulta en nuestra parte de select (p.name), porque me interesa ver también el nombre del producto. Ahora debemos indicar cuales son las dos columnas que debemos ligar.



En este caso tenemos la tabla user, la cual tiene la columna **id**, es la que nos interesa

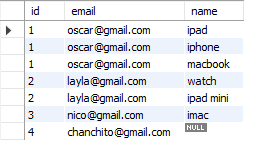
Tambien tenemos la tabla products, que contiene una columna id. La de name nos la vamos a saltar **porque nos interesa la de created\_by.**

Recordemos que nuestra columna created\_by contiene el id del usuario, por eso cuando hagamos el left join lo que vamos a hacer es indicarle a nuestra consulta que donde se encuentra la columna de created\_by vamos a hacer que se una con el id respectivo de la tabla user. **Osea que si en created\_by tenemos un id 1, en ese caso quiero que ese id 1 vaya a buscar a la tabla user al usuario de id 1**.

Escrito quedaria asi:

**select u.id, u.email, p.name from user u left join products p on u.id = p.created\_by**;

Aqui lo que decimos es: juntar, según el id de la tabla user, que este sea igual a la columna created\_by de la tabla product.



En este caso tenemos el id 1, oscar, creo 3 productos (ipad, iphone, macbook)

Layla creo el producto Apple watch y el Apple mini

Nico creo el i mac

Y chanchito no creo absolutamente nada.

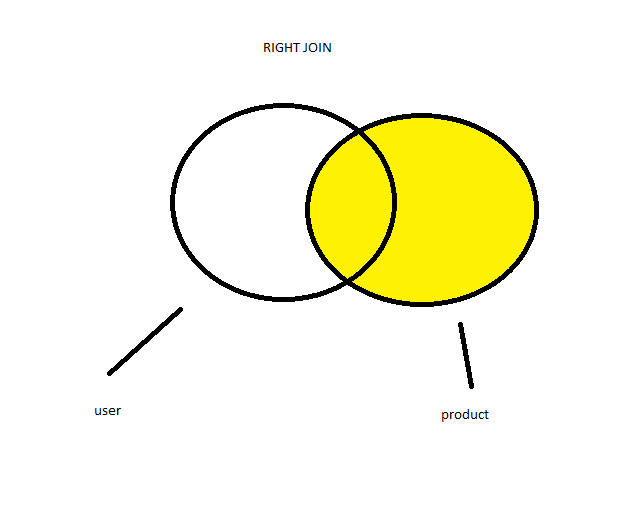
TENEMOS UNA TABLA USER Y PRODUCT; DONDE EN EL LEFT JOIN TIENES QUE TRAERME TODOS LOS REGISTROS QUE ENCUENTRES DENTRO DE LA TABLA USER QUE CORRESPONDA CON LA CONSULTA QUE YO ESTOY EJECUTANDO, Y, TRAEME LOS PRODUCTOS SOLO SI ESTAN ASOCIADOS A UN REGISTRO DE LA TABLA DE USER, ESTO ES UN LEFT JOIN.

**MAS INFORMACION DE LEFT JOIN EN EL SIGUIENTE LINK DESDE EL MINUTO 1.14.35**

**https://www.youtube.com/watch?v=uUdKAYl-F7g&t=2261s&ab\_channel=HolaMundo**

**RIGHT JOIN**

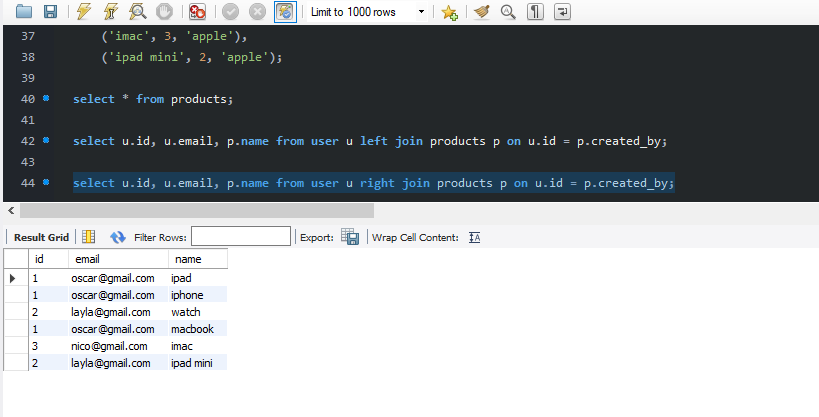
El right join es lo mismo que el left join pero por la derecha.



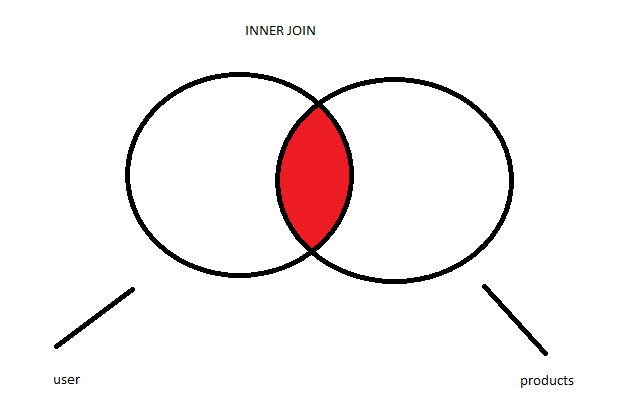
Aquí tenemos ambas tablas, user y products. Vendría siendo lo mismo que left join, pero el que va a traer absolutamente todos los datos, dependiendo de la consulta, va a ser la tabla de la derecha (products). Y, en el caso que el encuentre datos dentro de la tabla de usuarios, este los va a traer, pero, si no existe un dato que se logre relacionar con algún elemento de la tabla de producto entonces no lo va a traer de la tabla de usuario. Es como lo mismo que el left join, pero en lugar de utilizar la tabla de usuario como la tabla principal utiliza la de products.

La consulta sería entonces:

**select u.id, u.email, p.name from user u right join products p on u.id = p.created\_by**; // notese que se cambio left join por right join



**INNER JOIN**

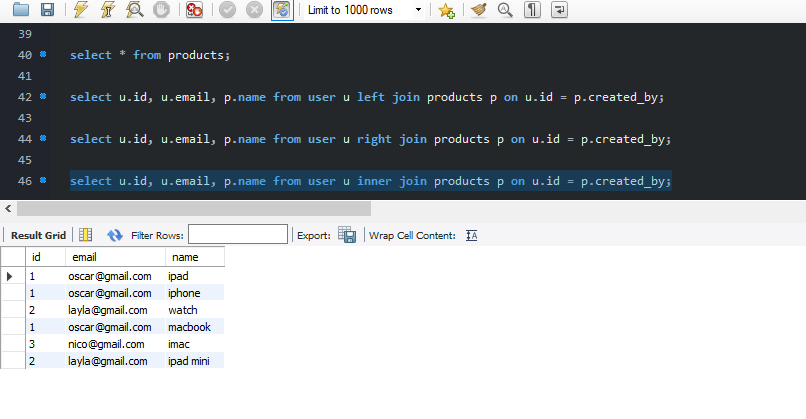
****

En el caso de **left** **join**, nosotros habíamos visto que lo que nos va a traer finalmente son los registros de usuarios que encuentre y si es que encuentra asociado a un producto nos va a traer ese producto. En el caso del **right** **join** lo que va a hacer es que nos va a traer el producto y en el caso que exista nos va a traer un usuario asociado.

En el caso de **inner join** nos va a traer tantos usuarios como productos pero siempre y cuando estos dos puedan ser asociados. Osea, como se ve en el gráfico de arriba, nos va a traer la intersección de estas dos tablas.

La consulta quedaría así:

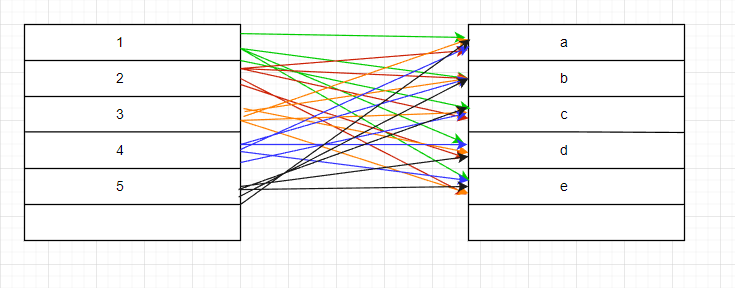
**select u.id, u.email, p.name from user u inner join products p on u.id = p.created\_by**; //



En este caso nos muestra todos los productos que se encuentra registrados, y es porque todos los productos que se encuentran registrados están asociados con un usuario.

**CROSS JOIN**

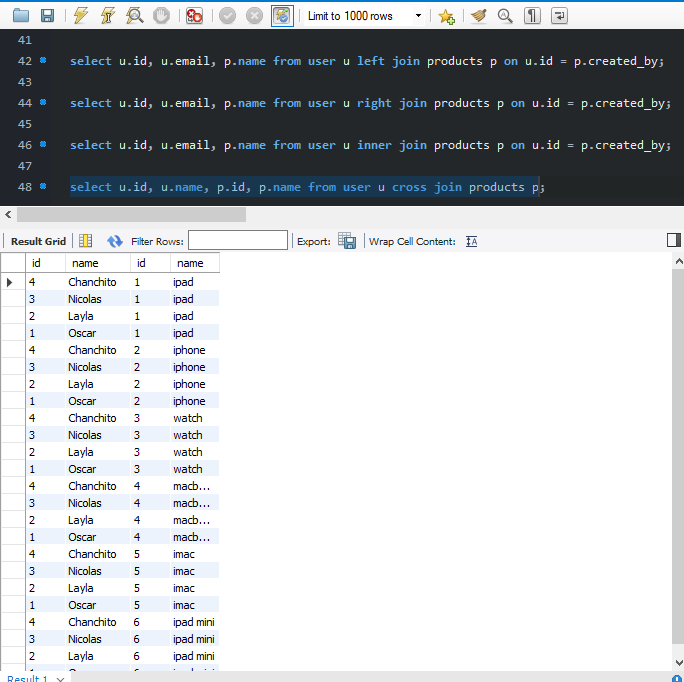
Te entrega el producto cartesiano entre todas las tablas. El **producto cartesiano seria juntar el producto de una tabla con todos los productos de otra tabla.**

****

**select u.id, u.name, p.id, p.name from user u cross join products p;**

**Debemos tener cuidado al usar cross join ya que los registros que nos va a devolver pueden ser bastante numerosos.**

**↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓**

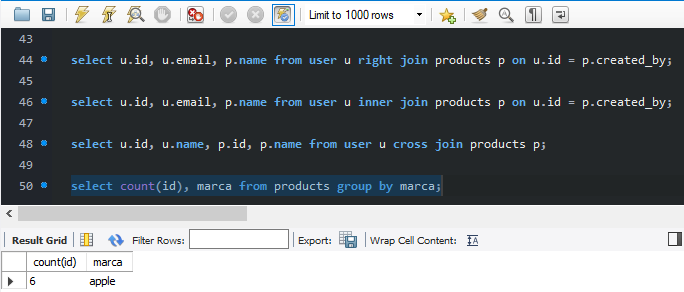
****

Nos devuelve todos los productos cruzados con usuarios.

**GROUP BY**

Se utiliza con instrucciones que por lo general tienden a agrupar elementos, por ejemplo, **count.** En este caso v a tomar todos los registros en base a un pivote, por ejemplo, **marca** y los va a agrupar en registros completamente individuales. En este caso nosotros tenemos 6 productos todos de la misma marca y por ende lo que hará será mostrarnos un registro y luego indicara que este, en total, son 6, pensando que son los productos que hemos ingresado hasta ahora. Si hiciéramos lo mismo con el usuario, nosotros tendríamos que esta instrucción tendría algo así como 3 productos para el primer usuario, 2 para el segundo, 2 para el tercero, y así sucesivamente, por ejemplo.

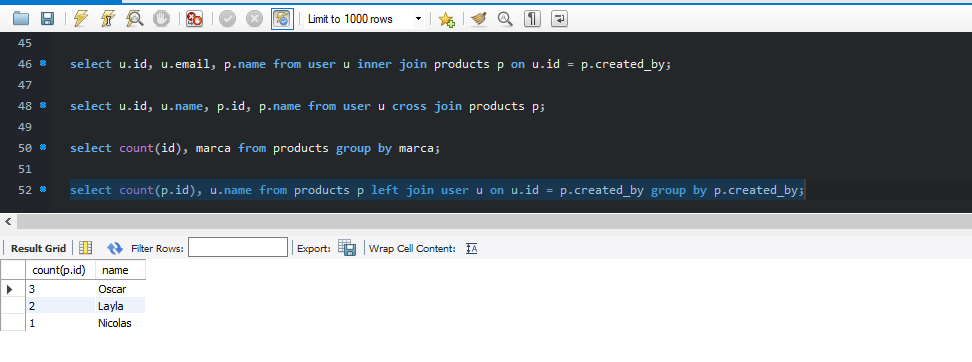
**select count(id), marca from products group by marca;**

****

Nos muestra que tenemos 6 productos de Apple.

Ahora, aprovecharemos para hacer un **left join** con la tabla de usuario, en este caso la tabla principal es la tabla de products (si quieren obtener el mismo resultado se puede hacer un right join, pero es a gusto propio)**:**

**select count(p.id), u.name from products p left join user u on u.id = p.created\_by group by p.created\_by;**

****

**Oscar creo 3 productos, Layla creo 2 productos, Nicolas creo 1 producto.**

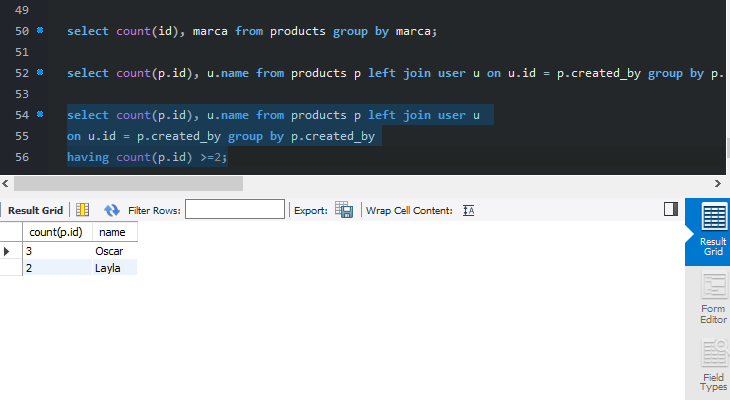
Nosotros podemos hace que estas consultas de **group by** solamente nos arroje resultados que para nosotros **sean relevantes**. Supongamos que queremos colocarle una **condición,** por ejemplo, que solamente nos devuelva los registros donde su conteo sean mayor o igual a 2.

En este caso, Nicolas no nos interesa ☹ y solo quiero que me devuelva el registro de Layla y de Oscar, con una nueva instrucción llamada **HAVING**:

*select count(p.id), u.name from products p left join user u*

*on u.id = p.created\_by group by p.created\_by*

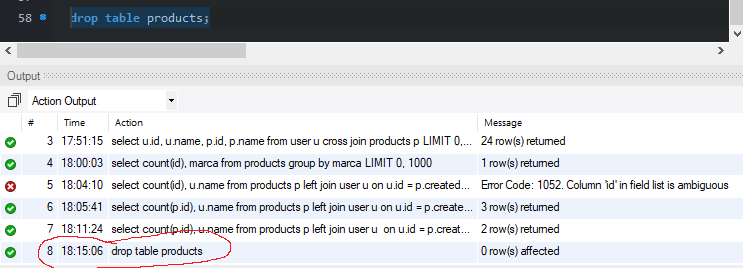
**having** *count(p.id) >=2;*

**

Nos arrojara a Oscar con 3 productos creados y tenemos a Layla que solamente creo 2 productos.

Ya hemos visto las instrucciones más importantes y la que vamos a utilizar de manera más recurrente con SQL, nos faltaría ver la última antes de pasar a **modelamiento de base de datos,** esta es **DROP TABLE,** la cual nos elimina una tabla, en este caso la de productos:

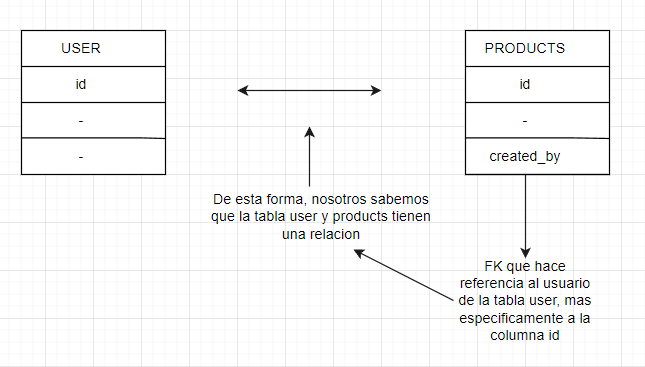
**drop table products;**

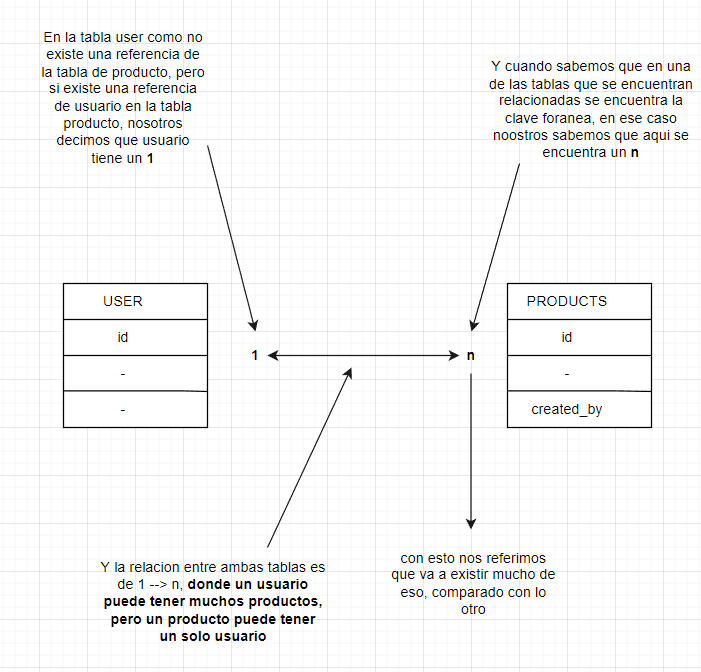
****

Podemos ver que nuestra tabla ha sido eliminada o dropeada con éxito.

***CARDINALIDAD***

Se refiere a las relaciones que pueden tener unas tablas con otras.





En este caso tenemos la **cardinalidad 1 🡨🡪n (uno a muchos)**

Otra cardinalidad que también existe es de **n🡨🡪n (muchos a muchos)**, por ejemplo, las órdenes de compra donde una puede contener muchos productos pero también un producto puede existir en múltiples ordenes de compra.

Nosotros podríamos hacer es tratar de colocar el **id** de la orden de compra en la tabla de **products**, pero estaríamos creando constantemente el mismo producto muchas veces si lo queremos asociar a la orden de compra. Lo mismo pasaría en la tabla de **order** si es que nosotros agregáramos un **product\_id**, ya que si quisiéramos agregar más productos a esta misma orden de compra (por ejemplo, en un supermercado), no nos va a servir. Por eso es que se crea una **tabla alterna** que contenga los nombres de ambas tablas (**product\_order**) y el **id** de cada una de las tablas a la que hace referencia.

