EVILCORP

Ian Nogueira

ÍNDICE

1-INTRODUCCIÓN

Para empezar este proyecto, primero hay que especificar unos cuantos puntos.

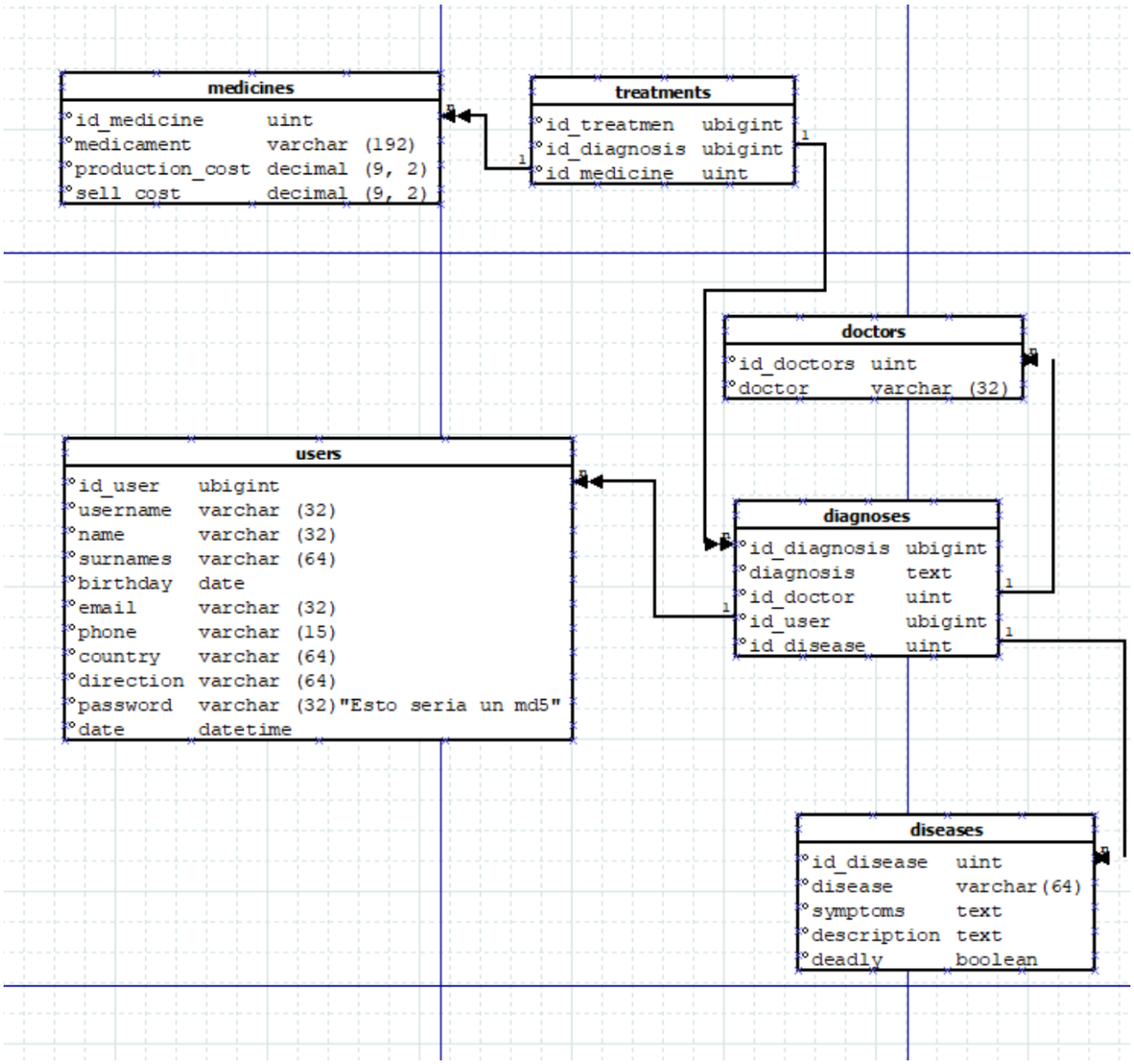
El proyecto de EvilCorp se basa en una base de datos que puede almacenar los datos de todo un universo entero. En esta base de datos hay varios apartados separados según la cronología de su creación. El primer apartado es el de HealthCare. Este apartado es la abstracción de un supuesto control total de la sanidad, es por eso que se almacenan los datos de médicos, medicinas, tratamientos, diagnósticos etc. Todo esto para luego sacar un supuesto beneficio a la hora de vender las medicinas para los tratamientos. El segundo apartado es el de Conspirations. En este apartado se trata todo el tema de conspiraciones como las estelas de los aviones, la negacion del holocausto y demas. Tiene como propósito almacenar todos los creyentes de estas conspiraciones para su futura eliminación. El tercer y último apartado es el más extenso, se trata de RealEstate. En este apartado se tratará todo lo relacionado con viviendas, direcciones y planetas. Todo esto para tener un mayor control sobre la población universal. Estos tres apartados son los que, por ahora, componen la base de datos de EvilCorp, una empresa creada por Hilon Musgo. En estos tres apartados encontraremos distintas funciones, views y distintos procedimientos con los que tratar los datos como el precio, la cantidad de personas que viven en un planeta y demás.

2-HealthCare

En este apartado se verán distintas tablas como medicines, treatments, diagnoses o doctors. Primero trataremos la estructura de la tabla.

2.1 DIA

La estructura en concreto es esta:



Como podemos ver, tenemos 6 tablas en este apartado, las cuales son:

USERS: *Esta tabla es la principal de toda la base de datos ya que estará en todos los apartados. Es una tabla que no entra en este proyecto ya que pertenece a uno anterior, por lo que omitiremos la explicación de sus campos y de su uso.*

MEDICINES: *En esta tabla se almacenarán las distintas medicinas que tendrá EvilCorp a su disposición. Contiene los campos de:*

- *Id_medicament: Campo universal en todas las tablas.*
- *Medicament: Este campo hace referencia al nombre del medicamento. Es un varchar de 192 caracteres debido a que el medicamento con el nombre más largo del mundo contiene 190 caracteres.*
- *Production_cost: Este campo es el coste que tendrá dicho medicamento en producirse. Es un decimal con 9 dígitos los cuales dos de ellos son decimales ya que los precios están muy abultados.*
- *Sell_cost: Este campo es el precio de venta de dicho medicamento. Es un decimal con 9 dígitos los cuales dos de ellos son decimales ya que los precios están muy abultados.*

DISEASES: *En esta tabla se almacenarán las distintas enfermedades que puedan existir o las que más interesen a EvilCorp. Contiene los campos de:*

- *Id_disease: Campo universal en todas las tablas.*
- *Disease: Este campo hace referencia al nombre de la enfermedad en cuestión. Es un varchar(64) por la posibilidad de algún nombre más largo de lo nombre.*

- *Symptoms: Este campo hace referencia a los síntomas de la enfermedad en concreto. Es un text ya que una enfermedad podría tener síntomas complejos los cuales no cabrían en un varchar.*
- *Description: Este campo hace referencia a la descripción de la enfermedad. Es un text ya que requiere de muchos caracteres el hacer una descripción.*
- *Deadly: Este campo hace referencia a si una enfermedad es mortal o no. Es un booleano ya que hace falta determinar true o false.*

DOCTORS: *Esta tabla es donde se guardaran los datos de los doctores que están a disposición de EvilCorp. Sus campos son:*

- *Id_doctor: Es un campo universal de las tablas.*
- *Doctor: Hace referencia al nombre del doctor. Es un varchar de 32 caracteres por que solo es el nombre y el más largo del mundo ocupa 27 caracteres.*

DIAGNOSES: *Esta tabla se encarga de almacenar los datos de un diagnóstico completo, el cual se compone de:*

- *Id_diagnosis: Es un campo universal de las tablas.*
- *Diagnosis: Una explicación del diagnóstico. Es un text ya que requiere de muchos caracteres el explicar algo de estas características.*
- *Id_doctor: Este campo hace referencia al campo id_doctor de la tabla de doctors. Esto es ya que queremos unir un doctor que ha realizado el diagnóstico.*
- *Id_user: Este campo hace referencia al campo id_user de la tabla de users. Esto es ya que queremos juntar el usuario que recibe el diagnóstico.*

- *Id_disease: Este campo hace referencia al campo de id_disease de la tabla diseases. Esto es ya que queremos juntar la enfermedad en concreto a tratar.*

TREATMENTS: *Esta tabla la usaremos para juntar una medicina en concreto y un diagnóstico. La tabla está compuesta de estos campos:*

- *Id_treatment: Campo universal de las tablas.*
- *Id_diagnosis: Este campo hace referencia a id_diagnosis de la tabla de diagnoses. Esto es por que queremos juntar un diagnóstico concreto con la medicina necesaria para tratar la enfermedad.*
- *Id_medicine: este campo hace referencia a id_medicine de la tabla de medicines. Esto es por que queremos juntar la medicina correcta para el diagnóstico que necesite esta medicina.*

2.2 SCRIPT

Ahora pasaremos a la creación del script para hacer este apartado en la base de datos. Primero comenzamos haciendo drops a las tablas en el orden de uso respecto a las foreign keys. Para saber el orden simplemente hay que mirar que tablas están relacionadas y saber que tabla no depende de ninguna otra.

2.2.1 DROPS

Estos son los drops.

```
drop table if exists treatments;
drop table if exists diagnoses;
drop table if exists `users`;
```

```
drop table if exists diseases;
drop table if exists medicines;
drop table if exists doctors;
DROP VIEW IF EXISTS costes;
```

Como podemos ver, primero van treatments y diagnoses. Esto es por que no hay ninguna otra tabla que use treatments, pero treatments si usa diagnoses, es por eso que va antes. Diagnoses va después porque la única tabla que depende de ella es treatments y la única que usa es users, por lo que hay que eliminarla una de las primeras, al menos antes que users. El resto no dependen de ninguna ni las usan otras tablas. Al final de los drops he puesto el drop de la única view que hay en el apartado. Esto es para guardar el orden.

Luego de los DROPS viene la creación de las tablas.

2.2.2 CREATES

Los creates son estos:

```
create table `users`(id_user BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, name VARCHAR(64) NOT NULL);

create table diseases (id_disease INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, disease VARCHAR(64) NOT NULL, symptoms TEXT NOT NULL, description TEXT NOT NULL, deadly BOOLEAN NOT NULL);

create table medicines (id_medicine INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, medicine VARCHAR(192) NOT NULL,
```

```
cost_production DECIMAL(9, 2) UNSIGNED NOT NULL, cost_sell
DECIMAL(9, 2) UNSIGNED NOT NULL);

create table doctors (id_doctor INT UNSIGNED NOT NULL
AUTO_INCREMENT PRIMARY KEY, doctor VARCHAR(32));

create table diagnoses (id_diagnosis BIGINT UNSIGNED NOT
NULL AUTO_INCREMENT PRIMARY KEY, diagnosis TEXT NOT NULL,
id_doctor INT UNSIGNED NOT NULL, id_user BIGINT UNSIGNED
NOT NULL, id_disease INT UNSIGNED NOT NULL, FOREIGN
KEY(id_doctor) REFERENCES doctors(id_doctor), FOREIGN
KEY(id_user) REFERENCES users(id_user), FOREIGN
KEY(id_disease) REFERENCES diseases(id_disease));

create table treatments (id_treatment BIGINT UNSIGNED NOT
NULL AUTO_INCREMENT PRIMARY KEY, id_diagnosis BIGINT
UNSIGNED NOT NULL, id_medicine INT UNSIGNED NOT NULL,
FOREIGN KEY(id_diagnosis) REFERENCES
diagnoses(id_diagnosis), FOREIGN KEY(id_medicine)
REFERENCES medicines(id_medicine));
```

Como podemos ver, los creates van en el orden contrario de los drops. Esto es por que para hacer una foreign key, ha de existir el campo con el que se está haciendo dicha key. Así que primero se hacen las tablas que serán usadas para el resto y luego las tablas que usan dichas tablas.

2.2.3 INSERTS

Ahora, sin mucha explicación, pondremos los inserts.

```
INSERT INTO users (`name`) VALUES ("Paco"),("Manolo"),("Francisco"),("Sergio"),("Paca"),("Manola"),("Francisca"),("Sergia");

insert into doctors(doctor) values ("Belcebu"), ("Satanas");

insert into medicines(medicine, cost_production, cost_sell) values ("Simvastatina", 921323.23, 9999999.99), ("Aspirina", 5000.99, 999999.99), ("Omeprazol", 7000.00, 999999.99), ("Ramipril", 96543.00, 9999999.22);

insert into diseases(disease, symptoms, description, deadly) values ("Bronquitis aguda", "Tos, Mucosidad", "Te pones mu malo de los bronquio",true), ("Resfriado comun", "Estornudos, moqueos, congestion nasal, tos, dolor de garganta", "Te pones malo por haber pillado frio", true), ("Infeccion de oido", "Dolor de oido, fiebre", "No oyes na", true), ("Gripe", "Fiebre, tos, dolor de gargatna, moqueo, dolor corporal", "Excusa perfecta para faltar a todos los sitios", true), ("Sinusitis", "Dolor de cabeza, congestion, moqueo, presion en la cara", "Te revienta la cabeza", true), ("Infeccion de piel", "Erojecimiento de la piel, inflamacion", "El popper", true), ("Dolor de garganta", "Dolor de garganta, ronquera", "Algo que el pablo no tendra", true);
```

```
insert into diagnoses(diagnosis, id_doctor, id_user, id_disease) values("Esta mu malo", 1, 2, 3), ("Esta mas malo aun", 2, 3, 1), ("Esta malo", 1, 2, 1), ("Esta malo maloso", 1, 1, 7);
```

```
insert into treatments(id_diagnosis, id_medicine) values(1,1),(1,2),(2,3),(4,3),(3, 4);
```

2.2.4 EXTRAS

En este caso, el único extra que hay es una view con la que comprobaremos todos los precios de las medicinas y el beneficio que generamos. La view es esta:

```
create view costes as select treatments.id_medicine, COUNT(medicine) medicines, cost_production, SUM(cost_production) total_cost_production, cost_sell, SUM(cost_sell) total_cost_sell from medicines LEFT JOIN treatments on medicines.id_medicine=treatments.id_medicine group by cost_production;
```

```
MariaDB [Healthcare]> create view costes as select treatments.id_medicine, COUNT(medicine) medicines, cost_production, SUM(cost_produc
ents.id_medicine group by cost_production;
Query OK, 0 rows affected (0.007 sec)

MariaDB [Healthcare]> select * from costes;
+-------------+-----------+-----------------+-----------------------+------------+-----------------+
| id_medicine | medicines | cost_production | total_cost_production | cost_sell  | total_cost_sell |
+-------------+-----------+-----------------+-----------------------+------------+-----------------+
|           2 |         1 |         5000.99 |               5000.99 |  999999.99 |       999999.99 |
|           3 |         2 |         7000.00 |              14000.00 |  999999.99 |      1999999.98 |
|           4 |         1 |        96543.00 |              96543.00 | 9999999.22 |      9999999.22 |
|           1 |         1 |       921323.23 |             921323.23 | 9999999.99 |      9999999.99 |
+-------------+-----------+-----------------+-----------------------+------------+-----------------+
4 rows in set (0.001 sec)
```

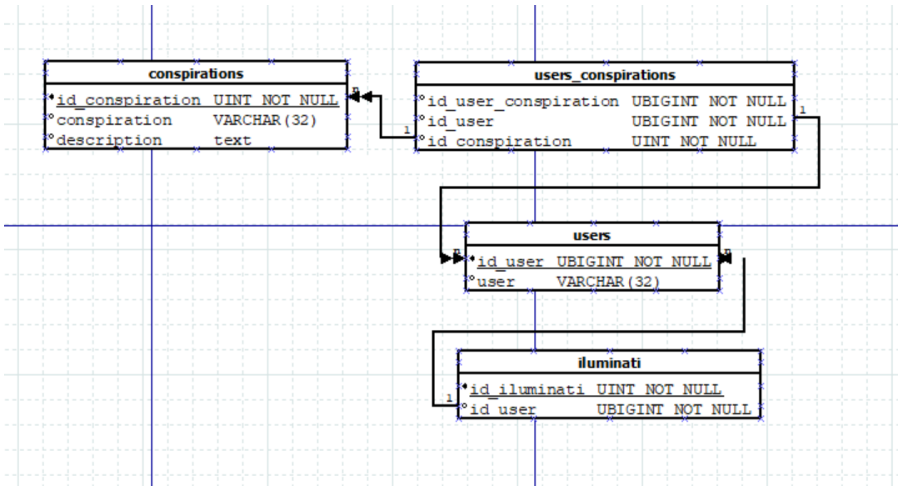
Con esta view podremos comprobar en cualquier momento el profit que haremos ya que suma de manera separada las medicinas que hay y hace la resta del coste de producción con el precio de venta.

3-Conspirations

Aqui vemos distintas tablas como conspirations o iluminatis. Este apartado es el más pequeño de los tres. Su estructura es la siguiente.

3.1 DIA

La estructura de este apartado es la siguiente:



Como podemos ver en la captura de pantalla, tenemos 4 tablas. Las cuales son:

USERS: *Esta tabla es la principal de toda la base de datos ya que estará en todos los apartados. Es una tabla que no entra en este proyecto ya que pertenece a uno anterior, por lo que omitiremos la explicación de sus campos y de su uso.*

ILUMINATIS: *En esta tabla solo se almacenaran los usuarios que son illuminatis. Sus campos son:*

- *Id_iluminati: Campo universal de las tablas.*

- *Id_user: Campo que hace referencia a id_user de la tabla users. Esto es ya que es necesario saber el usuario que es iluminati.*

CONSPIRATIONS: *En esta tabla guardaremos las conspiraciones que hay hoy en día. Se compone de:*

- *Id_conspiration: Campo universal de las tablas.*
- *Conspiration: Este campo es el nombre de la conspiración en concreto. Es un varchar de 32 caracteres ya que no hacen falta tantos caracteres para nombrar dichas conspiraciones.*
- *Description: Este campo será la descripción de la conspiración en cuestión. Es un text porque son necesarios muchos caracteres para rellenar una descripción.*

USER_CONSPIRATIONS: *Esta tabla almacenará las conspiraciones y los usuarios que creen en ellas. Esta compuesta por:*

- *Id_user_conspiration: Campo universal en todas las tablas.*
- *Id_user: Este campo hace referencia a id_user de la tabla de users. Esto es ya que queremos almacenar el usuario que cree en la conspiración.*
- *Id_conspiration: Este campo hace referencia a id_conspiration de la tabla de conspirations. Esto es por que necesitamos almacenar las conspiraciones que existen para luego relacionarlas con los usuarios.*

3.2 SCRIPT

Para hacer el script de este apartado solamente tendremos que tener en cuenta las foreign keys y el hacer los drops correctamente de las tablas y las funciones. Luego tendremos que hacer correctamente los procedimientos y las views.

Primero comenzaremos con los drops.

3.2.1 DROPS

Los drops son los siguientes:

```
DROP VIEW IF EXISTS ilumi_show;
DROP TABLE IF EXISTS iluminatis;
DROP TABLE IF EXISTS users_conspirations;
DROP TABLE IF EXISTS conspirations;
DROP PROCEDURE IF EXISTS add_conspiration;
DROP VIEW IF EXISTS ilumi_show;
DROP VIEW IF EXISTS ilumi_count;
```

En este caso tenemos mas drops de views y procedimientos que de tablas. El primer drop es una view ya que utiliza una tabla que tiene una foreign key, y como las views son tablas artificiales cuentan en el orden de eliminación.

Despues de esto, vienen los creates.

3.2.2 CREATES

Los creates son estos:

```
CREATE TABLE conspirations (id_conspiration INT UNSIGNED
NOT NULL AUTO_INCREMENT PRIMARY KEY, conspiration
VARCHAR(32) NOT NULL, description TEXT NOT NULL);

CREATE TABLE iluminatis (id_iluminati INT UNSIGNED NOT NULL
AUTO_INCREMENT PRIMARY KEY, id_user BIGINT UNSIGNED NOT
NULL, FOREIGN KEY(id_user) REFERENCES users(id_user));
```

```
CREATE TABLE users_conspirations (id_user_conspiration BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, id_user BIGINT UNSIGNED NOT NULL, id_conspiration INT UNSIGNED NOT NULL, FOREIGN KEY(id_user) REFERENCES users (id_user), FOREIGN KEY(id_conspiration) REFERENCES conspirations(id_conspiration));
```

Como podemos ver, solamente hay 3 tablas, una de ellas la tabla intermedia de users_conspirations, que usa la tabla principal de users. Esta última ya que usa la tabla de conspirations y hay que respetar el orden de creación.

3.2.3 INSERTS

Ahora pondremos los inserts.

```
INSERT INTO conspirations (conspiration, description) VALUES ("No Holo", "Los muy nazis"), ("Terraplanistas", "Cosas de planos"), ("Conspiracion", "No covid");

INSERT INTO iluminatis (id_user) VALUES (1), (3);
```

3.2.4 EXTRAS

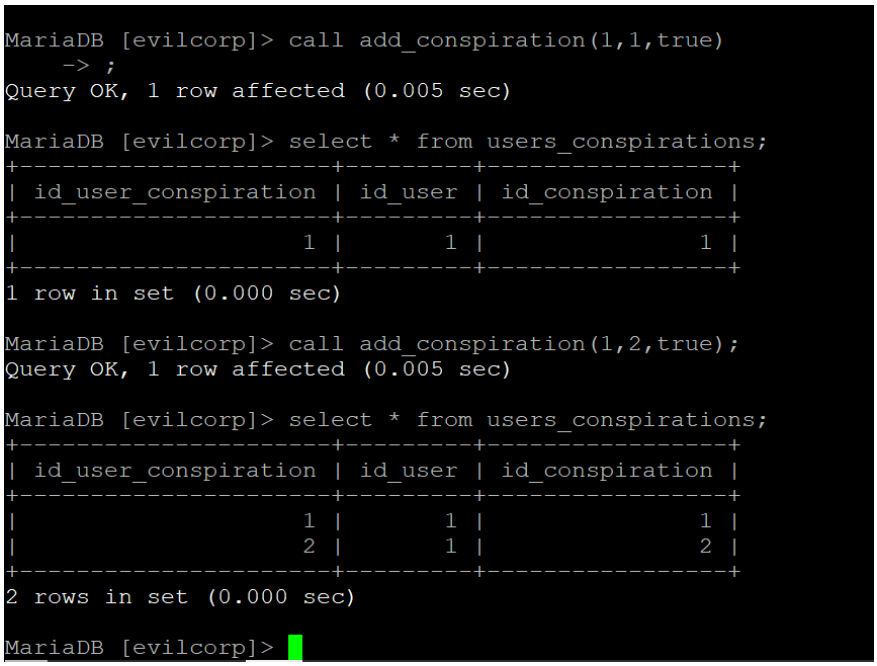
En este caso, tenemos dos views y una función. El codigo de estas views y de la función es el siguiente:

```
CREATE PROCEDURE add_conspiration( IN id_users BIGINT, IN id_conspirations INT, IN opinion BOOLEAN)
BEGIN

START TRANSACTION;
```

```
INSERT INTO users_conspirations(id_user, id_conspiration)
VALUES (id_users, id_conspirations);

IF opinion THEN
COMMIT;
ELSE
ROLLBACK;
END IF;

END $$

DELIMITER ;
```

Primero tenemos el procedimiento de add_conspiration. Este procedimiento lo que hace es añadir en la tabla de users_conspiration al usuario que le hemos pasado con la conspiración que le hemos pasado, dependiendo de si le pasamos true o false añadirá a la persona o no. Para hacer esto usaremos una transacción la cual dependerá de si el último valor que le hemos pasado es true o false.

Caso de que le pongamos true:



Caso que le pongamos false:

```
MariaDB [evilcorp]> select * from users_conspirations;
+----------------------+---------+-----------------+
| id_user_conspiration | id_user | id_conspiration |
+----------------------+---------+-----------------+
|                    1 |       1 |               1 |
|                    2 |       1 |               2 |
+----------------------+---------+-----------------+
2 rows in set (0.000 sec)

MariaDB [evilcorp]> call add_conspiration(2,2,false);
Query OK, 1 row affected (0.002 sec)

MariaDB [evilcorp]> select * from users_conspirations;
+----------------------+---------+-----------------+
| id_user_conspiration | id_user | id_conspiration |
+----------------------+---------+-----------------+
|                    1 |       1 |               1 |
|                    2 |       1 |               2 |
+----------------------+---------+-----------------+
2 rows in set (0.000 sec)
```

```
CREATE VIEW ilumi_show AS SELECT users.id_user, users.name FROM users LEFT JOIN iluminatis ON users.id_user = iluminatis.id_user;
```

La view de ilumi_show lo que hace es mostrar que usuarios son iluminatis. Esto lo hacemos con un select de la tabla iluminatis y un filtro en el id_user.

```
MariaDB [evilcorp]> select * from iluminatis;
+--------------+---------+
| id_iluminati | id_user |
+--------------+---------+
|            1 |       1 |
|            2 |       3 |
+--------------+---------+
2 rows in set (0.000 sec)

MariaDB [evilcorp]> select * from ilumi_show;
+---------+-----------+
| id_user | name      |
+---------+-----------+
|       1 | Paco      |
|       3 | Francisco |
+---------+-----------+
2 rows in set (0.001 sec)
```

```
CREATE VIEW ilumi_count AS SELECT COUNT(users.id_user) users FROM users,ilumi_show;
```

La view de ilumi_count lo que hace es contar cuantos illuminatis hay, para eso se hace un count de la view de ilumi_show, la view anterior.

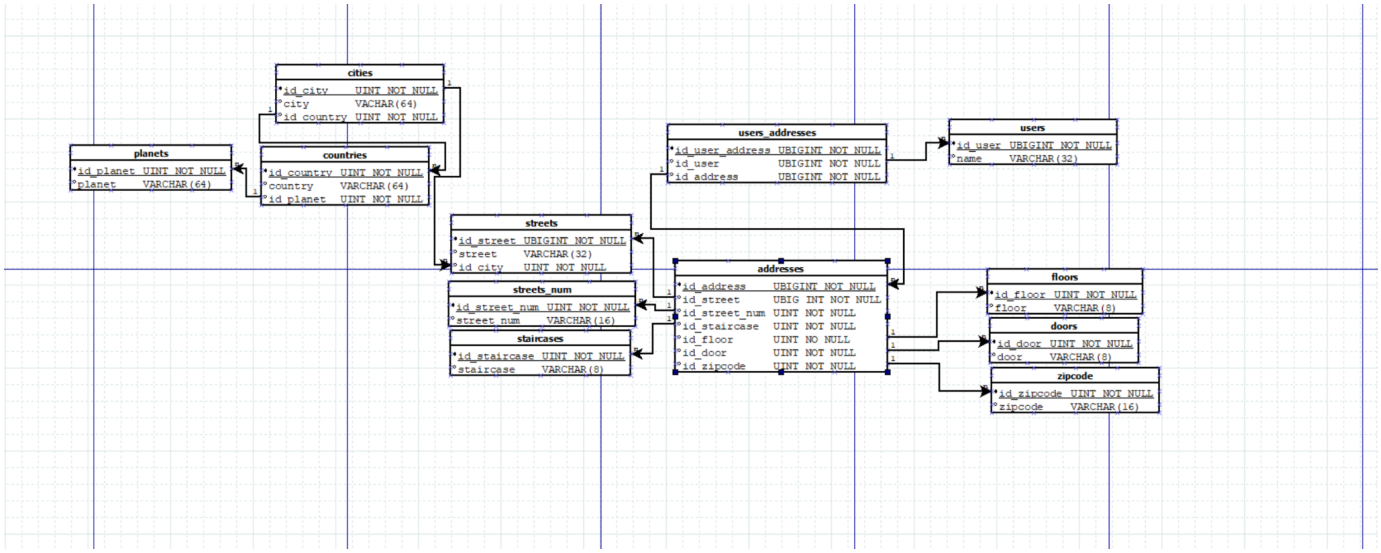
```
MariaDB [evilcorp]> select * from ilumi_show;
+---------+-----------+
| id_user | name      |
+---------+-----------+
|       1 | Paco      |
|       3 | Francisco |
+---------+-----------+
2 rows in set (0.001 sec)

MariaDB [evilcorp]> select * from ilumi_count;
+-------+
| users |
+-------+
|     2 |
+-------+
1 row in set (0.001 sec)

MariaDB [evilcorp]>
```

4-RealState

Este apartado es el más extenso y el más complejo de los tres. Podremos ver tablas como streets, addresses y muchas más. Todas estas tablas están conectadas entre ellas para optimizar posibles campos vacíos y funciones para contar direcciones. La estructura en concreto es esta:

4.1 DIA

La estructura es esta:



Como podemos ver hay 12 tablas, es el apartado más extenso y más complejo. Las tablas en concreto son:

USERS: *Esta tabla es la principal de toda la base de datos ya que estará en todos los apartados. Es una tabla que no entra en este proyecto ya que pertenece a uno anterior, por lo que omitiremos la explicación de sus campos y de su uso.*

PLANETS: *En esta tabla se almacenan los planetas que hay a disposición de EvilCorp. Está compuesto de estos campos:*

- *Id_planets: Campo universal en las tablas.*
- *Planets: Este campo es el nombre del planeta en concreto. Es un VARCHAR de 64 ya que hay nombres de planetas que pueden tener más de 34 caracteres.*

COUNTRIES: *Este campo almacena los países de todos los planetas de EvilCorp. Está compuesto por:*

- *Id_countries: Campo universal de las tablas.*

- *Country: Este campo es el nombre del país en cuestión. Es un VARCHAR de 64 caracteres porque hay países que pueden tener esa cantidad de caracteres.*
- *Id_planet: Este campo hace referencia a id_planet de la tabla planets. Esto es por que queremos almacenar de qué planeta pertenece este país.*

***CITIES**: Este campo almacena las ciudades. Está compuesto por:*

- *Id_cities: Campo universal de las tablas.*
- *City: Este campo es el nombre de la ciudad en concreto. Es un VARCHAR de 64 caracteres por que hay ciudades que pueden tener estos nombres tan largos.*
- *Id_country: Este campo hace referencia a id_country de la tabla countries. Esto es porque queremos saber a qué país pertenece esta ciudad.*

***STREETS**: Este campo almacena todas las calles. Esta compuesta por:*

- *Id_street: Campo universal de las tablas.*
- *Street: Almacena el nombre de la calle o los números de la calle. Es un VARCHAR de 32 caracteres por que hay calles que pueden ocupar 30 caracteres.*
- *Id_city: Este campo hace referencia a id_city de la tabla cities. Esto es porque queremos saber a qué ciudad pertenece esta calle.*

***STREETS_NUM**: Esta tabla almacena los números de las calles. Está compuesta por:*

- *Id_street_num: Campo universal en las tablas.*

- *Street_num: Este campo es el número de la calle. Es un VARCHAR de 16 caracteres para almacenar el número más grande posible.*

STAIRCASE: *Esta tabla almacena los números de escalera. Esta compuesta por:*

- *Id_staircase: Campo universal en las tablas.*
- *Staircase: Este campo almacena la escalera en concreto. Es un VARCHAR de 8 caracteres para almacenar todas las escaleras posibles.*

FLOORS: *Esta tabla almacena los pisos de los edificios. Esta compuesta por:*

- *Id_floor: Campo universal en las tablas.*
- *Floor: Este campo almacena el piso en concreto. Es un VARCHAR de 8 caracteres para almacenar los posibles pisos.*

DOORS: *Esta tabla almacena todas los números de puerta. Esta compuesta por:*

- *Id_door: Campo universal en las tablas.*
- *Door: Campo que almacena el número de puerta. Es un VARCHAR de 8 caracteres para almacenar todos los posibles números.*

ZIPCODES: *Esta tabla almacena todos los códigos postales posibles. Esta compuesta por:*

- *Id_zipcode: Campo universal en las tablas.*
- *Zipcode: Este campo almacena el código postal. Es un VARCHAR 16 para almacenar todos los posibles códigos postales.*

ADDRESSES: *Esta tabla almacena el conjunto de calle, número de calle, piso, puerta y código postal. Está compuesta por:*

- *Id_address: Campo universal en las tablas.*
- *Id_street: Este campo hace referencia a id_street de la tabla streets. Esto es porque queremos saber a qué calle pertenece esta dirección.*
- *Id_street_num: Este campo hace referencia a id_street_num de la tabla streets_num. Esto es porque queremos saber a qué número de la calle pertenece esta dirección.*
- *Id_staircase: Este campo hace referencia a id_staircase de la tabla staircases. Esto es porque queremos saber a qué escalera pertenece esta dirección.*
- *Id_floor: Este campo hace referencia a id_floor de la tabla floors. Esto es porque queremos saber a qué piso pertenece esta dirección.*
- *Id_door: Este campo hace referencia a id_door de la tabla doors. Esto es porque queremos saber a qué puerta pertenece esta dirección.*
- *Id_zipcode: Este campo hace referencia a id_zipcode de la tabla zipcodes. Esto es porque queremos saber qué código postal pertenece esta dirección.*

USERS_ADDRESSES: *Esta tabla almacena los usuarios y sus direcciones. Esta compuesta por:*

- *Id_user_address: Campo universal para las tablas.*
- *Id_user: Este campo hace referencia a id_user de la tabla users. Esto es porque queremos saber a qué usuario pertenece la dirección.*
- *Id_address: Este campo hace referencia a id_address de la tabla addresses. Esto es porque queremos saber qué direcciónes pertenecen a algún usuario.*
-

4.2 SCRIPT

En este caso vemos que para hacer este apartado necesitamos hacer muchas tablas y más funciones, views y procedimientos que los anteriores apartados. Aparte, tendremos que tener más cuidado con los drops, ya que hay muchas tablas que dependen de otras y si no vamos con cuidado, dará todo el rato error a la hora de ejecutar el script. Comenzamos con los drops.

4.2.1 DROPS

Estos son los drops:

```
DROP TABLE IF EXISTS users_addresses;
DROP TABLE IF EXISTS users_planets;
DROP TABLE IF EXISTS addresses;
DROP TABLE IF EXISTS streets;
DROP TABLE IF EXISTS cities;
DROP TABLE IF EXISTS countries;
DROP TABLE IF EXISTS streets_num;
DROP TABLE IF EXISTS staircases;
DROP TABLE IF EXISTS floors;
DROP TABLE IF EXISTS doors;
DROP TABLE IF EXISTS zipcodes;
DROP TABLE IF EXISTS planets;
DROP VIEW IF EXISTS planet_addresses;
DROP VIEW IF EXISTS count_users_planet;
DROP VIEW IF EXISTS planet_medicine_production;
DROP FUNCTION IF EXISTS get_city;
DROP FUNCTION IF EXISTS random_user;
DROP PROCEDURE IF EXISTS killer;
```

Como podemos ver, primeramente hacemos drop de todas las tablas intermedias para después ir borrando las tablas en el orden contrario de cómo se han creado. Esto hará que no tengamos ninguna foreign key mal borrada ni ningún error que nos pueda perjudicar más tarde. Al final de todo están las views, funciones y procedimientos.

4.2.2 CREATES

Los creates de este apartado son estos.

```
CREATE TABLE planets (id_planet INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, planet VARCHAR(64) NOT NULL);

CREATE TABLE countries (id_country INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, country VARCHAR(64) NOT NULL, id_planet INT UNSIGNED NOT NULL, FOREIGN KEY(id_planet) REFERENCES planets(id_planet));

CREATE TABLE cities (id_city INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, city VARCHAR(64) NOT NULL, id_country INT UNSIGNED NOT NULL, FOREIGN KEY(id_country) REFERENCES countries(id_country));

CREATE TABLE streets (id_street BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, street VARCHAR(32) NOT NULL, id_city INT UNSIGNED NOT NULL, FOREIGN KEY(id_city) REFERENCES cities(id_city));

CREATE TABLE streets_num (id_street_num INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, street_num VARCHAR(16));
```

```
CREATE TABLE staircases (id_staircase INT UNSIGNED NOT NULL
PRIMARY KEY AUTO_INCREMENT, staircase VARCHAR(8));

CREATE TABLE floors (id_floor INT UNSIGNED NOT NULL PRIMARY
KEY AUTO_INCREMENT, `floor` VARCHAR(8));

CREATE TABLE doors (id_door INT UNSIGNED NOT NULL PRIMARY
KEY AUTO_INCREMENT, door VARCHAR(8));

CREATE TABLE zipcodes (id_zipcode INT UNSIGNED NOT NULL
PRIMARY KEY AUTO_INCREMENT, zipcode VARCHAR(16) NOT NULL);

CREATE TABLE addresses (id_address BIGINT UNSIGNED NOT NULL
PRIMARY KEY AUTO_INCREMENT,
id_street BIGINT UNSIGNED NOT NULL,
id_street_num INT UNSIGNED NOT NULL,
id_staircase INT UNSIGNED NOT NULL,
id_floor INT UNSIGNED NOT NULL,
id_door INT UNSIGNED NOT NULL,
id_zipcode INT UNSIGNED NOT NULL,
FOREIGN KEY(id_street) REFERENCES streets(id_street),
FOREIGN KEY(id_street_num) REFERENCES
streets_num(id_street_num),
FOREIGN KEY(id_staircase) REFERENCES
staircases(id_staircase),
FOREIGN KEY(id_floor) REFERENCES floors(id_floor),
FOREIGN KEY(id_door) REFERENCES doors(id_door),
FOREIGN KEY(id_zipcode) REFERENCES zipcodes(id_zipcode))
ENGINE=InnoDB;

CREATE TABLE users_addresses (id_user_address BIGINT
UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, id_user
```

```
BIGINT UNSIGNED NOT NULL, id_address BIGINT UNSIGNED NOT NULL, FOREIGN KEY(id_user) REFERENCES users(id_user), FOREIGN KEY (id_address) REFERENCES addresses(id_address)) ENGINE=InnoDB;
```

```
CREATE TABLE users_planets (id_user_planet BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT, id_user BIGINT UNSIGNED NOT NULL, id_planet INT UNSIGNED NOT NULL, FOREIGN KEY(id_user) REFERENCES users(id_user), FOREIGN KEY(id_planet) REFERENCES planets(id_planet));
```

Como podemos ver, primero creamos la tabla planets, countries y cities por qué se usan en la tabla de streets, la cual se usa en la tabla de addresses. Dicha tabla de addresses se crea al final ya que es necesario que existan todas las tablas que usa para que no de un error. Dicho error es que intenta relacionar tablas que no existen. Luego se crean las tablas intermedias entre addresses y users, y planets y users.

4.2.3 INSERTS

Los inserts son estos:

```
INSERT INTO planets(planet) VALUES ("Mercurio"),("Venus"), ("Tierra"), ("Marte");
```

```
INSERT INTO countries(country, id_planet) VALUES ("Sudan del Sur", 3), ("Zambia", 3), ("Lumele", 1), ("Titutul", 2), ("Ju", 4), ("So", 4), ("Z012", 1), ("Vietnam", 3);
```

```
INSERT INTO cities(city, id_country) VALUES ("A", 1), ("B", 2), ("C", 3), ("D", 4), ("E", 5), ("F", 6), ("G", 7), ("H", 8);

INSERT INTO streets(street, id_city) VALUES ("Calle S", 1), ("Calle I", 2), ("Calle D", 3), ("Calle O", 4), ("Calle S", 5), ("Calle O", 6), ("Calle S", 7), ("Calle :)", 8), ("Calle A", 1), ("Calle L", 2), ("Calle E", 3), ("Calle G", 4), ("Calle R", 5), ("Calle I", 6), ("Calle A", 7);

INSERT INTO streets_num (street_num) VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13),(14),(15);

INSERT INTO staircases (staircase) VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13),(14),(15);

INSERT INTO floors (`floor`) VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13),(14),(15);

INSERT INTO doors (door) VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13),(14),(15);

INSERT INTO zipcodes (zipcode) VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13),(14),(15);

INSERT INTO addresses (id_street, id_street_num, id_staircase, id_floor, id_door, id_zipcode) VALUES
```

```
(1,1,1,1,1,1), (2,2,2,2,2,2), (3,3,3,3,3,3), (4,4,4,4,4,4),
(5,5,5,5,5,5), (6,6,6,6,6,6), (7,7,7,7,7,7), (8,8,8,8,8,8),
(9,9,9,9,9,9), (10,10,10,10,10,10), (11,11,11,11,11,11),
(12,12,12,12,12,12), (13,13,13,13,13,13),
(14,14,14,14,14,14),(15,15,15,15,15,15);

INSERT INTO users (name) VALUES
(1),(2),(3),(4),(5),(6),(7),(8);

INSERT INTO users_addresses(id_user, id_address) VALUES
(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(1,9),(2,10
),(3,11),(4,12),(5,13),(6, 14),(7,15);

INSERT INTO users_planets(id_user, id_planet) VALUES
(1,1),(2,2),(3,3);
```

4.2.4 EXTRAS

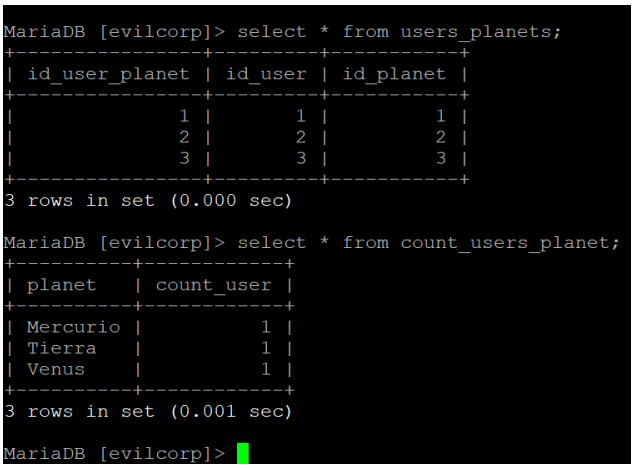
Este apartado es el que tiene más views, funciones y procedimientos, con un total de 3 views, 2 funciones y 1 procedimiento. Este es el código de estas views, funciones y procedimientos:

```
CREATE VIEW planet_addresses AS SELECT planets.planet,
COUNT(*) planet_addresses FROM planets, addresses, users,
users_planets, users_addresses WHERE
users.id_user=users_planets.id_user AND
users_planets.id_planet=planets.id_planet AND
users.id_user=users_addresses.id_user AND
addresses.id_address=users_addresses.id_address GROUP BY
planets.planet;
```

Está view lo que hace es contar todas las direcciones que hay en cada planeta. Para hacer eso lo que hacemos es poner multiples filtros que junten las tablas que relacionan los planetas, los usuarios y las direcciones. Con esos filtros conseguimos acotar lo que devuelve el select a únicamente las cantidad de direcciones que tiene cada planeta. Para contar lo que hacemos es contar las direcciones de los usuarios que hay en cada planeta. Con el group by conseguimos que muestre más de un campo

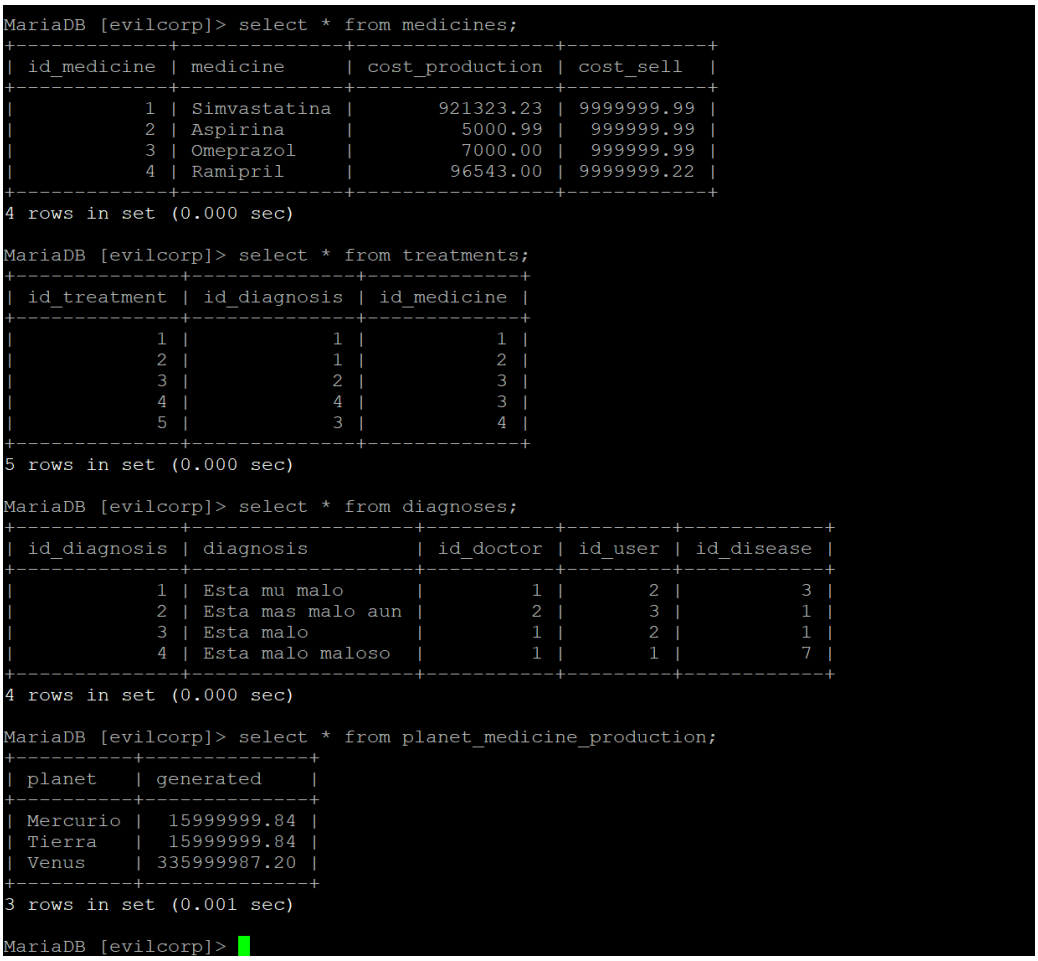
```
CREATE VIEW count_users_planet AS SELECT planets.planet, COUNT(users_planets.id_user) count_user FROM users, planets, users_planets WHERE users.id_user=users_planets.id_user AND planets.id_planet = users_planets.id_planet GROUP BY planets.planet;
```

Está view lo que hace es contar cuántas personas hay en cada planeta. Para hacer eso utilizamos los filtros para que solamente nos muestran las ids de los usuarios que coincidan en la tabla intermedia con la tabla de usuarios y los planetas que coincidan en la tabla intermedia y en la tabla de planetas. Con el group by hacemos que muestre más de un campo.



```
CREATE VIEW planet_medicine_production AS SELECT planets.planet, SUM(cost_sell) generated FROM users, medicines, diagnoses, planets, users_planets, treatments WHERE planets.id_planet=users_planets.id_planet AND diagnoses.id_user=users_planets.id_user AND diagnoses.id_diagnosis=treatments.id_diagnosis AND medicines.id_medicine=treatments.id_medicine GROUP BY planets.planet;
```

Está view lo que hace es contar cuánto genera cada planeta en medicinas. Para hacer eso utilizamos la misma técnica que antes. Gracias a los filtros, podemos ir acotando los resultados determinando los usuarios, las medicinas, los planetas, los tratamientos y los diagnósticos.

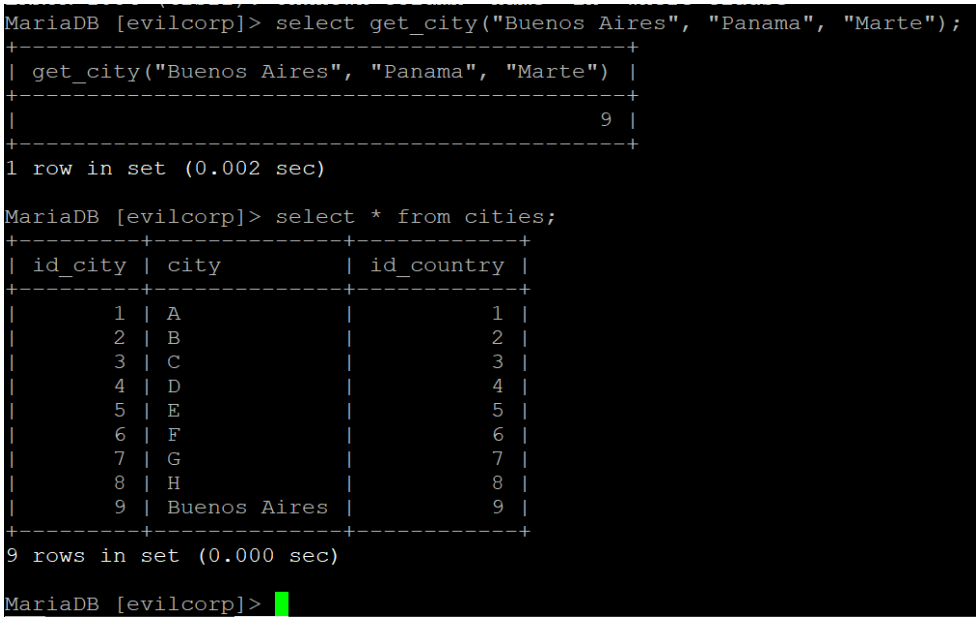


```
MariaDB [evilcorp]> select * from medicines;
+-------------+-------------+-----------------+------------+
| id_medicine | medicine    | cost_production | cost_sell  |
+-------------+-------------+-----------------+------------+
|           1 | Simvastatina |       921323.23 | 9999999.99 |
|           2 | Aspirina    |         5000.99 |  999999.99 |
|           3 | Omeprazol   |         7000.00 |  999999.99 |
|           4 | Ramipril    |        96543.00 | 9999999.22 |
+-------------+-------------+-----------------+------------+
4 rows in set (0.000 sec)

MariaDB [evilcorp]> select * from treatments;
+--------------+--------------+-------------+
| id_treatment | id_diagnosis | id_medicine |
+--------------+--------------+-------------+
|            1 |            1 |           1 |
|            2 |            1 |           2 |
|            3 |            2 |           3 |
|            4 |            4 |           3 |
|            5 |            3 |           4 |
+--------------+--------------+-------------+
5 rows in set (0.000 sec)

MariaDB [evilcorp]> select * from diagnoses;
+--------------+------------------+-----------+---------+------------+
| id_diagnosis | diagnosis        | id_doctor | id_user | id_disease |
+--------------+------------------+-----------+---------+------------+
|            1 | Esta mu malo     |         1 |       2 |          3 |
|            2 | Esta mas malo aun |         2 |       3 |          1 |
|            3 | Esta malo        |         1 |       2 |          1 |
|            4 | Esta malo maloso |         1 |       1 |          7 |
+--------------+------------------+-----------+---------+------------+
4 rows in set (0.000 sec)

MariaDB [evilcorp]> select * from planet_medicine_production;
+----------+--------------+
| planet   | generated    |
+----------+--------------+
| Mercurio |  15999999.84 |
| Tierra   |  15999999.84 |
| Venus    | 335999987.20 |
+----------+--------------+
3 rows in set (0.001 sec)

MariaDB [evilcorp]>
```

```
DELIMITER $$
CREATE FUNCTION get_city(city_name VARCHAR(64),
country_name VARCHAR(64), planet_name VARCHAR(64))
RETURNS INT UNSIGNED
BEGIN
DECLARE planet_id INT UNSIGNED;
DECLARE country_id INT UNSIGNED;
DECLARE city_id INT UNSIGNED;

SELECT id_planet INTO planet_id FROM planets WHERE name =
planet_name;

IF planet_id IS NULL THEN
INSERT INTO planets (name) VALUES (planet_name);
SET planet_id = LAST_INSERT_ID();
END IF;

SELECT id_country INTO country_id FROM countries WHERE
country = country_name AND id_planet = planet_id;

IF country_id IS NULL THEN
INSERT INTO countries (country, id_planet) VALUES
(country_name, planet_id);
SET country_id = LAST_INSERT_ID();
END IF;

SELECT id_city INTO city_id FROM cities WHERE city =
city_name AND id_country = country_id;
```

```
IF city_id IS NULL THEN
INSERT INTO cities (city, id_country) VALUES (city_name,
country_id);
SET city_id = LAST_INSERT_ID();
ELSE
SET city_id = (SELECT id_city FROM cities WHERE city =
city_name AND id_country = country_id LIMIT 1);
END IF;

RETURN city_id;
END$$
DELIMITER ;
```

Esta función lo que hace es añadir una ciudad a la tabla de cities junto el planeta y el país al que pertenece esta ciudad. Para hacer esto primero añadimos el planeta, luego añadimos el país y luego la ciudad. Lo hacemos así para que no haya problema con las foreign keys. Si el planeta, el país o la ciudad ya existen, se añadirán a dicho planeta y no crearán otro nuevo con el mismo nombre.



```
MariaDB [evilcorp]> select get_city("Buenos Aires", "Panama", "Marte");
+-------------------------------------------+
| get_city("Buenos Aires", "Panama", "Marte") |
+-------------------------------------------+
|                                         9 |
+-------------------------------------------+
1 row in set (0.002 sec)

MariaDB [evilcorp]> select * from cities;
+---------+-------------+------------+
| id_city | city        | id_country |
+---------+-------------+------------+
|       1 | A           |          1 |
|       2 | B           |          2 |
|       3 | C           |          3 |
|       4 | D           |          4 |
|       5 | E           |          5 |
|       6 | F           |          6 |
|       7 | G           |          7 |
|       8 | H           |          8 |
|       9 | Buenos Aires |          9 |
+---------+-------------+------------+
9 rows in set (0.000 sec)

MariaDB [evilcorp]>
```

```
DELIMITER $$

CREATE FUNCTION random_user(planet_id INT)
RETURNS VARCHAR(32)
BEGIN
DECLARE users INT;
DECLARE random_num INT;
DECLARE random_user INT;
DECLARE random_name VARCHAR(32);

SELECT COUNT(*) INTO users FROM users_planets WHERE
users_planets.id_planet = planet_id;
SET random_num = FLOOR(RAND() * users);
SELECT id_user INTO random_user FROM users_planets WHERE
users_planets.id_planet = planet_id LIMIT random_num, 1;
SELECT users.name INTO random_name FROM users WHERE
users.id_user = random_user;
RETURN random_name;
END $$

DELIMITER ;
```

Esta función lo que hace es sacar a un usuario random de un planeta. Para hacer esto primero contamos cuántos usuarios hay alojados en un planeta. Con ese número luego seleccionamos la id de un usuario utilizando la función RAND() poniendo como el número máximo de random el número de gente que hay en el planeta.

```
MariaDB [evilcorp]> select * from users_planets;
+----------------+---------+-----------+
| id_user_planet | id_user | id_planet |
+----------------+---------+-----------+
|              1 |       1 |         1 |
|              2 |       2 |         2 |
|              3 |       3 |         3 |
|              4 |       1 |         3 |
|              5 |       2 |         3 |
|              6 |       3 |         3 |
+----------------+---------+-----------+
6 rows in set (0.000 sec)

MariaDB [evilcorp]> select * from count_users_planet;
+----------+------------+
| planet   | count_user |
+----------+------------+
| Mercurio |          1 |
| Tierra   |          4 |
| Venus    |          1 |
+----------+------------+
3 rows in set (0.001 sec)

MariaDB [evilcorp]> select random_user(3);
+-----------------+
| random_user(3)  |
+-----------------+
| Paco            |
+-----------------+
1 row in set (0.001 sec)

MariaDB [evilcorp]> select random_user(3);
+-----------------+
| random_user(3)  |
+-----------------+
| Manolo          |
+-----------------+
1 row in set (0.000 sec)

MariaDB [evilcorp]> select random_user(3);
+-----------------+
| random_user(3)  |
+-----------------+
| Manolo          |
+-----------------+
1 row in set (0.001 sec)

MariaDB [evilcorp]> select random_user(3);
+-----------------+
| random_user(3)  |
+-----------------+
| Francisco       |
+-----------------+
1 row in set (0.001 sec)

MariaDB [evilcorp]>
```

```
CREATE PROCEDURE killer(IN name_user VARCHAR(64))
BEGIN
DECLARE user_dead VARCHAR(64);
DECLARE user_alive BOOLEAN;

-- Verificar si el usuario está muerto
SELECT dead INTO user_alive
```

```
FROM users
WHERE name = name_user;
IF user_alive THEN

-- Usuario muerto, mostrar mensaje
SELECT CONCAT('El usuario ', name_user, ' ya está muerto.')
AS mensaje;
ELSE

-- Usuario vivo, matarlo
UPDATE users
SET dead = TRUE
WHERE name = name_user;
SELECT CONCAT('Hemos matado al usuario ', name_user, '.')
AS mensaje;
END IF;
END $$

DELIMITER ;
```

Este procedimiento lo que hace es coger el usuario que le pasamos por parámetros y matarlo añadiendo en el campo de dead de users un 1. Para hacer eso lo que hacemos es crear una variable dentro del procedimiento con la que indicar si está vivo o muerto. Haremos un select del campo de dead y si es un 1 (muerto), el valor será true, y si es 0 (vivo), false. Luego con un if comprobaremos si está vivo, si lo está lo mataremos, si está muerto mandaremos un mensaje diciendo que el usuario ya está muerto.

CASO VIVO:

```
MariaDB [evilcorp]> call killer(2);
+----------------------------+
| mensaje                    |
+----------------------------+
| Hemos matado al usuario 2. |
+----------------------------+
1 row in set (0.002 sec)

Query OK, 2 rows affected (0.002 sec)

MariaDB [evilcorp]> select * from users;
+---------+-----------+------+
| id_user | name      | dead |
+---------+-----------+------+
|       1 | Paco      |    0 |
|       2 | Manolo    |    0 |
|       3 | Francisco |    0 |
|       4 | Sergio    |    0 |
|       5 | Paca      |    0 |
|       6 | Manola    |    0 |
|       7 | Francisca |    0 |
|       8 | Sergia    |    0 |
|       9 | 1         |    0 |
|      10 | 2         |    1 |
|      11 | 3         |    0 |
|      12 | 4         |    0 |
|      13 | 5         |    0 |
|      14 | 6         |    0 |
|      15 | 7         |    0 |
|      16 | 8         |    0 |
+---------+-----------+------+
16 rows in set (0.000 sec)

MariaDB [evilcorp]>
```

CASO MUERTO:

```
MariaDB [evilcorp]> call killer(2);
+--------------------------------+
| mensaje                        |
+--------------------------------+
| El usuario 2 ya está muerto.   |
+--------------------------------+
1 row in set (0.001 sec)

Query OK, 1 row affected (0.001 sec)

MariaDB [evilcorp]> select * from users;
+---------+-----------+------+
| id_user | name      | dead |
+---------+-----------+------+
|       1 | Paco      |    0 |
|       2 | Manolo    |    0 |
|       3 | Francisco |    0 |
|       4 | Sergio    |    0 |
|       5 | Paca      |    0 |
|       6 | Manola    |    0 |
|       7 | Francisca |    0 |
|       8 | Sergia    |    0 |
|       9 | 1         |    0 |
|      10 | 2         |    1 |
|      11 | 3         |    0 |
|      12 | 4         |    0 |
|      13 | 5         |    0 |
|      14 | 6         |    0 |
|      15 | 7         |    0 |
|      16 | 8         |    0 |
+---------+-----------+------+
16 rows in set (0.000 sec)
```

5-PARCA

Primero de todo, para hacer el usuario de la parca, ha de existir el concepto de la muerte. Para eso, lo que haremos será en la función donde asesinaremos a sangre fría a una persona, añadiremos el update del campo dead. Concretamente en esta parte:

```
-- Usuario vivo, matarlo
UPDATE users
SET dead = TRUE
WHERE name = name_user;
SELECT CONCAT('Hemos matado al usuario ', name_user, '.')
AS mensaje;
END IF;
END $$
```

Después de haber creado la muerte, crearemos a la parca de esta forma:

```
CREATE USER 'parca'@'localhost' IDENTIFIED BY 'parca';

GRANT SELECT, UPDATE ON evilcorp.users TO
'parca'@'localhost';
```

Lo que hemos hecho ha sido crear el usuario de parca en el localhost con la contraseña de parca. Luego de crearlo lo que hemos hecho ha sido darle los privilegios pedidos en el proyecto. Ahora lo que haremos será probar la función de la parca.

```
root@Ian:/home/enti# mysql -u parca -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 59
Server version: 10.5.18-MariaDB-0+deb11u1 Debian 11

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

```
Database changed
MariaDB [evilcorp]> select random_user(3);
ERROR 1370 (42000): execute command denied to user 'parca'@'localhost' for routine 'evilcorp.random_user'
MariaDB [evilcorp]> call killer("Francisco");
ERROR 1370 (42000): execute command denied to user 'parca'@'localhost' for routine 'evilcorp.killer'
MariaDB [evilcorp]>
```

Como podemos ver, no podemos hacer absolutamente nada con la parca. Esto se debe a que en los permisos no le hemos puesto los permisos para hacer drop, ni los permisos en las tablas necesarias para sacar el usuario random.