Databases Capstone

Software Solutions Cheems

Marín Márquez Jonathan
Ramírez Flores Eslavica Monserrat
Rea Aparicio Angel David
Soriano Bonilla Carlos Ivan
Valenzuela García de León Fernando Rodrigo

Databases - Group 1
August 14, 2021

1. Introduction

This document defines the reach of the System of a Stationery Chain Business, it has established the reasons and needs to implement this system and described the high level requirements, alongside describing the priorities and most important considerations, characteristics and objectives in general.

We had a proposal to make this system using a database with various functions as a way to help any worker who interacts with this database when generating a sale. Having consideration about the providers, products, the selling and the clients that are the principal involved in a sale.

With all the previous notes, our solution proposal is a database system that can achieve every necessary requirement, and also be more efficient in response times and to be ergonomic for the users working with it.

2. Workplan

In this workplan we will describe the different needs that the database proposal will have and how we understood the different requirements, risks and the organization of the people who are working in the different sections of the database system.

2.1. Management

We distributed the team in different teams and tasks to do as it follows:

- ✓ Marín Márquez Jonathan - **Front-End Programmer and Model Designer**

- ✓ Ramírez Flores Eslavica Monserrat - **Model Designer and Documentation Supporter**

- ✓ Rea Aparicio Angel David - **Translator, Tester and Main Supporter**

- ✓ Soriano Bonilla Carlos Ivan - **Back-End Programmer and Logic Checker**

- ✓ Valenzuela García de León Fernando Rodrigo - **Documentation Writer and Project Manager**

2.2. Reach

The Stationery System will be a web application that works online. It will allow to the administration to consult information attached to the following processes:

- ✓ Registration of the providers with business name, address, name and phone number.

- ✓ Registration of the clients with RFC, name, address and at least one email address.

- ✓ Manage an inventory of different products with bar-codes on them, price at purchase, date of purchase and number in stock.

- ✓ Data of the products such as brand, description and price of gifts, stationery items, prints and recharge, if and only if its corresponding record is in the inventory.

- ✓ Data of the sale such as number, date, sum of the sale, as well price and quantity of each item.

The control system of the Stationery Chain will allow an easy access to the information about the programmed services of the services and dynamic update of itself.

It will also allow the registration of possible clients through a computing system with Ethernet connection.

2.3. System Context

This document contains information about the characteristics of the software, user interfaces, system interfaces, other user needs, description of the functional and non-functional requirements and system characteristics.

2.4. Product Perspective

The system will be designed to work online which will allow a decentralized use, which means it is an independent system who will not interact with any other systems.

2.5. Restrictions

- ✓ The administrative task of a Stationery Chain must work on the personal computers owned by the business.
- ✓ Due to the price of specialized software, free software has to be used.
- ✓ It must contain Web access for the stakeholders.

2.6. Political Risks

The app shall be developed through free licence software with no payment of a Web server, Database Management System or Programming language, therefore, the use of these software will be attached to the established licensed politics.

2.7. Technology Risks

- ✓ The specs of the owned PC may not be enough for the implementation of the system.
- ✓ The integration of the database may pose connection issues.
- ✓ The use of free software may pose a lack of support.
- ✓ Some internet browsers may not have correct access with some web inquiries.
- ✓ It is impossible to try the system in every internet browser.

2.8. Resources Risks

The use of free software may pose a lack of support.

2.9. Ability Risks

The lack of skill while solving a payment problem using the online service, may cause an inconvenience.

2.10. Requirement Risks

The specs of the owned PC may not be enough for the implementation of the system.

2.11. Minutes

The minutes are documents of every reunion the team had, why the team is making the reunion and the different tasks that each member of the team have to do, so we can advance at a satisfactory rate in the development of the capstone.

2.11.1. July 13

We organized the work in different teams, know what each member knows about the database systems and also the different aptitudes each member has, so we could make a good teamwork and set the different online services we are going to need in order to work better and fast.

2.11.2. July 15

We analyzed the initial problem, in order to start de Entity-Relation Model, so we can make the diagram that represents the information of the problem correctly.

2.11.3. July 17

We verified the diagram of the Entity-Relation Model and also we verified the Relational Model that needs a correct ERM to work as it should be. We also talked about every question we had to do to the teacher in the next reunion.

2.11.4. July 18

We talked to the teacher and showed to him all the work so far, so he could give us some feedback, after the teacher logged out of the session, we make all the corrections he pointed out to us, so we could work on the Relational Model with a fixed ERM.

2.11.5. July 20

We checked the ERM and the EM, after we agreed it was fine, we divided us in two teams, one team will be focused on the documentation making, and the other team will be working in the programming database system.

2.11.6. July 26

We had another session with the teacher, he have us some last feedback of the ERM and fixed the RM, we asked him all the doubts that we had about the programming section, such as how far we need to go with the graphic and web parts, also we asked some questions about the documentation, after we got the answers the teacher logged off and we continue with our previous tasks.

2.11.7. July 28

We talked about our progress in the documentation and programming areas, where both teams had a remarkable progress, the next goal of the documentation team is to summarize what the programming team did and the programming team have to start putting the final touches in the DB and start with the web implementation.

2.11.8. August 2

We talked about the web dissing was made and all the characteristics it has, also we checked an application with similar features so we as a team can decide which option is the best for the implementation of the database system.

2.11.9. August 9

We talked about how was the documentation process and the final part of the implementation section of the project.

3. Design

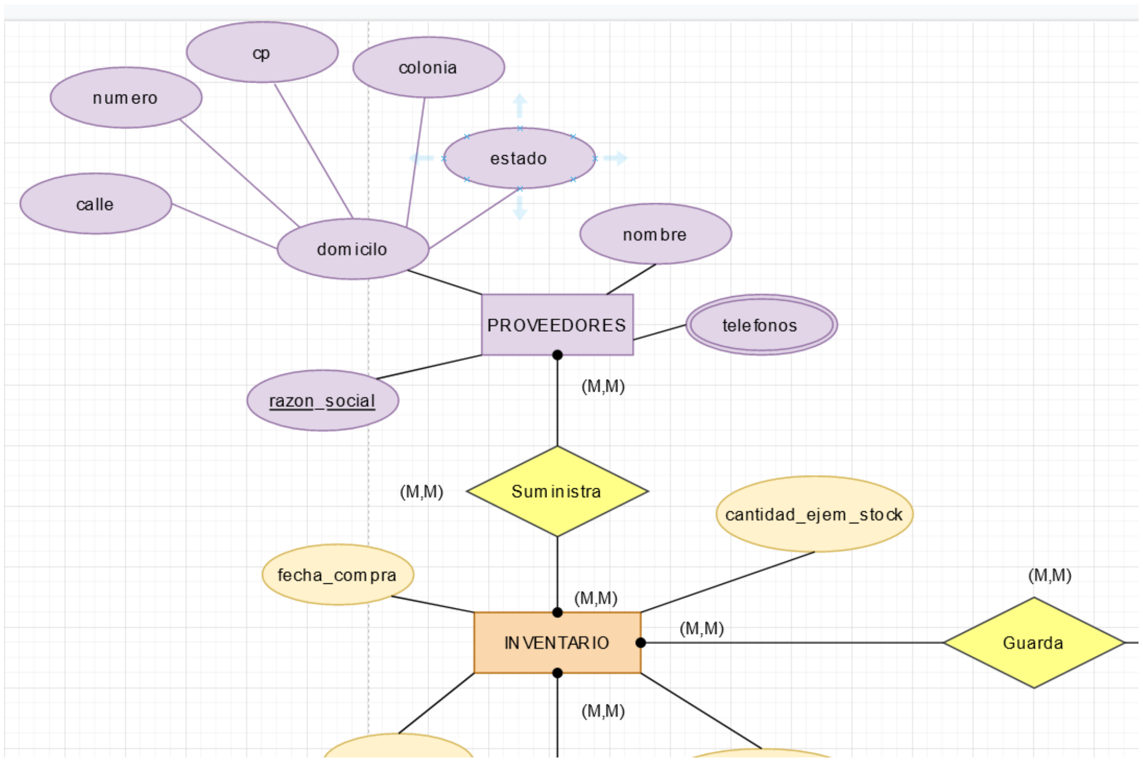
This section contains the design process with the different steps and modifications, in regards to the opinion of each member with the objective of arriving at a satisfactory design based on the requirements.

3.1. First ERM Design

This first EMR design has five entities named *PROVEEDORES*, *INVENTARIO*, *PRODUCTO*, *VENTA* and *CLIENTES*.

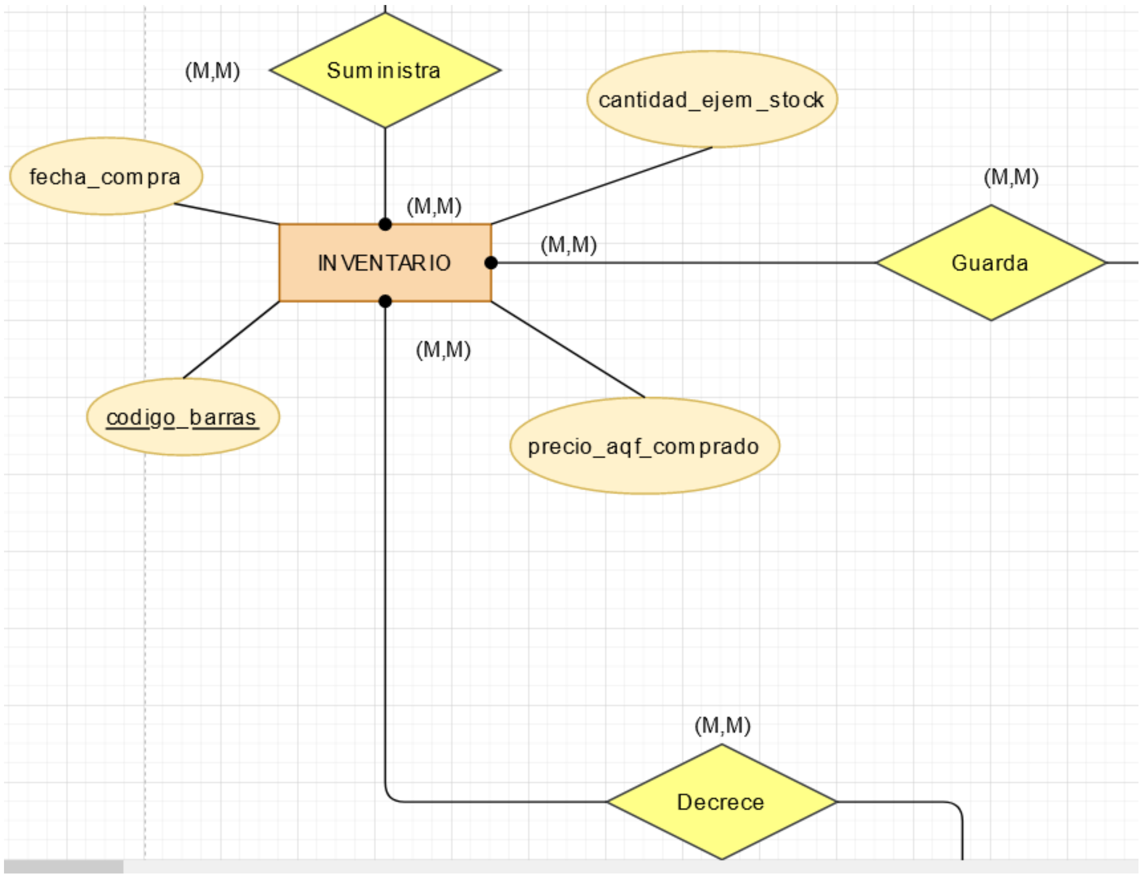
The *PROVEEDORES* entity has *direccion* as a composite attribute by *calle, numero, cp, colonia* and *estado*; it also has a nombre attribute, a *razon_social* attribute as primary key and *telefono* as a multi-valued attribute. The name *PROVEEDORES*, in plural, is a mistake.

Figura 1. Entity-Relation Model - 1, PROVEEDORES entity



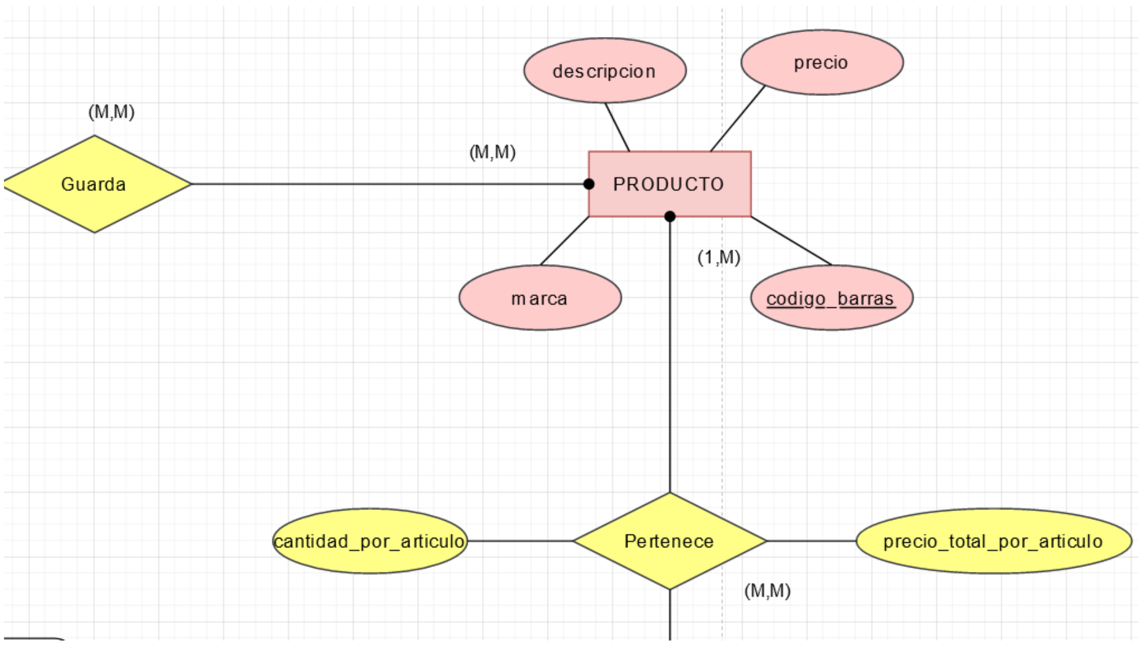
The entity *INVENTARIO* has the attributes *fecha_compra*, *cantidad_ejem_stock*, *precio_adf_comprado* and *codigo_barras*, the last one is it's primary key.

Figura 2. Entity-Relation Model - 1, INVENTARIO entity



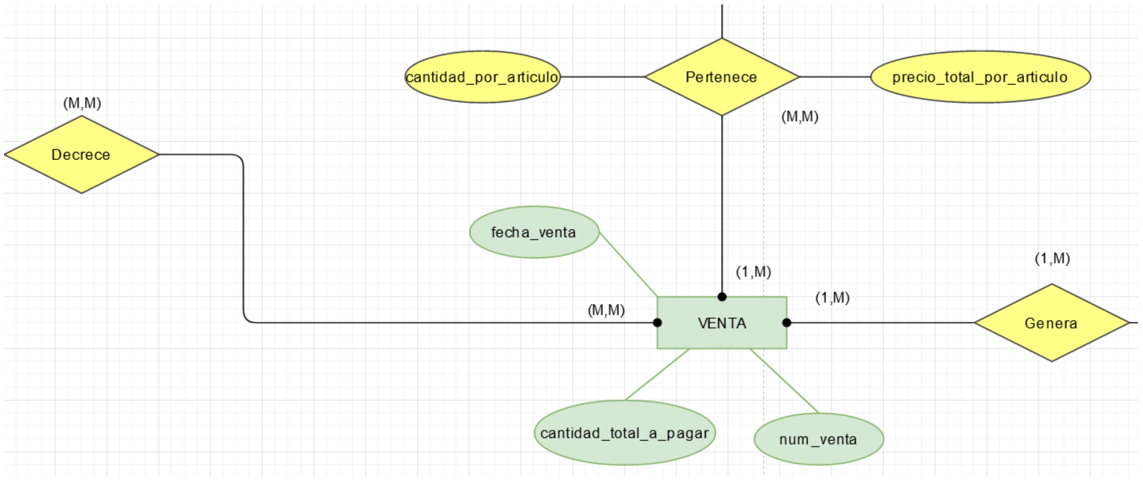
The entity *PRODUCTO* has the attributes *descripcion, precio, marca* and has *codigo_barra* as primary key. With *INVENTARIO* having the same primary key, this represents a problem that was solved in the next EMR design.

Figura 3. Entity-Relation Model - 1, PRODUCTO entity



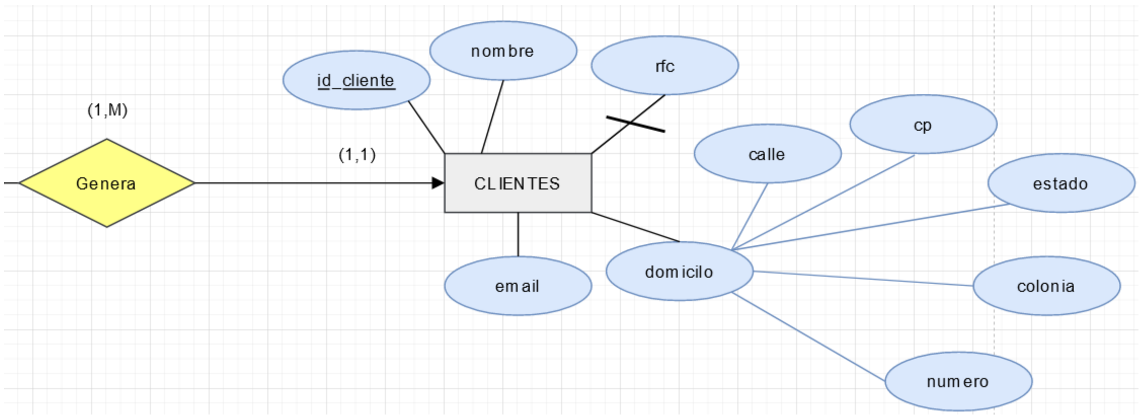
In the entity *VENTA* has the attribute *fecha_Venta, cantitad_total_a_pagar* and *num_venta* without a primary key at the time it was created, we thought to have an artificial attribute, such as *venta_id* but it was never created.

Figura 4. Entity-Relation Model - 1, VENTA entity



Lastly the entity *CLIENTES* has a composite attribute *direccion* the same way as *Proveedores*, it also has email and *nombre* as different attributes. This entity also has *RFC* as attribute and candidate key and *id_cliente* as primary key. This entity also has the problem of *PROVEEDORES*, that means that its name is wrong.

Figura 5. Entity-Relation Model - 1, CLIENTES entity

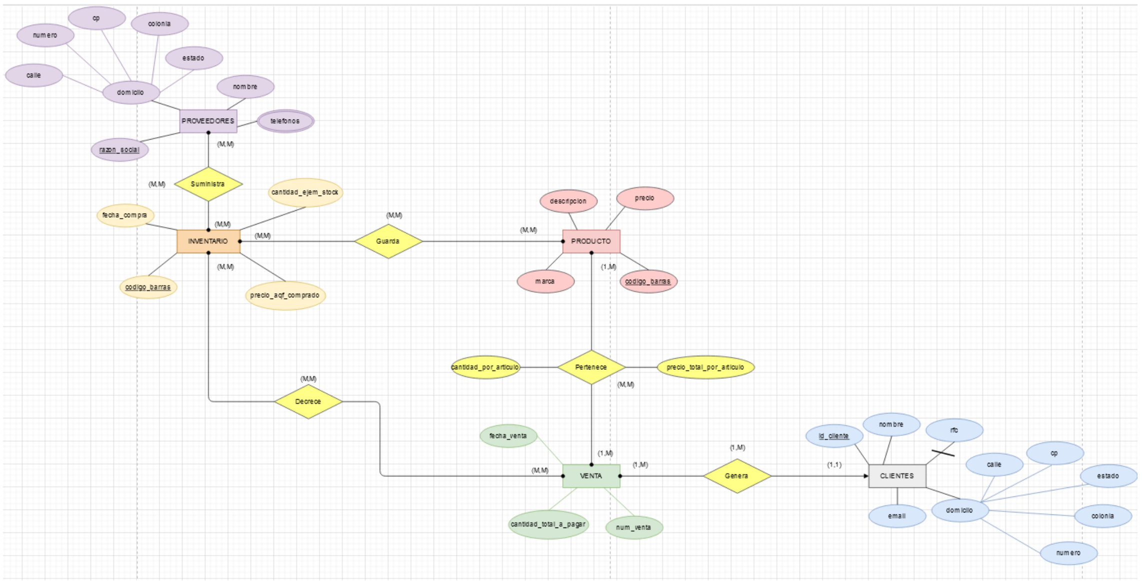


The entities are connected by different relationships:

PROVEDORES and *INVENTORIO* are related by *Suministrar*, *INVENTORIO* and *PRODUCTO* have *Guardar*, *VENTA* and *CLIENTES* are connected by *Genera* as for *PRODUCTO* and *VENTA* their relationship is known by *Pertenece* and has attributes *cantidad_por_articulo* and *precio_total_por_articulo*; We thought to make a relationship between *INVENTARIO* and *VENTA*, known as *Decrementar*, but after a reunion and analyzing the problem, we thought on discard it in the next design.

It is important to mention there are mistakes in the relationships due to the lower limit of interactions between them, it is always "M"but it has to be "1".

Figura 6. Entity-Relation Model - 1



3.2. First RM Desing

For the Relational Model we had to consider the Entity-Relationship Model because this model is the base for the creation of the RM.

Transformation of the Entity Relationship Model to Relational Model:

```
PROVEEDOR:{
razon_social varchar(100) (PK),
nombre varchar (100),
calle_prov varchar(60),
numero_prov smallint,
cp_prov smallint,
colonia_prov varchar(60),
estado_prov varchar(60)
}

TELEFONO:{
 telefono int (PK),
 razon_social varchar (100) (FK)
}
```

The entity is transformed into a table. It also creates another table for the multi-valued attribute *telefonos* where the primary key of the previous table is linked as foreign key in this new table *telefonos*.

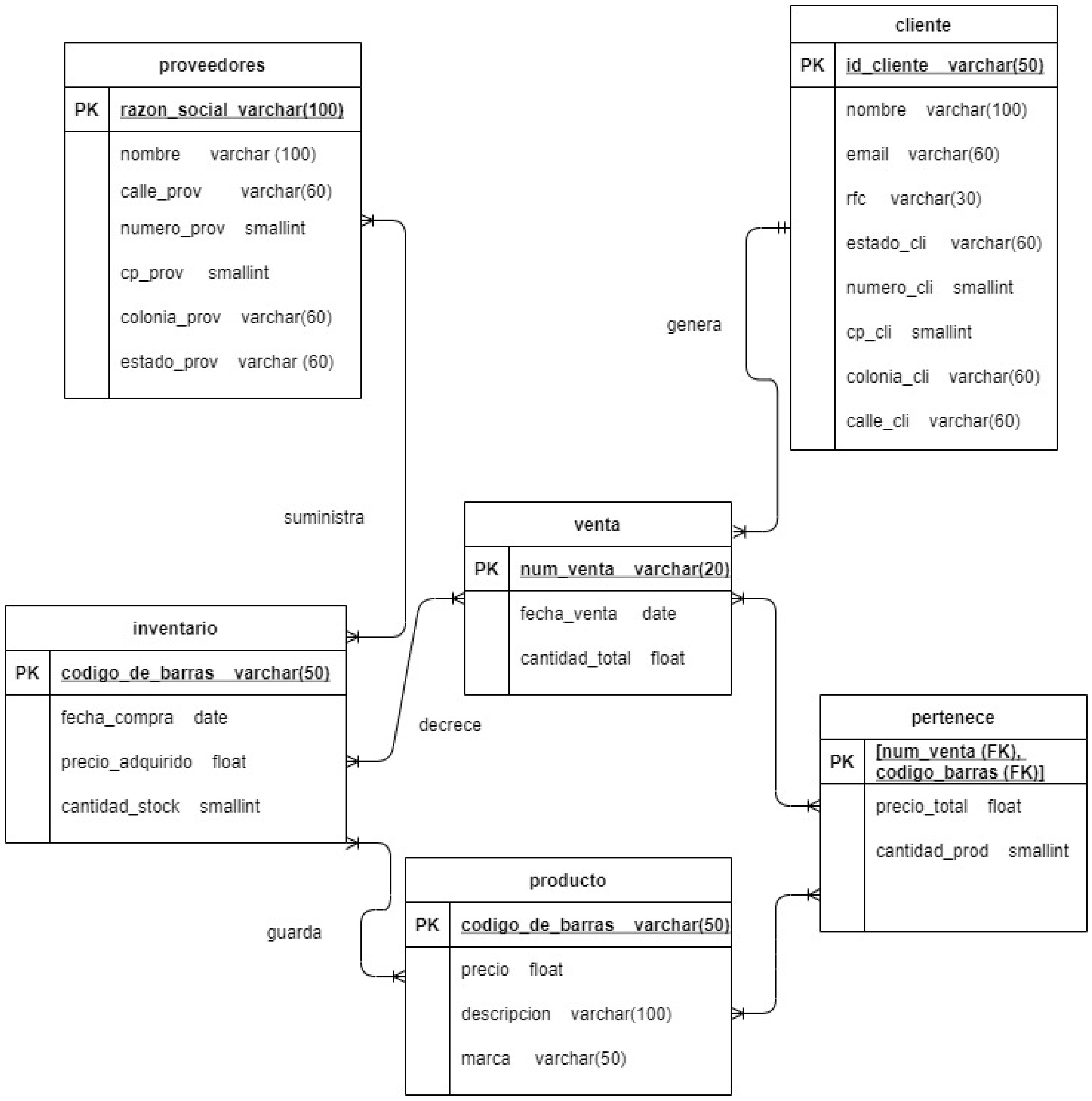
```
SUMINISTRA:{
[razon_social varchar (100)(FK),
codigo_barras varchar(50)(FK)](PK)
}
```

The transformation on entities relationships forms a composite key with its primary keys coming from both entities.

```
INVENTARIO:{
 codigo_barras varchar(50) (PK),
 fecha_compra date,
 precio_comprado float,
 cantidad_stock smallint
}

PRODUCTO:{
codigo_barras varchar (50)(PK),
marca varchar (50),
precio float,
descripcion varchar (100)
}

PERTENECE:{
[codigo_barras  varchar  (50)  (FK),  num_venta  varchar  (20)
(FK)](PK),
cantidad_articulo smallint,
precio_total_articulo float
}
VENTA:{
num_venta varchar (20) (PK),
fecha_venta date,
cantidad_total float
}

CLIENTE:{
id_cliente varchar(50) (PK),
nombre varchar (100),
rfc varchar(30) (U),
calle_cli varchar(60),
numero_cli smallint,
cp_cli smallint,
colonia_cli varchar(60),
estado_cli varchar(60)
email varchar (60)
}

DECRECE:{
[id_cliente varchar(50) (FK),
codigo_barras varchar(50) (FK)](PK)
}
```

All the mistakes committed on the Entity Relationship Model were carried to the Relational Model, some tables lack relationships with many cardinality as the lower value it can have, and some attributes were renamed so they don't match the ERM.

Figura 7. Relational Model - 1



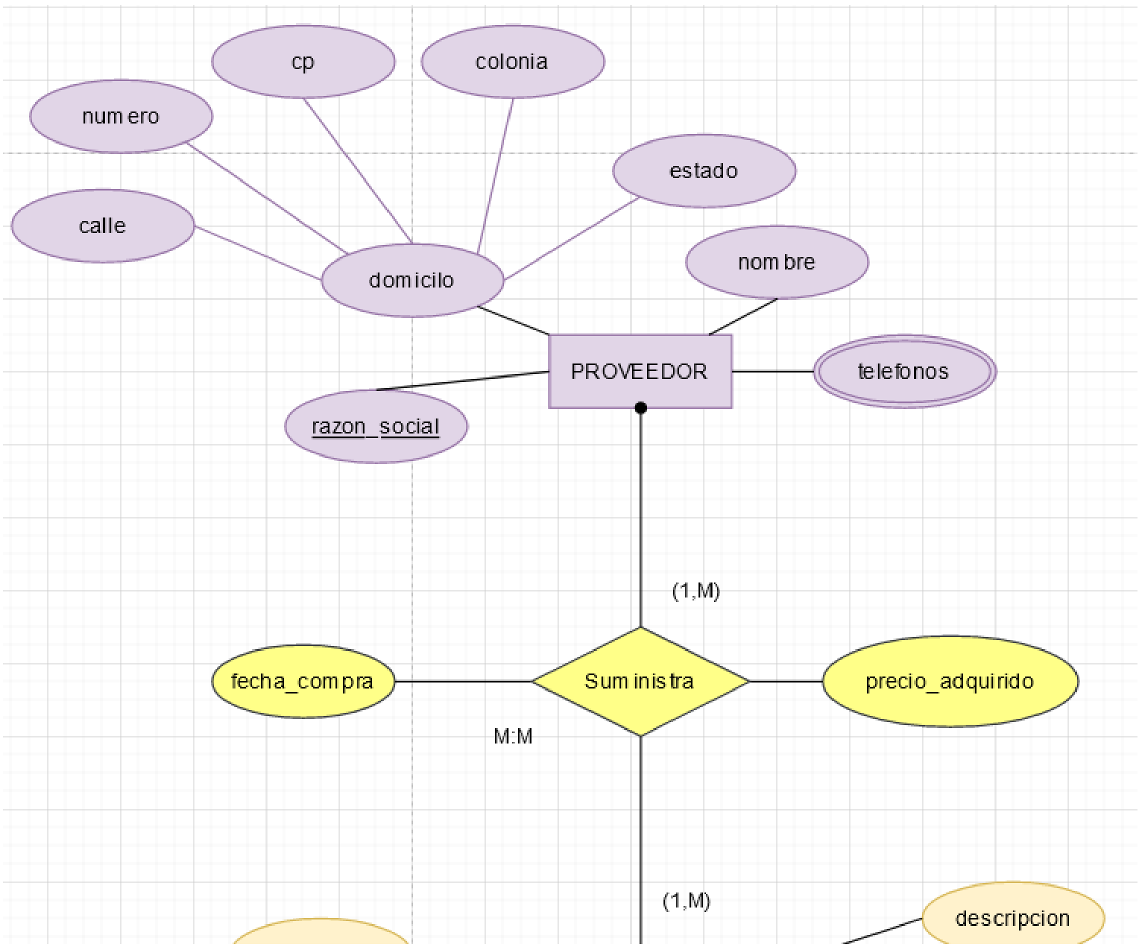
3.3. Second ERM Design

For this design we had a meeting with the teacher in order to correct the previous model.

It contains four entities: *PROVEEDOR, INVENTARIO, VENTA and CLIENTE*; solving the problem that some entities had with their names.

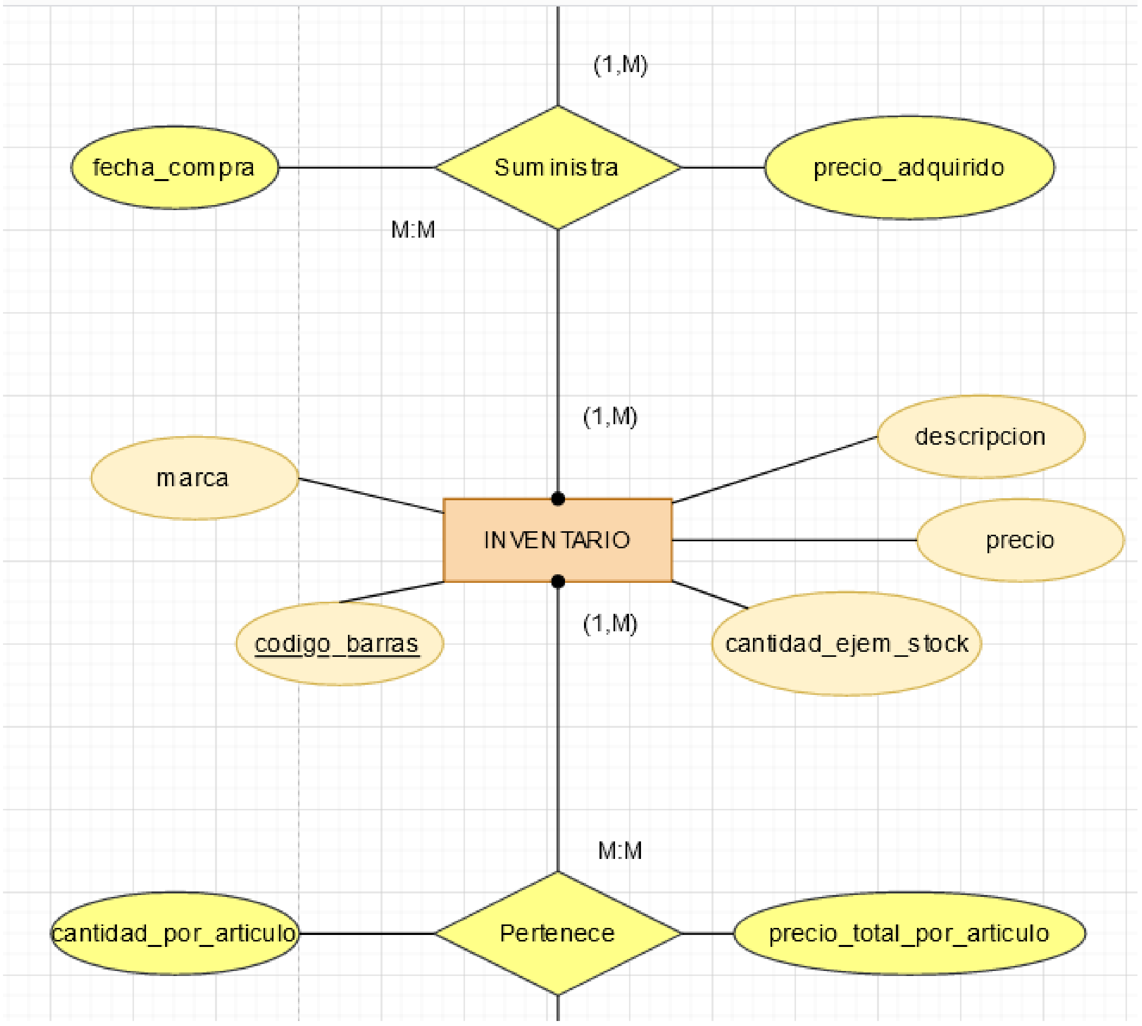
The entity *PROVEEDOR* does not have any change in its attributes: *domicilio, nombre, telefonos* and *razón_social* remain as the primary key.

Figura 8. Entity-Relation Model - 2, PROVEEDOR entity



The entity *INVENTARIO* was combined with the entity producto so bar-code became the only primary key, solving the problem that we had in the previous design, and contains the attributes marca, descripcion, precio and cantidad_ejem_stock.

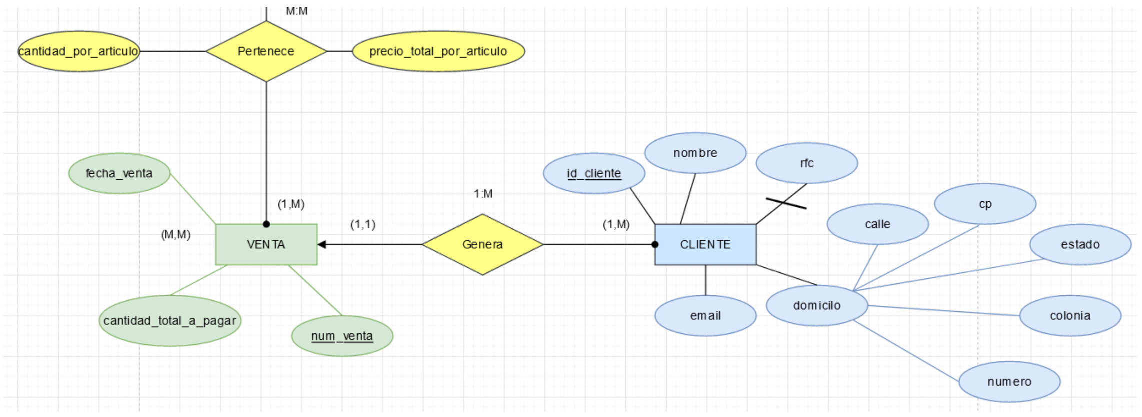
Figura 9. Entity-Relation Model - 2, INVENTARIO entity



The entity *VENTA* changed its primary key, being *num_venta* and maintains its other attributes, these being *fecha_venta* y *cantidad_total_a_pagar*.

The entity *CLIENTE* remains the same, its attributes are: *nombre, domicilio, email, rfc* this one being a candidate key, and *id_cliente* being the primary key.

Figura 10. Entity-Relation Model - 2, VENTA and CLIENTE entity



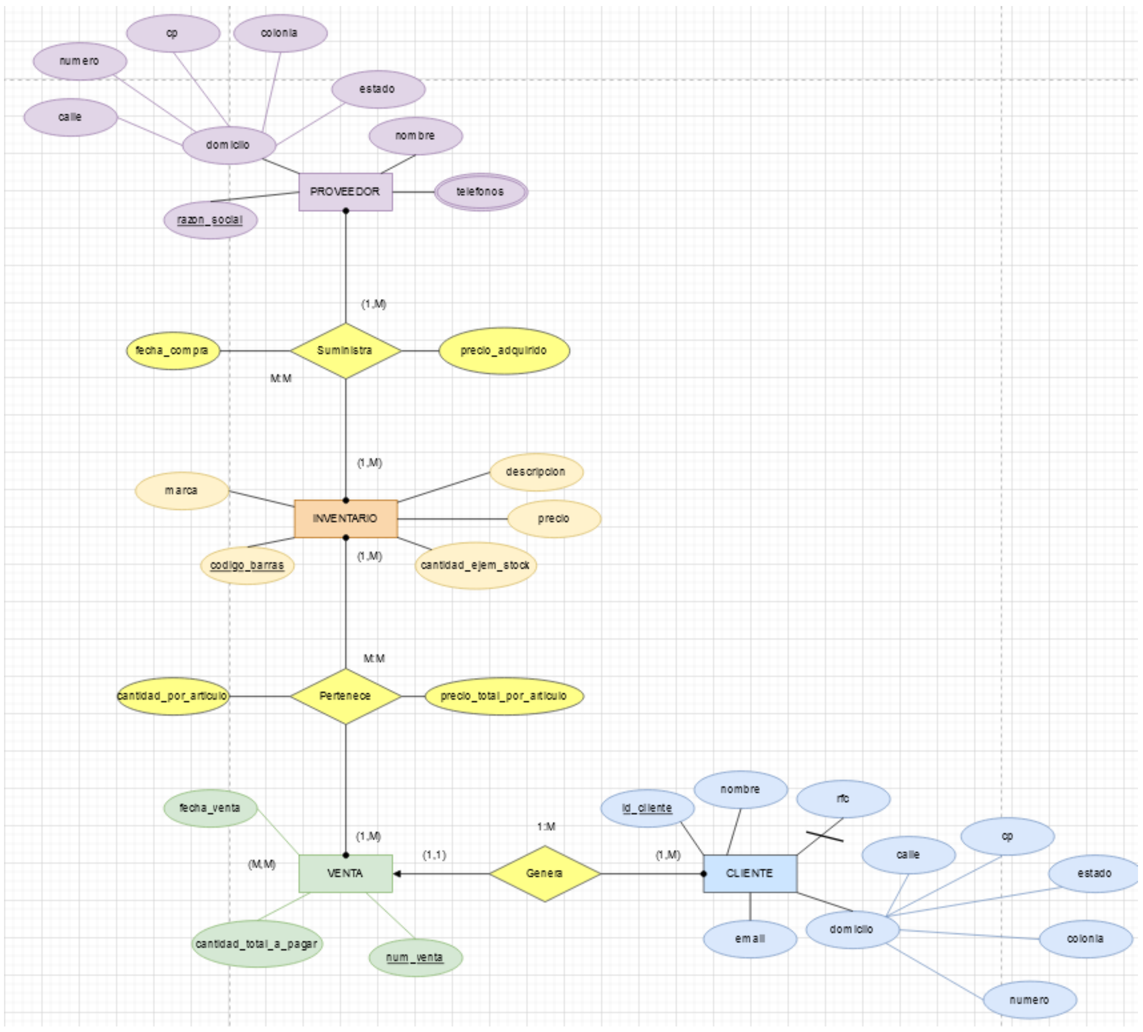
The relationships had various changes, specifically in their cardinality due to the minimum values used, that now they have the value "1" and not "M".

The relationship between *PROVEEDOR* and *INVENTARIO*, is known as *Suministra*, its maximum cardinality is now many to many, and we added two attributes, these being *fecha_compra* and *precio_adquirido*.

The other relationship is between *INVENTARIO* and *VENTA*, known as *Pertenece*, and it also has attributes, these are: *cantidad_por_articulo* and *precio_total_por_articulo*, being a many to many relationship.

Lastly, the relationship of *VENTA* and *CLIENTE*, being *Genera* has a one to many cardinality without any attributes.

Figura 11. Entity-Relation Model - 2



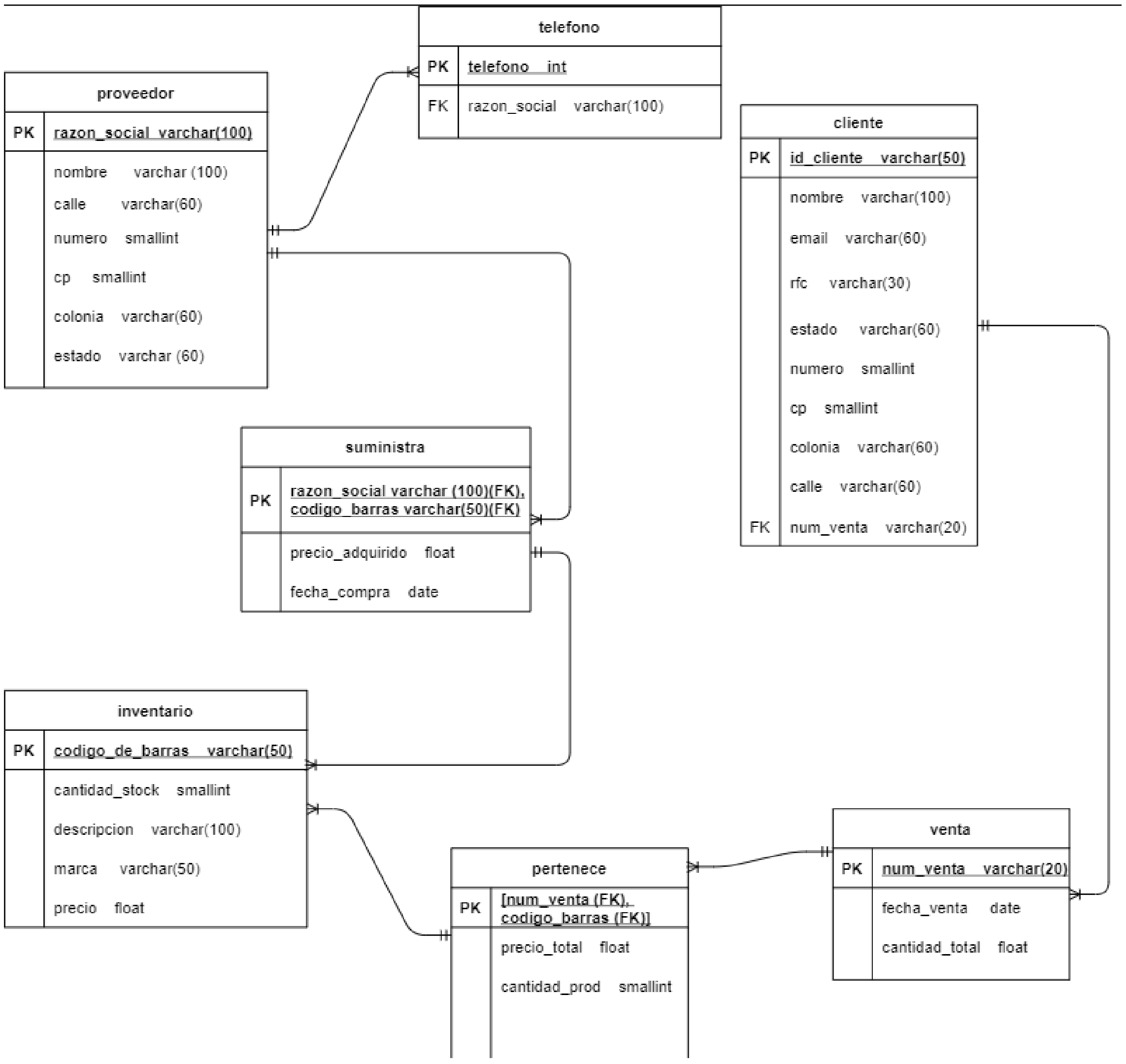
This model have one last problem, the cardinality of the last relationship is incorrect, so in the next design, this must be corrected.

3.4. Second RM Design

This design is a simplified version of the previous EMR with the needed corrections applied, some of these is to make *telefonos* as a sole table because it is a multi-valued attribute, every relationship with their maximum values being 'many to many' have their own table as well, so they can be related to the different entities.

```
PROVEEDOR:{
razon_social varchar(100) (PK),
nombre varchar (100),
calle varchar(60),
numero smallint,
cp smallint,
colonia varchar(60),
estado varchar(60)
}

TELEFONO:{
 telefono int (PK),
 razon_social varchar (100) (FK)
}

SUMINISTRA:{
 [razon_social varchar (100)(FK),
codigo_barras varchar(50)(FK)](PK),
fecha_compra date,
 precio_adquirido float,

}

INVENTARIO:{
 codigo_barras varchar(50) (PK),
 cantidad_stock smallint,
marca varchar (50),
precio float,
descripcion varchar (100)
}

PERTENECE:{
[codigo_barras  varchar  (50)  (FK),  num_venta  varchar  (20)
(FK)](PK),
cantidad_articulo smallint,
precio_total_articulo float
}

VENTA:{
num_venta varchar (20) (PK),
fecha_venta date,
cantidad_total float
}

CLIENTE:{
id_cliente varchar(50) (PK),
nombre varchar (100),
rfc varchar(30) (U),
calle varchar(60),
numero smallint,
cp smallint,
colonia varchar(60),
estado varchar(60),
email varchar (60),
num_venta varchar (20) (FK)
}
```

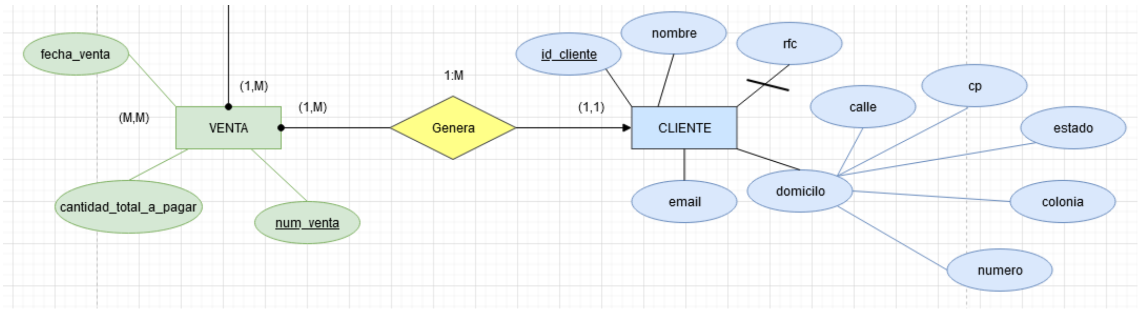
Figura 12. Relational Model - 2



3.5. Final ERM Design

The only difference between the last design and this one, is how the entity *VENTA* and *CLIENTE* are connected, with the relationship *Genera*, the problem was that the arrows in the relationship were inverted. We fixed that and generate the final ERM Design.

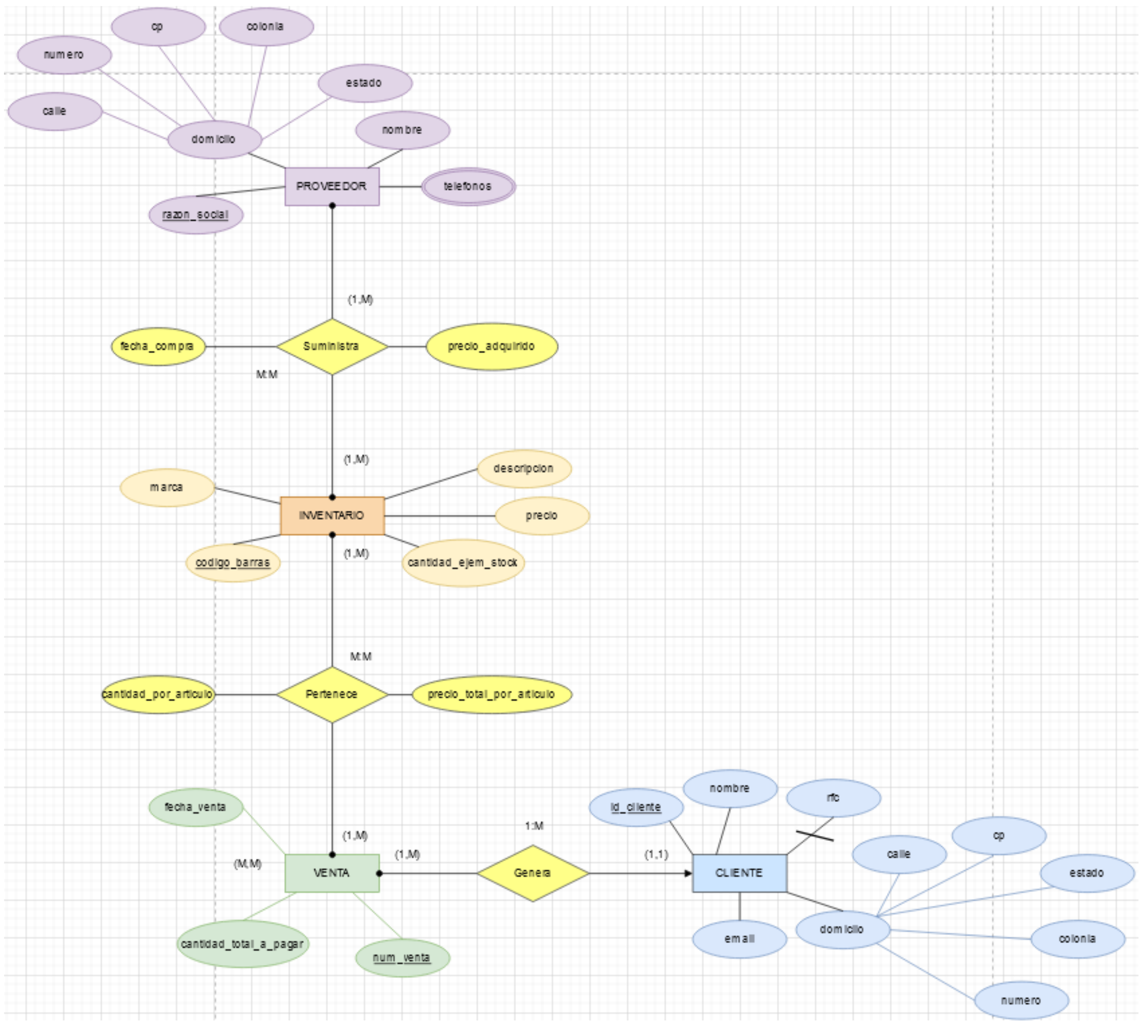
Figura 13. Entity-Relation Model - 3, Relationship Correction



All the other entities, attributes and relationships maintain the same names, properties and characteristics.

This model has been selected for the final release and to start working in the programming area.

Figura 14. Entity-Relation Model - 3



3.6. Final RM Desgign

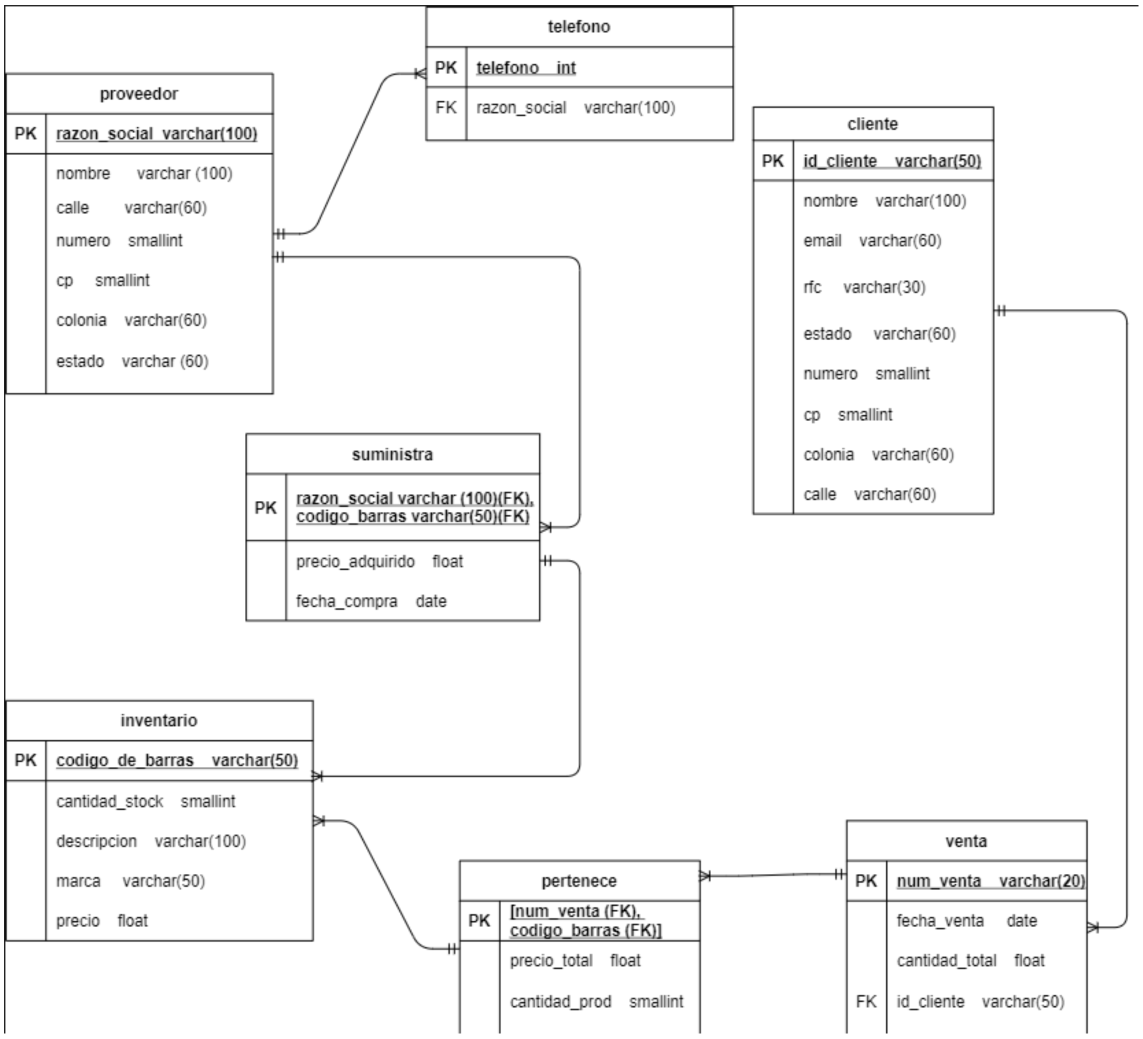
The only thing that was fixed, are the attributes of the table *VENTA* and *CLIENTE*, we changed the place of the Foreign Key from an attribute to the other one.

The correct way to make the rules for the RM of the table *VENTA* and *CLIENTE* are:

```
VENTA:{
num_venta varchar (20) (PK),
fecha_venta date,
cantidad_total float,
id_cliente varchar(50) (FK)
}

CLIENTE:{
```

```
id_cliente varchar(50) (PK),
nombre varchar (100),
rfc varchar(30) (U),
calle varchar(60),
numero smallint,
cp smallint,
colonia varchar(60),
estado varchar(60),
email varchar (60)
}
```

Figura 15. Relational Model - 3



3.7. Normalization Process

We verified if each table of the final RM achieve the three normalization levels, we analyzed each table and reach the third normalization level due to our analysis.

3.7.1. PROVEEDOR

For the *PROVEEDOR* table we can say it achieve the first level because all the columns we have (*razon_social, nombre, calle, numero, cp, colonia* and *estado*) in this table are atomic, this means they only have one data for each time someone enteres data on the table, also it is important to know that *razon_social* is the primary key, we can assure all the information of this table will be unique.

For the second level of normalization, we need to get rid of partial functional dependence between columns, but there is none in our table, we can guarantee that with the next way to see our columns:

$razon\_social = A$
$nombre = B$
$calle = C$
$numero = D$
$cp = E$
$colonia = F$
$estado = G$

```
{A} -> {B, C, D, E, F, G}
```

With the previous analysis we can tell there is not any transitive relations, so we also achieve the third and final level of normalization.

3.7.2. TELEFONO

We achieve the first normalization level because the data will be atomic and the primary key will be *telefono*, and this one can not be owned by two or more *razon_social.*

The second normalization level is achieved because there is no partial functional dependence:

$telefono = A$
$razon\_social = B$

```
{A} - > {B}
```

The third normalization level is achieved because there are not transitive relations, as we separated this table from the *PROVEEDOR* table.

3.7.3. SUMINISTRA

The first normalization level is achieved because all the data are going to be atomic and this time we have a compound primary key, this means we have two primary keys that are *razon_social* and *codigo_barras*, so we can assure the data will be unique too.

The second normalization level is achieved because we do not have partial functional dependence:

razon_social = *A*
codigo_barros = *B*
fecha_comrpa = *C*
precio_adquirido = *D*

```
{A, B} - > {C, D}
```

The third normalization level is achieved because there are not transitive relations.

3.7.4. INVENTARIO

The first normalization level is achieved because all the data will be atomic and unique, the primary key is *codigo_barras*.

The second normalization level is achieved because there are not partial functional dependence:

codigo_barras = *A*
cantidad_stock = *B*
marca = *C*
precio = *D*
descripcion = *E*

```
{A} - > {B, C, D, E}
```

The third normalization level is achieved because there are not transitive relations.

3.7.5. PERTENECE

The first normalization level is achieved because all data will be atomic and unique but we have a compound primary key, the keys are *codigo_barras* and *num_venta*.

The second normalization level is achieved because there are not partial functional dependence:

codigo_barras = *A*

num_venta = *B*
cantidad_articulo = *C*
precio_total_art = *D*

```
{A, B} - > {C, D}
```

The third normalization level is achieved because there are not transitive relations.

3.7.6. VENTA

The first normalization level is achieved because all the data will be atomic and unique, the primary key is *num_venta*.

The second normalization level is achieved because there are not partial functional dependence:

num_ventas = *A*
fecha_venta = *B*
cantidad_total = *C*
id_cliente = *D*

```
{A} - > {B, C}
```

The third normalization level is achieved because there are not transitive relations.

3.7.7. CLIENTE

The first normalization level is achieved because all the data will be atomic and unique, the primary key is *id_cliente*.

The second normalization level is achieved because there are not partial functional dependence:

id_cliente = *A*
rfc = *B*
nombre = *C*
calle = *D*
numero = *E*
cp = *F*
colonia = *G*
estado = *H*
email = *I*

```
{A} - > {B, C, D, E, F, G, H, I}
```

The third normalization level is achieved because there are not transitive relations.

4. Implementation

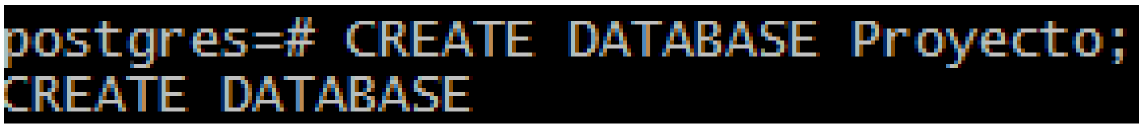
This section will explain how was created the database system, how it works including the stored procedures, triggers and functions needed to the correct implementation of the problem we need to solve.

4.1. DataBase Creation

The first thing we have to do at PostGreSQL is to create the different tables we need, this is the main part of the Database System we are creating.

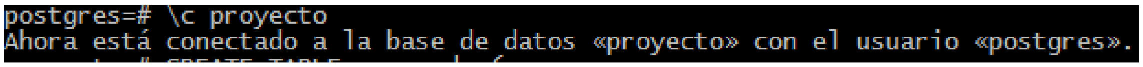
We need to create the Database and enter to it, as it is seen in the next images:

Figura 16. Create Database



```
postgres=# CREATE DATABASE Proyecto;
CREATE DATABASE
```

Figura 17. Enter Database



```
postgres=# \c proyecto
Ahora está conectado a la base de datos «proyecto» con el usuario «postgres».
```

For the tables we used:

`CREATE TABLE`

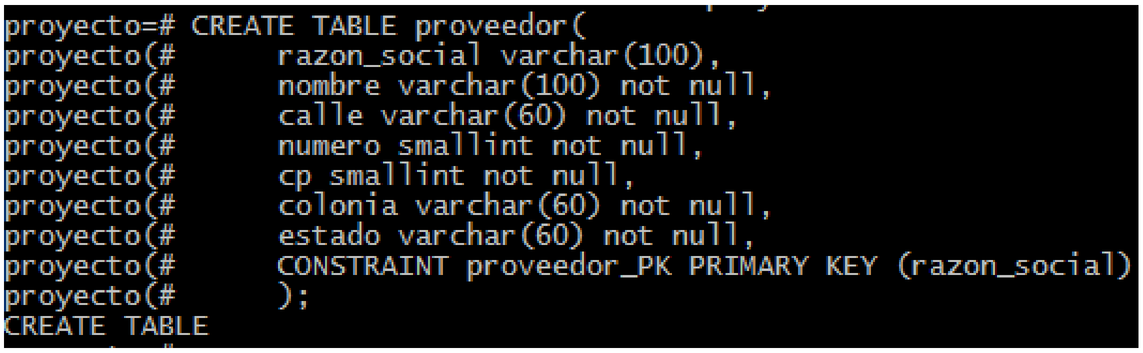
This will help us to create the different tables, after that sentence there must be the name of the table and then between parenthesis there must be all the columns, the type of data they will have and write if it will have null information.

We also add a *CONSTRAINT*, this means that is a conditional that refers to the primary key and also works for foreign keys when they are needed, finally, when a foreign key is needed, we add the *CASCADE* word, this means every time we need to make an update and delete instruction, the father table will update or delete the register on the children table, so it can have a better optimization.

These are the different tables we created from the Relational Model:

```
    CREATE TABLE proveedor(
razon_social varchar(100),
nombre varchar(100) not null,
calle varchar(60) not null,
numero smallint not null,
cp smallint not null,
colonia varchar(60) not null,
estado varchar(60) not null,
CONSTRAINT proveedor_PK PRIMARY KEY (razon_social)
);
```

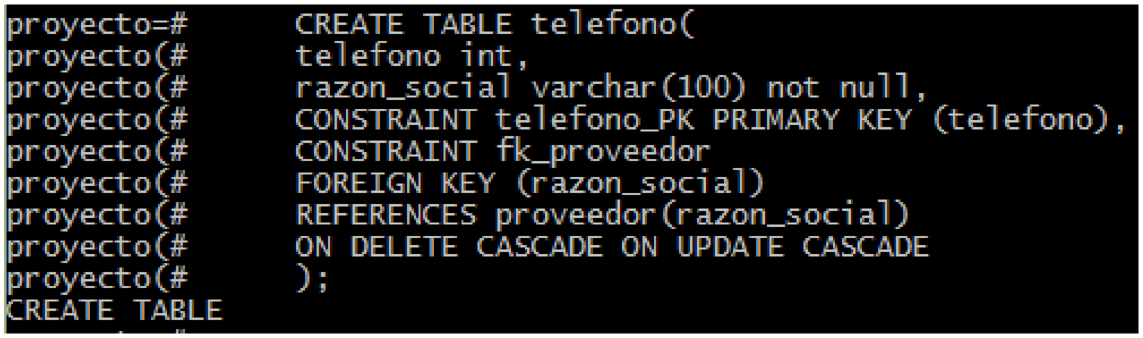
Figura 18. Table Proveedor



```
proyecto=# CREATE TABLE proveedor(
proyecto(#      razon_social varchar(100),
proyecto(#      nombre varchar(100) not null,
proyecto(#      calle varchar(60) not null,
proyecto(#      numero smallint not null,
proyecto(#      cp smallint not null,
proyecto(#      colonia varchar(60) not null,
proyecto(#      estado varchar(60) not null,
proyecto(#      CONSTRAINT proveedor_PK PRIMARY KEY (razon_social)
proyecto(#      );
CREATE TABLE
```

```
    CREATE TABLE telefono(
telefono int,
razon_social varchar(100) not null,
CONSTRAINT telefono_PK PRIMARY KEY (telefono),
CONSTRAINT fk_proveedor
FOREIGN KEY (razon_social)
REFERENCES proveedor(razon_social)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

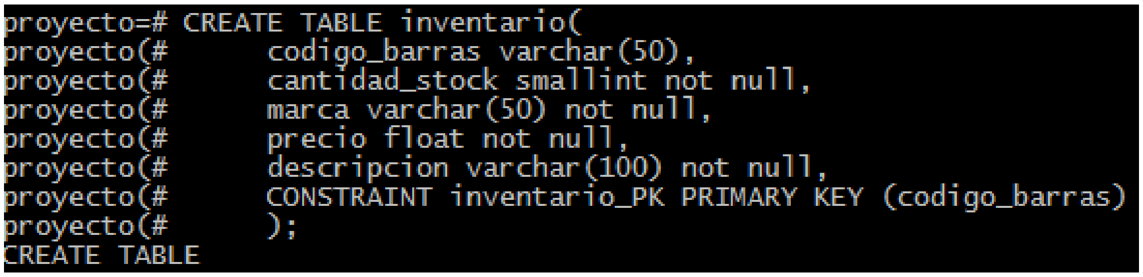
Figura 19. Table Telefono



```
proyecto=#      CREATE TABLE telefono(
proyecto(#      telefono int,
proyecto(#      razon_social varchar(100) not null,
proyecto(#      CONSTRAINT telefono_PK PRIMARY KEY (telefono),
proyecto(#      CONSTRAINT fk_proveedor
proyecto(#      FOREIGN KEY (razon_social)
proyecto(#      REFERENCES proveedor(razon_social)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE
proyecto(#      );
CREATE TABLE
```

```
CREATE TABLE inventario(
codigo_barras varchar(50),
cantidad_stock smallint not null,
marca varchar(50) not null,
precio float not null,
descripcion varchar(100) not null,
CONSTRAINT inventario_PK PRIMARY KEY (codigo_barras)
);
```

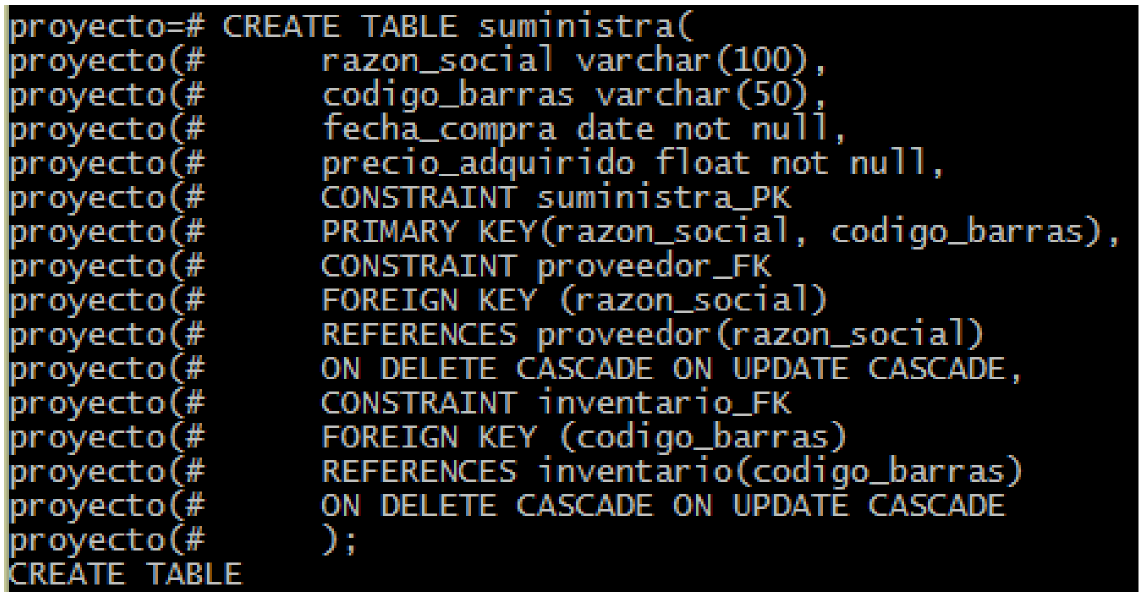
Figura 20. Table Inventario



```
proyecto=# CREATE TABLE inventario(
proyecto(#      codigo_barras varchar(50),
proyecto(#      cantidad_stock smallint not null,
proyecto(#      marca varchar(50) not null,
proyecto(#      precio float not null,
proyecto(#      descripcion varchar(100) not null,
proyecto(#      CONSTRAINT inventario_PK PRIMARY KEY (codigo_barras)
proyecto(#      );
CREATE TABLE
```

```
    CREATE TABLE suministra(
razon_social varchar(100),
codigo_barras varchar(50),
fecha_compra date not null,
precio_adquirido float not null,
CONSTRAINT suministra_PK
PRIMARY KEY(razon_social, codigo_barras),
CONSTRAINT proveedor_FK
FOREIGN KEY (razon_social)
REFERENCES proveedor(razon_social)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT inventario_FK
FOREIGN KEY (codigo_barras)
REFERENCES inventario(codigo_barras)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

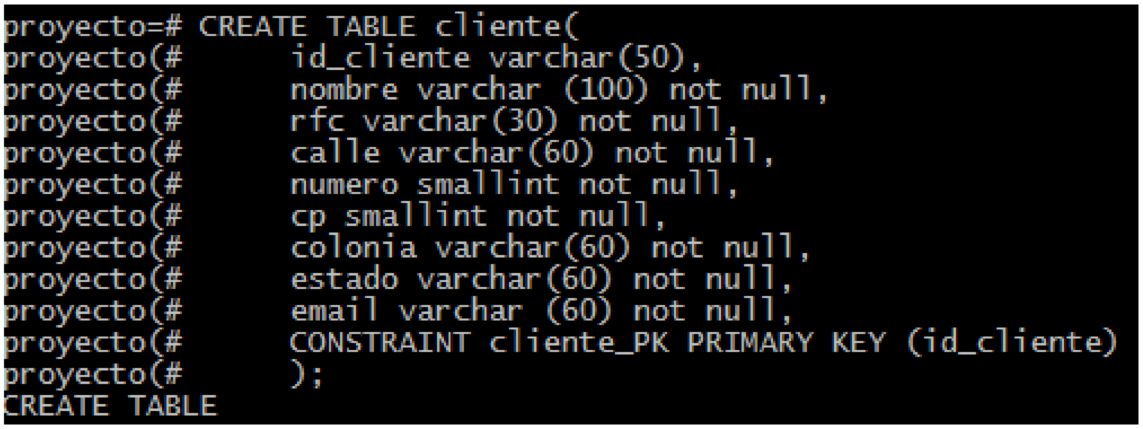
Figura 21. Table Suministra



```
proyecto=# CREATE TABLE suministra(
proyecto(#      razon_social varchar(100),
proyecto(#      codigo_barras varchar(50),
proyecto(#      fecha_compra date not null,
proyecto(#      precio_adquirido float not null,
proyecto(#      CONSTRAINT suministra_PK
proyecto(#      PRIMARY KEY(razon_social, codigo_barras),
proyecto(#      CONSTRAINT proveedor_FK
proyecto(#      FOREIGN KEY (razon_social)
proyecto(#      REFERENCES proveedor(razon_social)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE,
proyecto(#      CONSTRAINT inventario_FK
proyecto(#      FOREIGN KEY (codigo_barras)
proyecto(#      REFERENCES inventario(codigo_barras)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE
proyecto(#      );
CREATE TABLE
```

```
    CREATE TABLE cliente(
id_cliente varchar(50),
nombre varchar (100) not null,
rfc varchar(30) not null,
calle varchar(60) not null,
numero smallint not null,
cp smallint not null,
colonia varchar(60) not null,
estado varchar(60) not null,
email varchar (60) not null,
CONSTRAINT cliente_PK PRIMARY KEY (id_cliente)
);
```

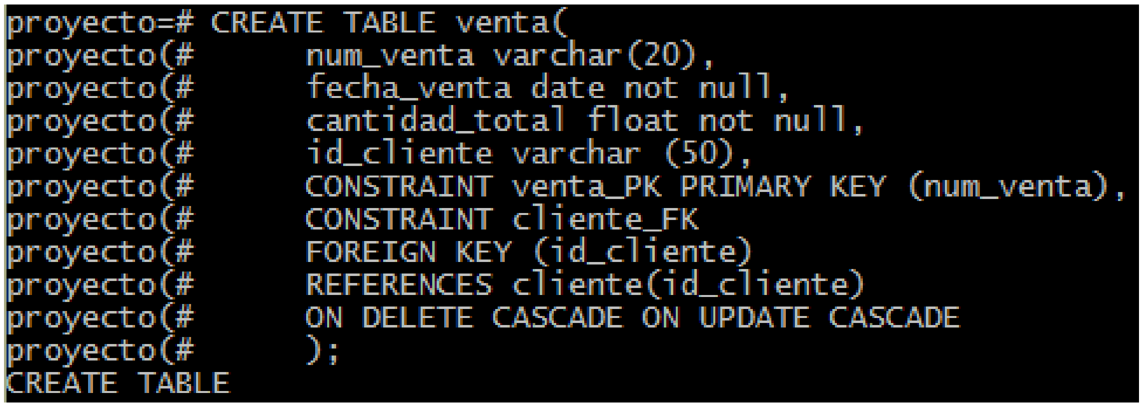
Figura 22. Table Cliente



```
proyecto=# CREATE TABLE cliente(
proyecto(#      id_cliente varchar(50),
proyecto(#      nombre varchar (100) not null,
proyecto(#      rfc varchar(30) not null,
proyecto(#      calle varchar(60) not null,
proyecto(#      numero smallint not null,
proyecto(#      cp smallint not null,
proyecto(#      colonia varchar(60) not null,
proyecto(#      estado varchar(60) not null,
proyecto(#      email varchar (60) not null,
proyecto(#      CONSTRAINT cliente_PK PRIMARY KEY (id_cliente)
proyecto(#      );
CREATE TABLE
```

```
    CREATE TABLE venta(
num_venta varchar(20),
fecha_venta date not null,
cantidad_total float not null,
id_cliente varchar (50),
CONSTRAINT venta_PK PRIMARY KEY (num_venta),
CONSTRAINT cliente_FK
FOREIGN KEY (id_cliente)
REFERENCES cliente(id_cliente)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

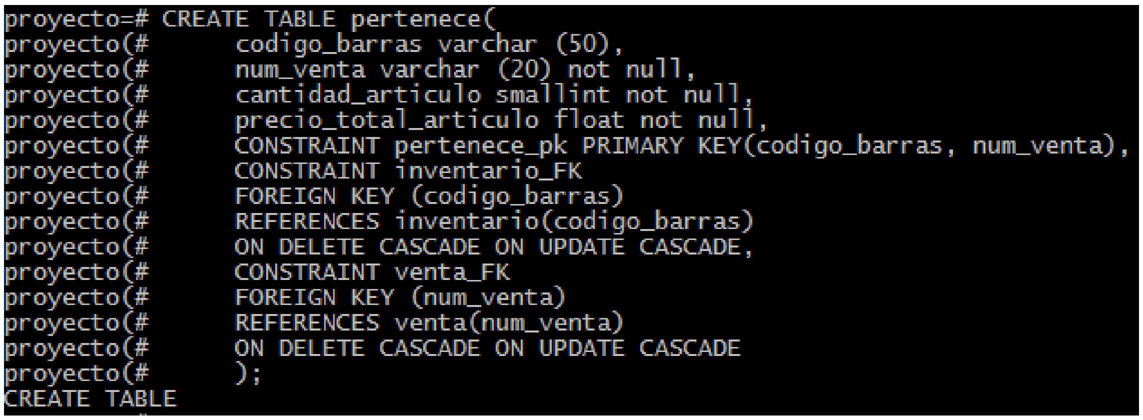
Figura 23. Table venta



```
proyecto=# CREATE TABLE venta(
proyecto(#      num_venta varchar(20),
proyecto(#      fecha_venta date not null,
proyecto(#      cantidad_total float not null,
proyecto(#      id_cliente varchar (50),
proyecto(#      CONSTRAINT venta_PK PRIMARY KEY (num_venta),
proyecto(#      CONSTRAINT cliente_FK
proyecto(#      FOREIGN KEY (id_cliente)
proyecto(#      REFERENCES cliente(id_cliente)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE
proyecto(#      );
CREATE TABLE
```

```
    CREATE TABLE pertenece(
codigo_barras varchar (50),
num_venta varchar (20) not null,
cantidad_articulo smallint not null,
precio_total_articulo float not null,
CONSTRAINT pertenece_pk PRIMARY KEY(codigo_barras, num_venta),
CONSTRAINT inventario_FK
FOREIGN KEY (codigo_barras)
REFERENCES inventario(codigo_barras)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT venta_FK
FOREIGN KEY (num_venta)
REFERENCES venta(num_venta)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

Figura 24. Table Pertenece



```
proyecto=# CREATE TABLE pertenece(
proyecto(#      codigo_barras varchar (50),
proyecto(#      num_venta varchar (20) not null,
proyecto(#      cantidad_articulo smallint not null,
proyecto(#      precio_total_articulo float not null,
proyecto(#      CONSTRAINT pertenece_pk PRIMARY KEY(codigo_barras, num_venta),
proyecto(#      CONSTRAINT inventario_FK
proyecto(#      FOREIGN KEY (codigo_barras)
proyecto(#      REFERENCES inventario(codigo_barras)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE,
proyecto(#      CONSTRAINT venta_FK
proyecto(#      FOREIGN KEY (num_venta)
proyecto(#      REFERENCES venta(num_venta)
proyecto(#      ON DELETE CASCADE ON UPDATE CASCADE
proyecto(#      );
CREATE TABLE
```

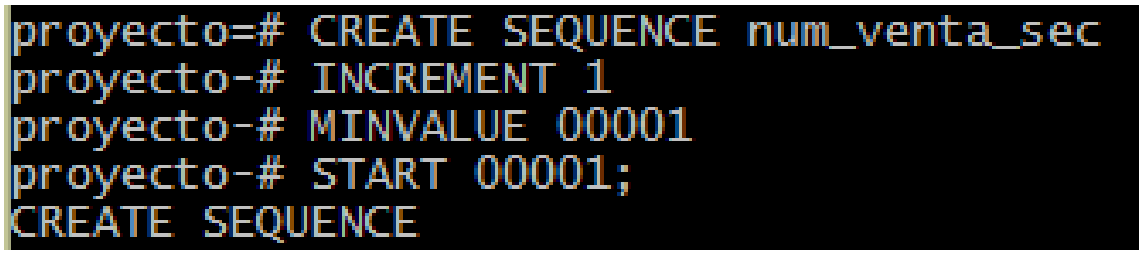
4.2. Sequence

We need to create sequences in order to count the different actions our database system have to record when we need this special information. The way we will create this sequences are via *CREATE SEQUENCE*, that will create the sequence, then we have to write the name of the sequence and use other reserved words to tell what we want that sequence to do.

The first sequence is about counting the number of every sale, it is called *num_venta*, it will increment every time a sale is being made, there must be a min value, in code it is:

```
CREATE SEQUENCE num_venta_sec
INCREMENT 1
MINVALUE 00001
START 00001;
```

Figura 25. Sequence num_venta_sec



```
proyecto=# CREATE SEQUENCE num_venta_sec
proyecto-# INCREMENT 1
proyecto-# MINVALUE 00001
proyecto-# START 00001;
CREATE SEQUENCE
```

4.3. Insertion Triggers and Messages Functions

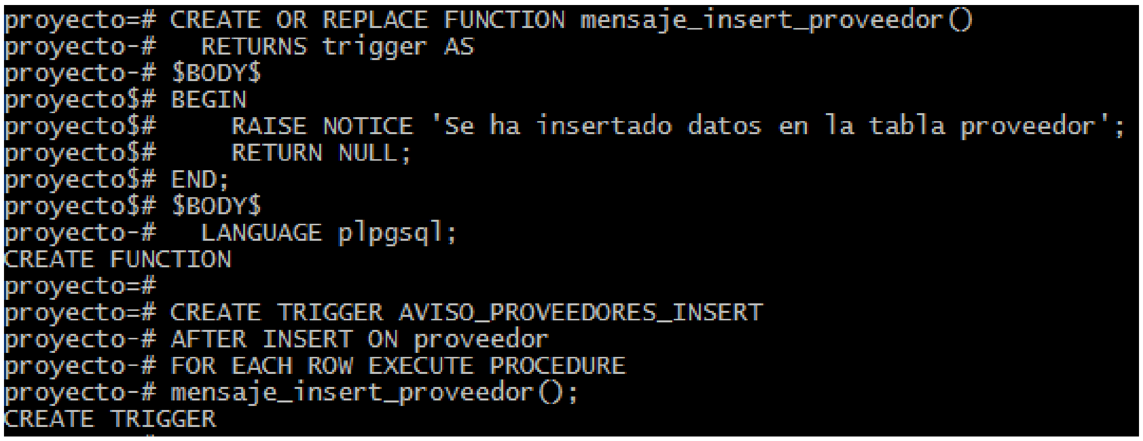
The first thing we did at this part is to create a function that will notify the one who is managing the Database System about a insertion in the system, this is made with the words *CREATE OR REPLACE FUNCTION* to start the function. Because we need to make a notification system, we need *RAISE NOTICE* so the system can send the message written after that sentence to work, finally is important to mention that we have to write *RETURN NULL* because it is a function, so it has to return a value.

After creating the function, we have to send to it a trigger, the trigger will work with *CREATE TRIGGER*, then after that sentence we write the name of the trigger, after creating the trigger, we have to write *INSERT AFTER INSERT ON* so we can know which table will be affected by the trigger, and finally write *FOR EACH ROW EXECUTE PROCEDURE* as it says, it means that every time we insert information on the table we wrote, the system will call a procedure, the one that will be called is the one created previous of the trigger, so we need to write the name of the function.

We can see how we created the different triggers for insertion as it follows:

```
CREATE OR REPLACE FUNCTION mensaje_insert_proveedor()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla proveedor';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PROVEEDORES_INSERT
AFTER INSERT ON proveedor
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_proveedor();
```

Figura 26. Insert Trigger Proveedor

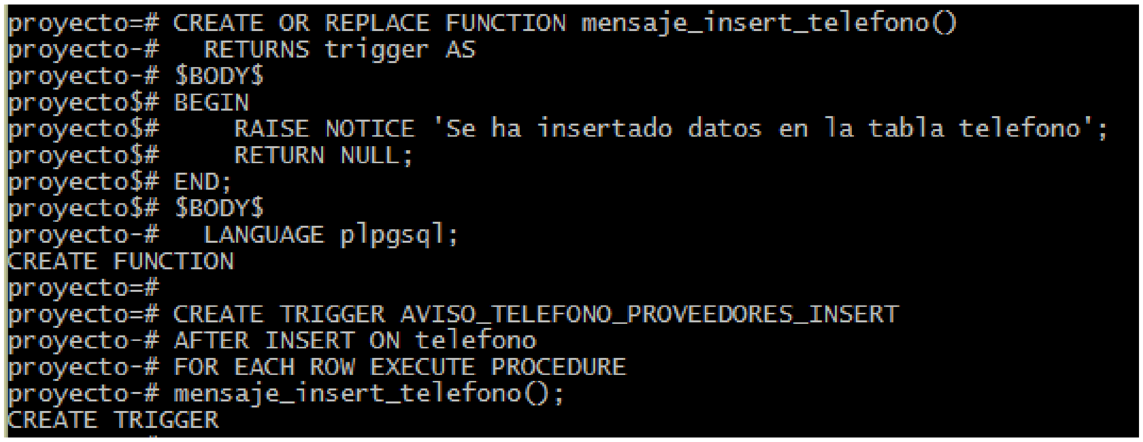


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_proveedor()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla proveedor';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PROVEEDORES_INSERT
proyecto-# AFTER INSERT ON proveedor
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_proveedor();
CREATE TRIGGER
```

```
    CREATE OR REPLACE FUNCTION mensaje_insert_telefono()
  RETURNS trigger AS
```

```
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla telefono';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_INSERT
AFTER INSERT ON telefono
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_telefono();
```

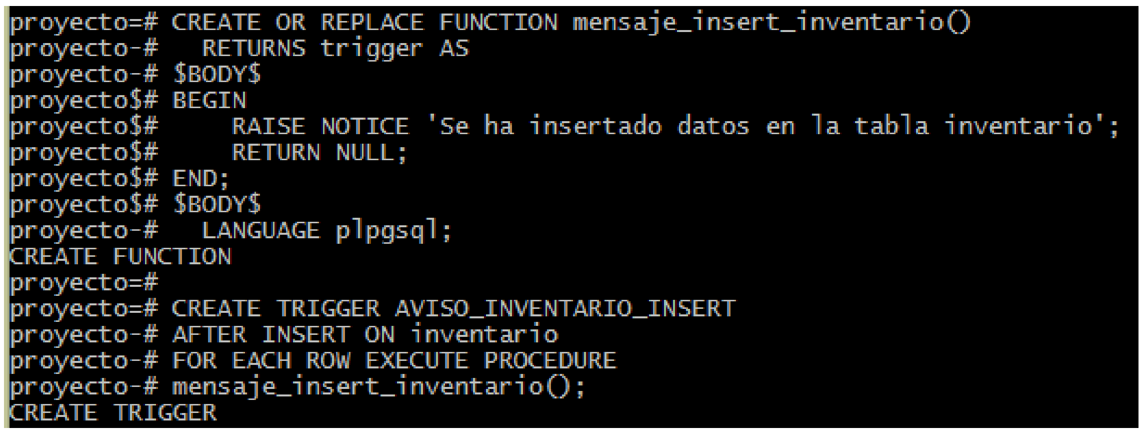
Figura 27. Insert Trigger Telefono



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_telefono()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla telefono';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_INSERT
proyecto-# AFTER INSERT ON telefono
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_telefono();
CREATE TRIGGER
```

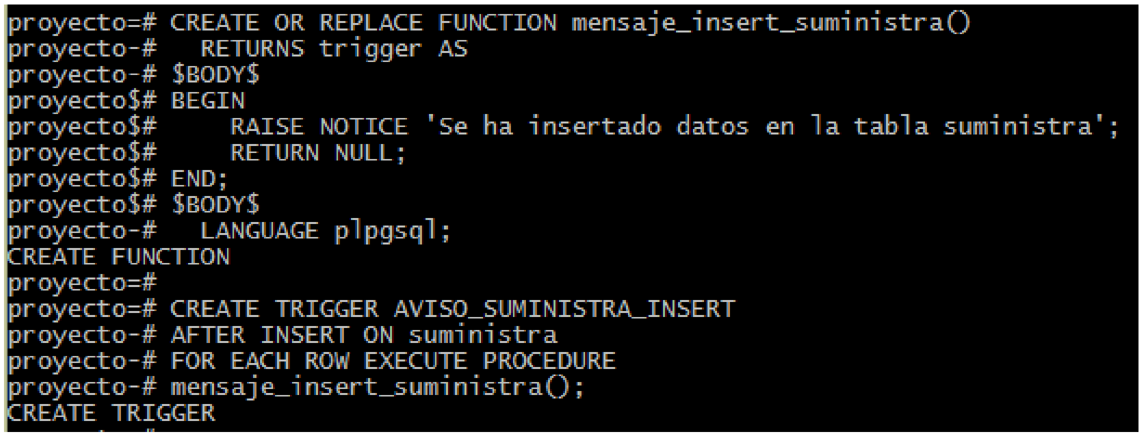
```
    CREATE OR REPLACE FUNCTION mensaje_insert_inventario()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla inventario';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_INVENTARIO_INSERT
AFTER INSERT ON inventario
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_inventario();
```

Figura 28. Insert Trigger Inventario


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_inventario()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla inventario';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_INVENTARIO_INSERT
proyecto-# AFTER INSERT ON inventario
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_inventario();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_insert_suministra()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE ’Se ha insertado datos en la tabla suministra’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_SUMINISTRA_INSERT
AFTER INSERT ON suministra
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_suministra();
```

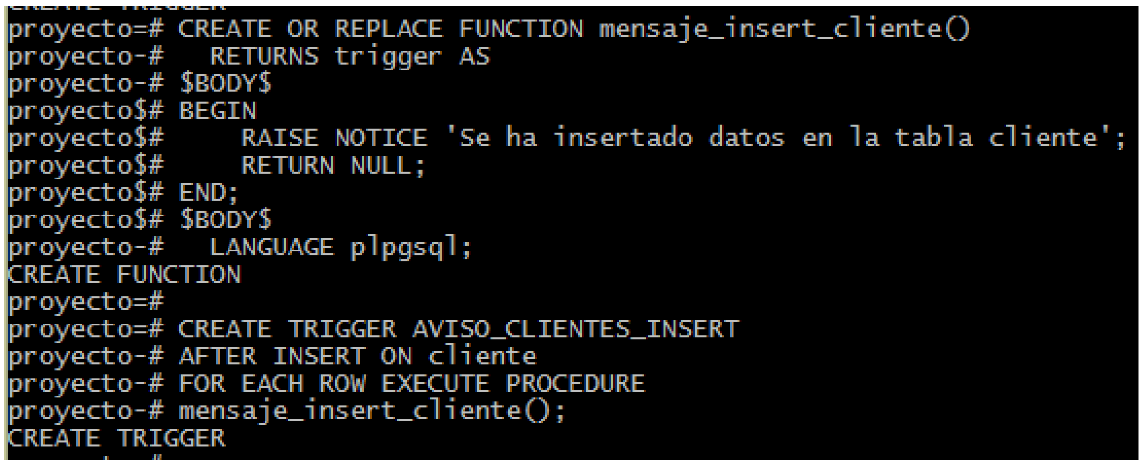
Figura 29. Insert Trigger Suministra


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_suministra()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla suministra';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_SUMINISTRA_INSERT
proyecto-# AFTER INSERT ON suministra
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_suministra();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_insert_cliente()
```

```
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla cliente';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_CLIENTES_INSERT
AFTER INSERT ON cliente
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_cliente();
```

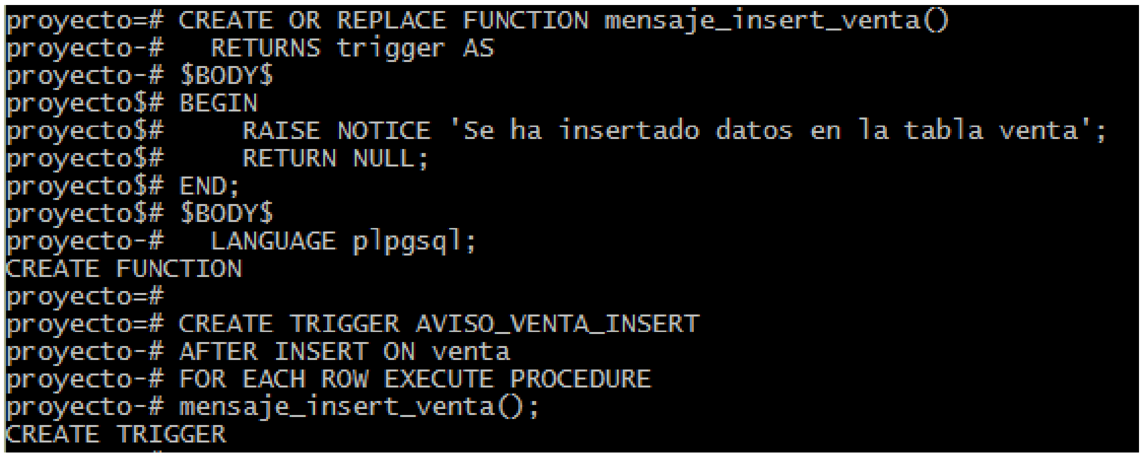
Figura 30. Insert Trigger Cliente



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_cliente()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla cliente';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_CLIENTES_INSERT
proyecto-# AFTER INSERT ON cliente
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_cliente();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_insert_venta()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla venta';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_VENTA_INSERT
AFTER INSERT ON venta
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_venta();
```

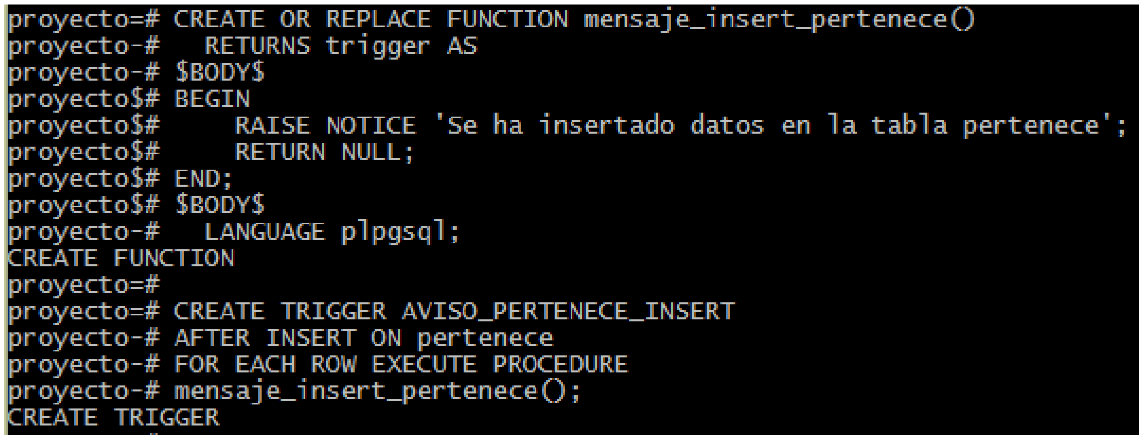
Figura 31. Insert Trigger Venta



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_venta()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla venta';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_VENTA_INSERT
proyecto-# AFTER INSERT ON venta
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_venta();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_insert_pertenece()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha insertado datos en la tabla pertenece';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PERTENECE_INSERT
AFTER INSERT ON pertenece
FOR EACH ROW EXECUTE PROCEDURE
mensaje_insert_pertenece();
```

Figura 32. Insert Trigger Pertenece



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_insert_pertenece()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha insertado datos en la tabla pertenece';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PERTENECE_INSERT
proyecto-# AFTER INSERT ON pertenece
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_insert_pertenece();
CREATE TRIGGER
```

4.4. Update Triggers and Messages Functions

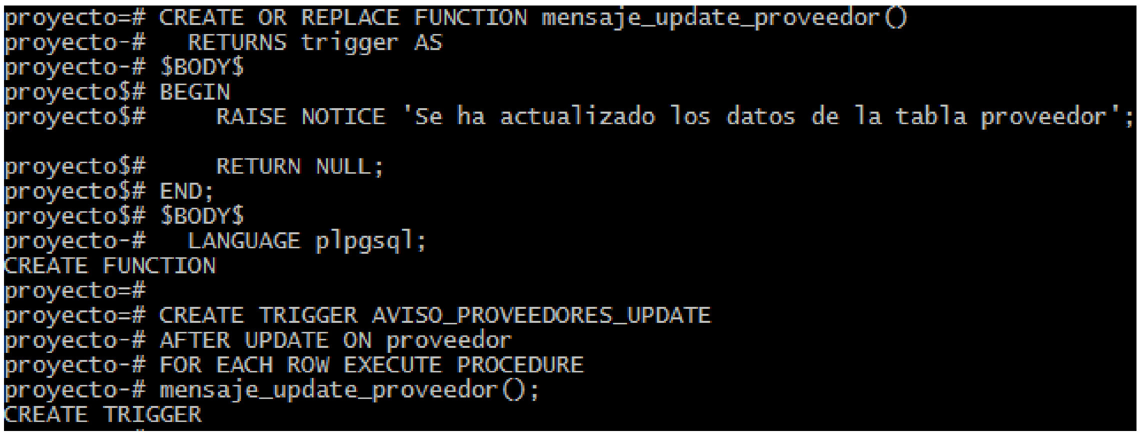
As the same as the previous section, we created a function to notify about a change in the Database System, the creation of the function is completely same as the previous ones, with the difference that this will notify about an update and not about an insertion of a register.

After creating the function, we change the sentence *INSERT AFTER INSERT ON* to *UPDATE AFTER UPDATE ON* so this trigger will just be executed when an update is being made on the Database system, then we call the previous created function to message the one who is managing the system using the same words as the insert triggers.

We can see how we created the different triggers for updating as it follows:

```
CREATE OR REPLACE FUNCTION mensaje_update_proveedor()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla proveedor';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PROVEEDORES_UPDATE
AFTER UPDATE ON proveedor
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_proveedor();
```

Figura 33. Update Trigger Proveedor

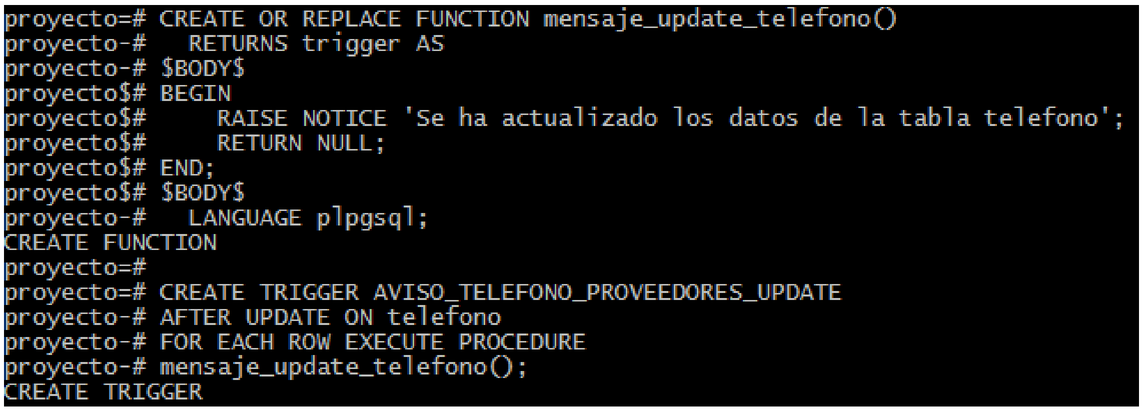


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_proveedor()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla proveedor';

proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PROVEEDORES_UPDATE
proyecto-# AFTER UPDATE ON proveedor
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_proveedor();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_telefono()
  RETURNS trigger AS
```

```
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla telefono';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_UPDATE
AFTER UPDATE ON telefono
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_telefono();
```

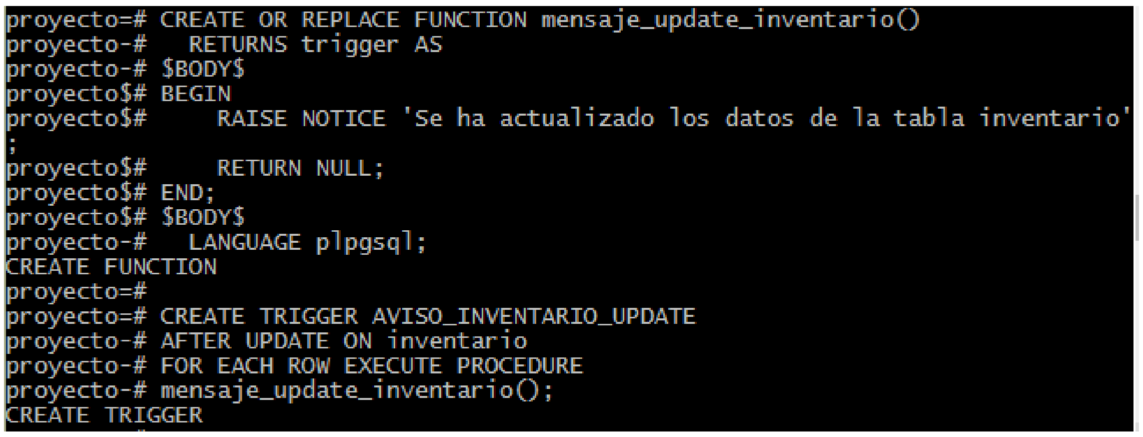
Figura 34. Update Trigger Telefono



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_telefono()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla telefono';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_UPDATE
proyecto-# AFTER UPDATE ON telefono
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_telefono();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_inventario()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla inventario';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_INVENTARIO_UPDATE
AFTER UPDATE ON inventario
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_inventario();
```

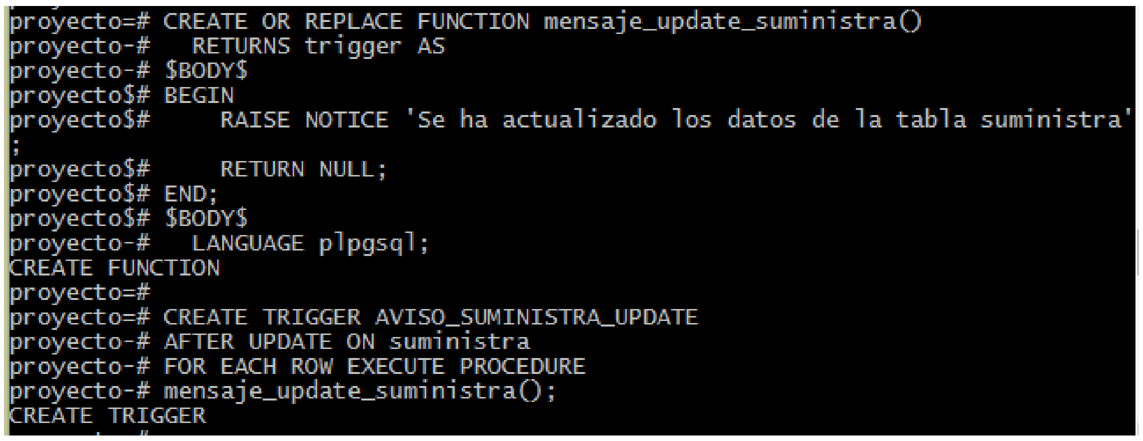
Figura 35. Update Trigger Inventario



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_inventario()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla inventario'
;
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_INVENTARIO_UPDATE
proyecto-# AFTER UPDATE ON inventario
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_inventario();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_suministra()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla suministra';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_SUMINISTRA_UPDATE
AFTER UPDATE ON suministra
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_suministra();
```

Figura 36. Update Trigger Suministra

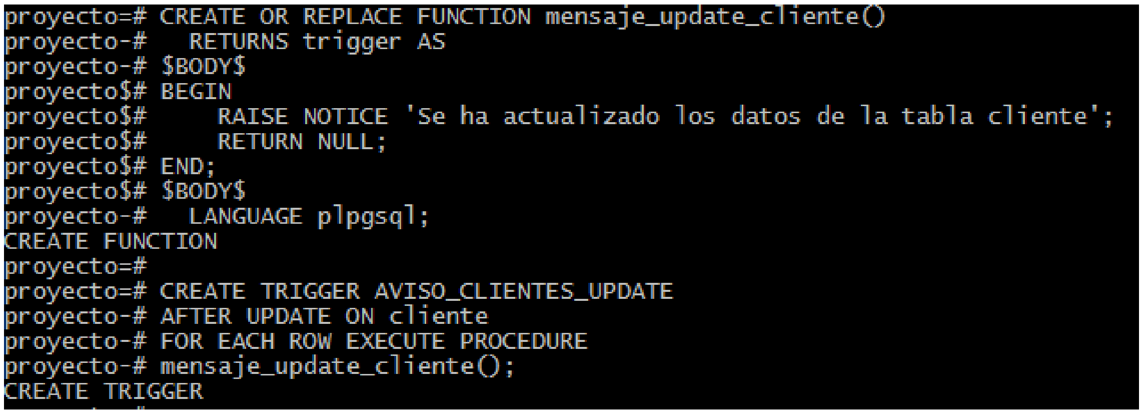


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_suministra()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla suministra'
;
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_SUMINISTRA_UPDATE
proyecto-# AFTER UPDATE ON suministra
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_suministra();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_cliente()
```

```
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla cliente';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_CLIENTES_UPDATE
AFTER UPDATE ON cliente
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_cliente();
```

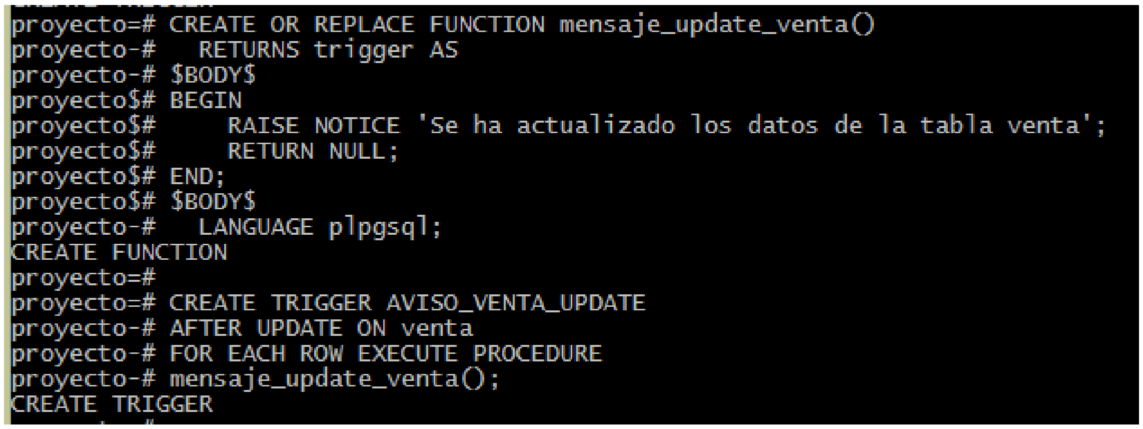
Figura 37. Update Trigger Cliente



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_cliente()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla cliente';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_CLIENTES_UPDATE
proyecto-# AFTER UPDATE ON cliente
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_cliente();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_venta()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha actualizado los datos de la tabla venta';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_VENTA_UPDATE
AFTER UPDATE ON venta
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_venta();
```

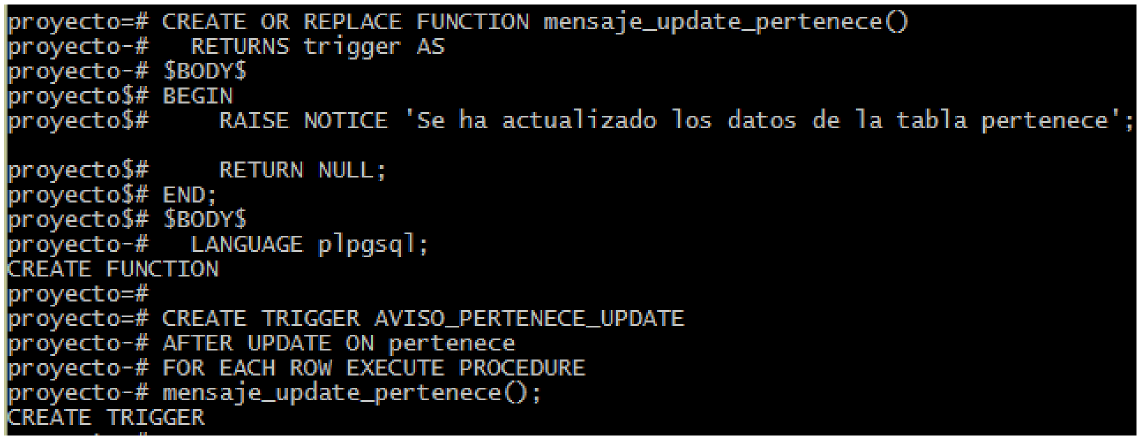
Figura 38. Update Trigger Venta



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_venta()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla venta';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_VENTA_UPDATE
proyecto-# AFTER UPDATE ON venta
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_venta();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_update_pertenece()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE ’Se ha actualizado los datos de la tabla pertenece’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PERTENECE_UPDATE
AFTER UPDATE ON pertenece
FOR EACH ROW EXECUTE PROCEDURE
mensaje_update_pertenece();
```

Figura 39. Update Trigger Pertenece



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_update_pertenece()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha actualizado los datos de la tabla pertenece';

proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PERTENECE_UPDATE
proyecto-# AFTER UPDATE ON pertenece
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_update_pertenece();
CREATE TRIGGER
```

4.5. Delete Triggers and Messages Functions

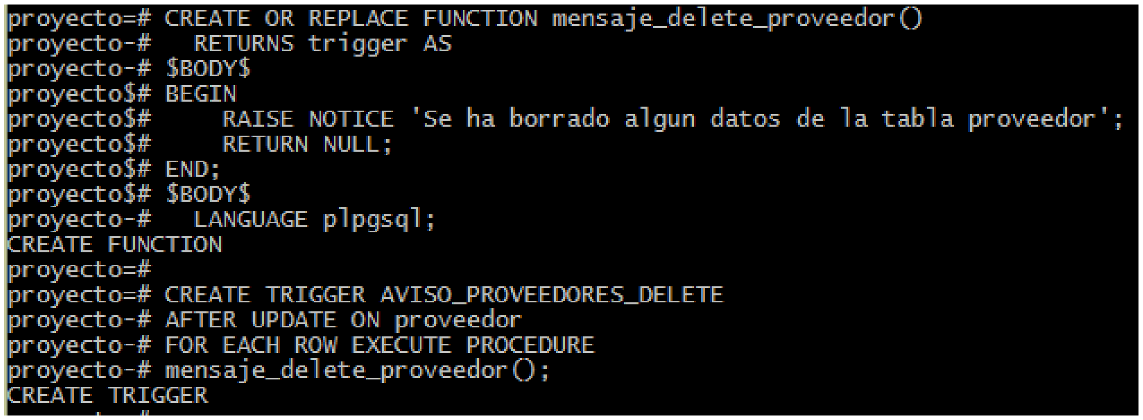
As for the previous functions, we use the exact same sentences and word to make the function, the difference is that in the *RAISE NOTE* sentence we changed updated for delete, so we can know this message function is working for the porpouse of deleting information of the system.

For the delete triggers, we change *UPDATE AFTER UPDATE ON* to *DELETE AFTER UPDATE ON*, this means the trigger will be executed if a register is being deleted, and the call the previous function we created for the trigger.

We can see how we created the different triggers for deleting as it follows:

```
CREATE OR REPLACE FUNCTION mensaje_delete_proveedor()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha borrado algun datos de la tabla proveedor';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PROVEEDORES_DELETE
AFTER UPDATE ON proveedor
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_proveedor();
```

Figura 40. Delete Trigger Proveedor

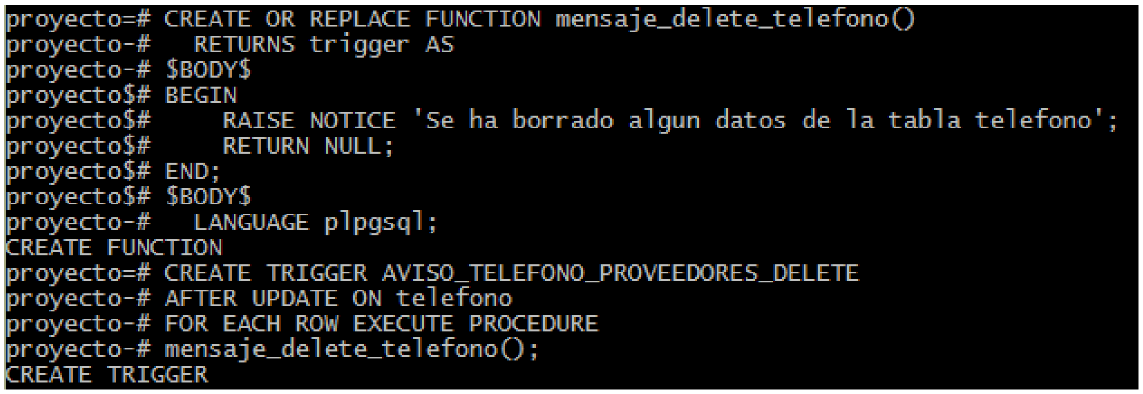


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_proveedor()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla proveedor';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PROVEEDORES_DELETE
proyecto-# AFTER UPDATE ON proveedor
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_proveedor();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_telefono()
  RETURNS trigger AS
$BODY$
```

```
BEGIN
    RAISE NOTICE ’Se ha borrado algun datos de la tabla telefono’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_DELETE
AFTER UPDATE ON telefono
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_telefono();
```

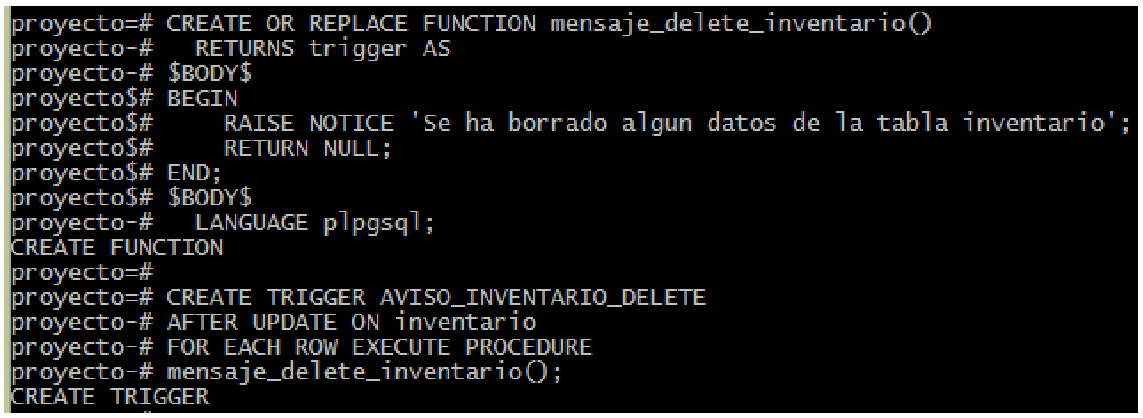
Figura 41. Delete Trigger Telefono



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_telefono()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla telefono';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=# CREATE TRIGGER AVISO_TELEFONO_PROVEEDORES_DELETE
proyecto-# AFTER UPDATE ON telefono
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_telefono();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_inventario()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE ’Se ha borrado algun datos de la tabla inventario’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_INVENTARIO_DELETE
AFTER UPDATE ON inventario
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_inventario();
```

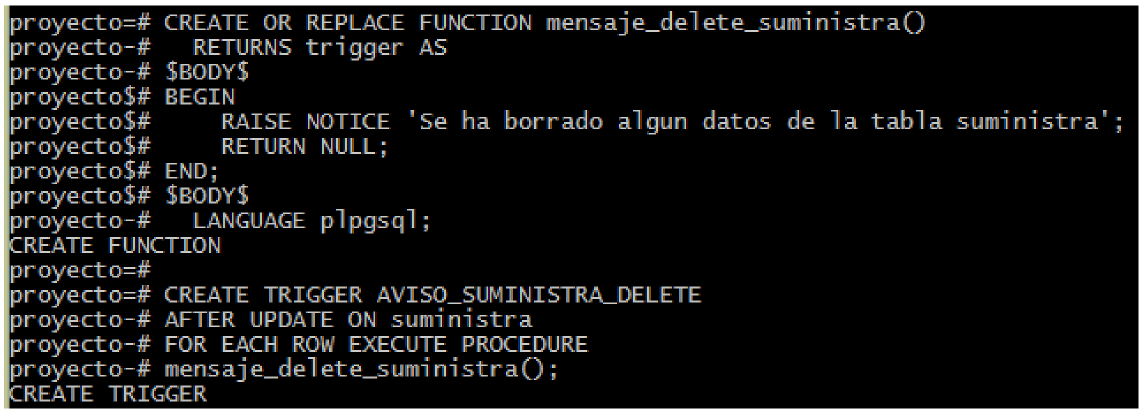
Figura 42. Delete Trigger Inventario



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_inventario()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla inventario';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_INVENTARIO_DELETE
proyecto-# AFTER UPDATE ON inventario
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_inventario();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_suministra()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE 'Se ha borrado algun datos de la tabla suministra';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_SUMINISTRA_DELETE
AFTER UPDATE ON suministra
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_suministra();
```

Figura 43. Delete Trigger Suministra

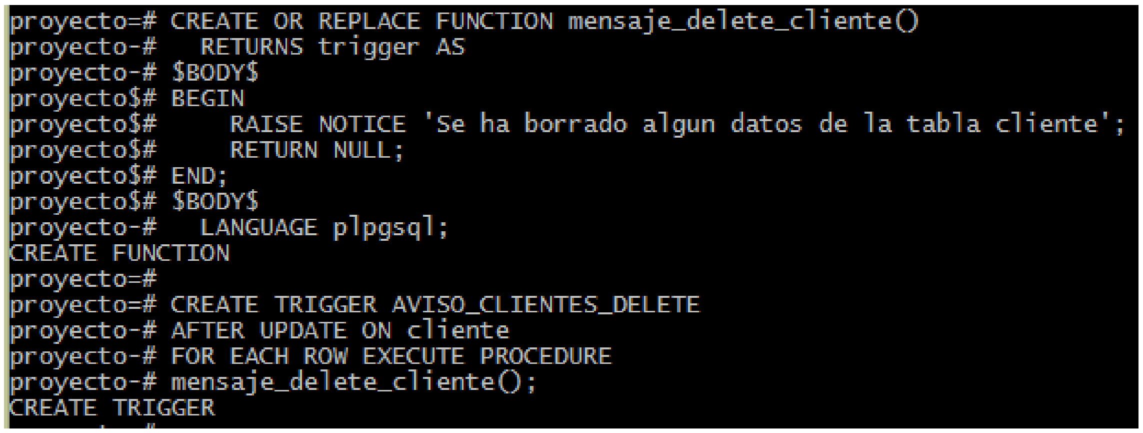


```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_suministra()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla suministra';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_SUMINISTRA_DELETE
proyecto-# AFTER UPDATE ON suministra
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_suministra();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_cliente()
  RETURNS trigger AS
```

```
$BODY$
BEGIN
    RAISE NOTICE ’Se ha borrado algun datos de la tabla cliente’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_CLIENTES_DELETE
AFTER UPDATE ON cliente
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_cliente();
```

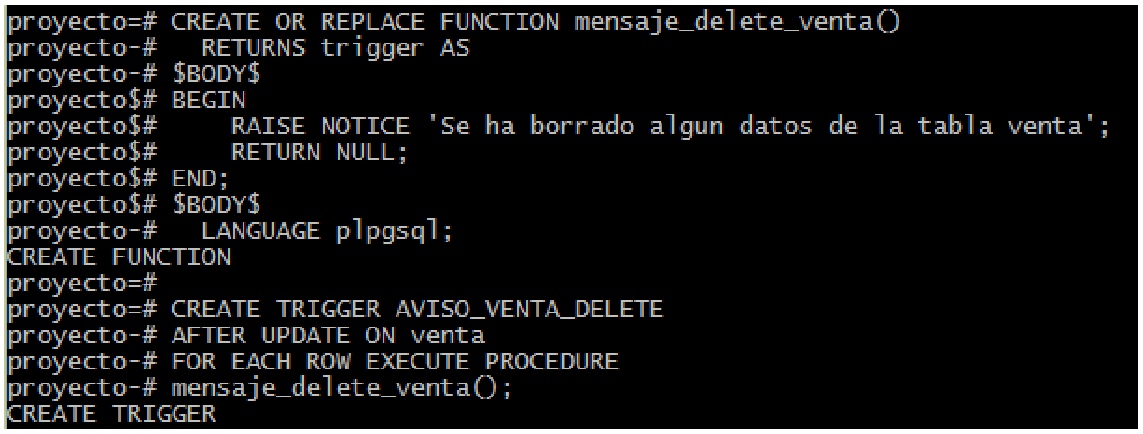
Figura 44. Delete Trigger Cliente



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_cliente()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla cliente';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_CLIENTES_DELETE
proyecto-# AFTER UPDATE ON cliente
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_cliente();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_venta()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE ’Se ha borrado algun datos de la tabla venta’;
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_VENTA_DELETE
AFTER UPDATE ON venta
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_venta();
```

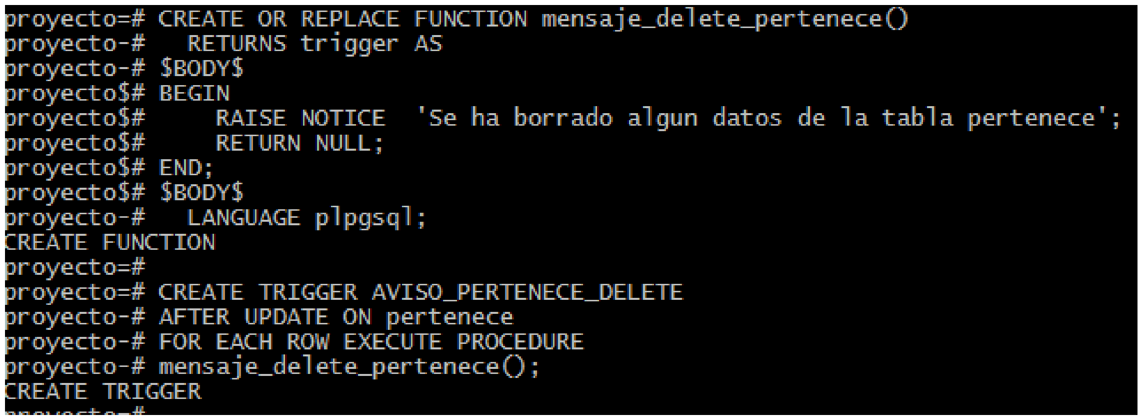
Figura 45. Delete Trigger Venta



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_venta()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE 'Se ha borrado algun datos de la tabla venta';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_VENTA_DELETE
proyecto-# AFTER UPDATE ON venta
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_venta();
CREATE TRIGGER
```

```
CREATE OR REPLACE FUNCTION mensaje_delete_pertenece()
  RETURNS trigger AS
$BODY$
BEGIN
    RAISE NOTICE  'Se ha borrado algun datos de la tabla pertenece';
    RETURN NULL;
END;
$BODY$
  LANGUAGE plpgsql;

CREATE TRIGGER AVISO_PERTENECE_DELETE
AFTER UPDATE ON pertenece
FOR EACH ROW EXECUTE PROCEDURE
mensaje_delete_pertenece();
```

Figura 46. Delete Trigger Pertenece



```
proyecto=# CREATE OR REPLACE FUNCTION mensaje_delete_pertenece()
proyecto-#   RETURNS trigger AS
proyecto-# $BODY$
proyecto$# BEGIN
proyecto$#     RAISE NOTICE  'Se ha borrado algun datos de la tabla pertenece';
proyecto$#     RETURN NULL;
proyecto$# END;
proyecto$# $BODY$
proyecto-#   LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER AVISO_PERTENECE_DELETE
proyecto-# AFTER UPDATE ON pertenece
proyecto-# FOR EACH ROW EXECUTE PROCEDURE
proyecto-# mensaje_delete_pertenece();
CREATE TRIGGER
```

4.6. Audits for Insert, Update and Delete Triggers

To make a proper audit for each of our tables, we need to create an extra table for each of the previous ones, so the different updates can be seen in that table. The table will have the exact same attributes so it can be tested.

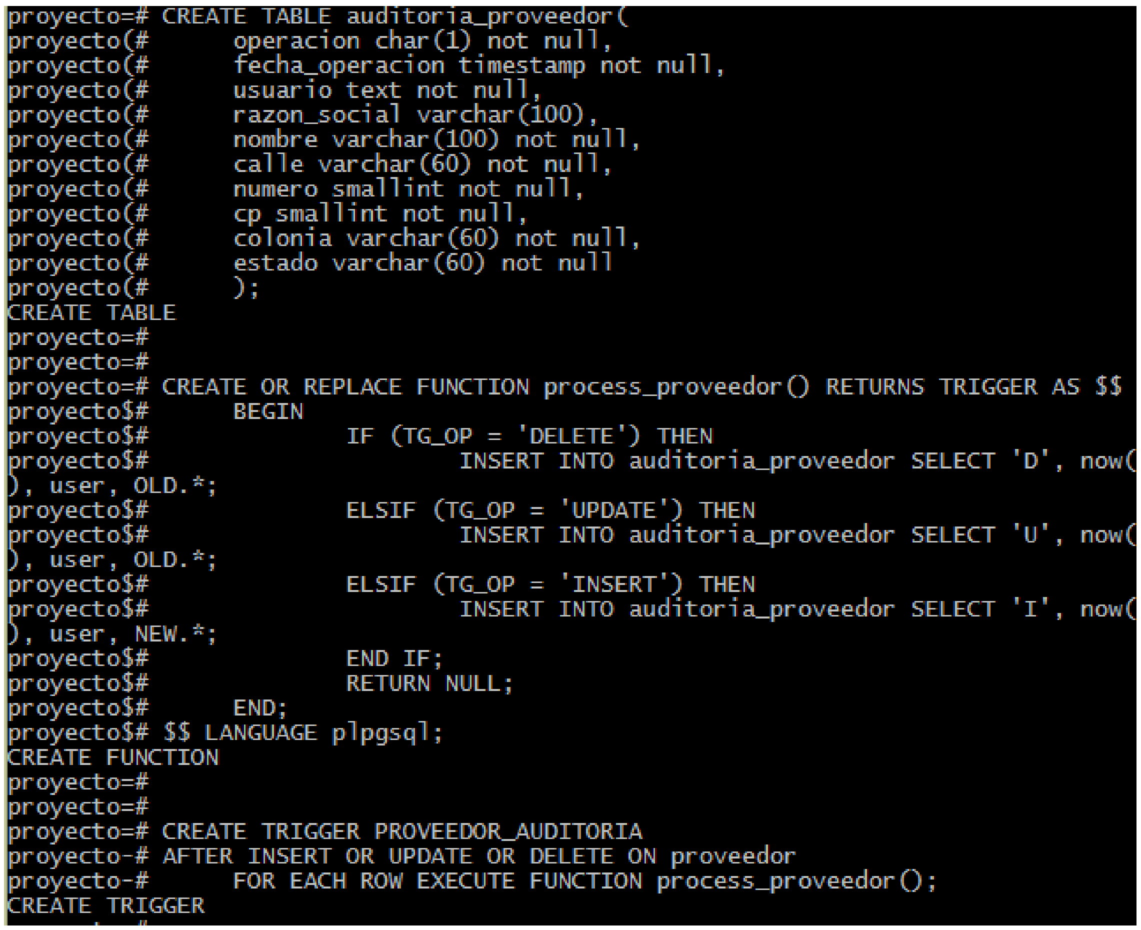
After creating the table, we created a function with conditionals *IF* and *ELSIF*, in each part of the conditional, we check if we are working with an insert, an update or with a delete function that is being applied to the system.

Finally we created a trigger with *AFTER INSERT OR UPDATE OR DELETE ON* and then the name of the table, so we can execute the previous function.

We can see how we created the process of the audits as it follows:

```
CREATE TABLE auditoria_proveedor(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
razon_social varchar(100),
nombre varchar(100) not null,
calle varchar(60) not null,
numero smallint not null,
cp smallint not null,
colonia varchar(60) not null,
estado varchar(60) not null
);


CREATE OR REPLACE FUNCTION process_proveedor() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN
INSERT INTO auditoria_proveedor SELECT 'D', now(), user, OLD.*;
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_proveedor SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_proveedor SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER PROVEEDOR_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON proveedor
FOR EACH ROW EXECUTE FUNCTION process_proveedor();
```

Figura 47. Proveedor Audit

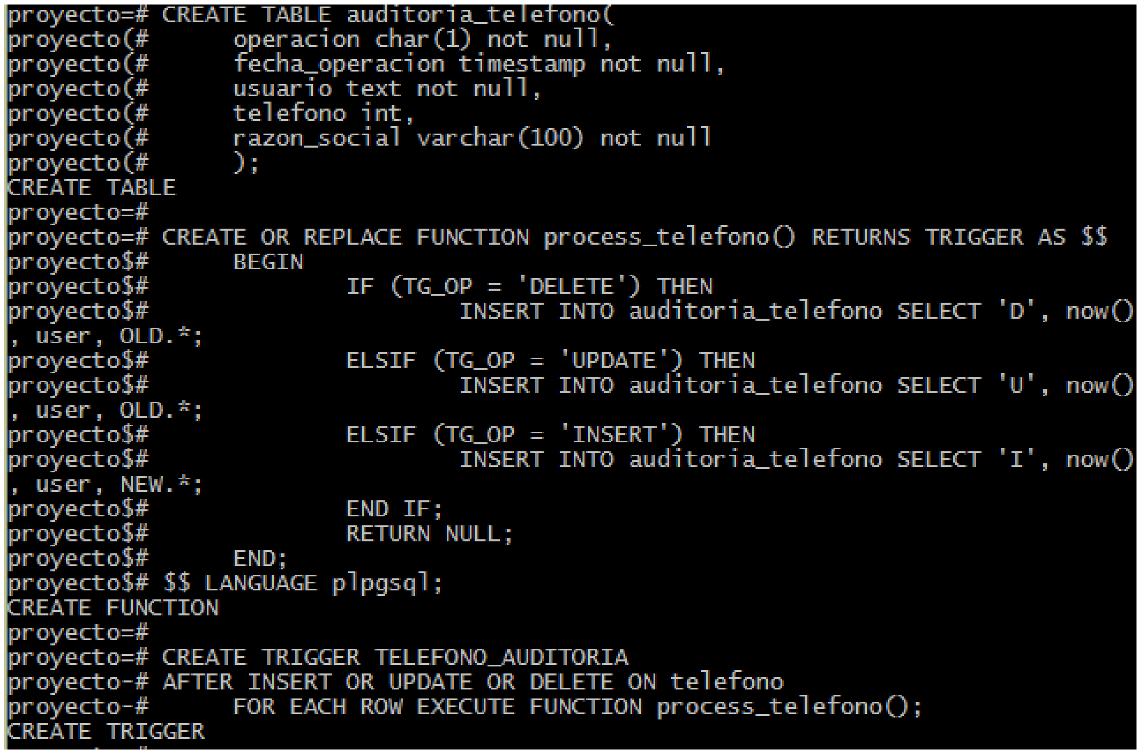


```
proyecto=# CREATE TABLE auditoria_proveedor(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      razon_social varchar(100),
proyecto(#      nombre varchar(100) not null,
proyecto(#      calle varchar(60) not null,
proyecto(#      numero smallint not null,
proyecto(#      cp smallint not null,
proyecto(#      colonia varchar(60) not null,
proyecto(#      estado varchar(60) not null
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_proveedor() RETURNS TRIGGER AS $$
proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_proveedor SELECT 'D', now(
), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_proveedor SELECT 'U', now(
), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_proveedor SELECT 'I', now(
), user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=#
proyecto=# CREATE TRIGGER PROVEEDOR_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON proveedor
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_proveedor();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_telefono(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
telefono int,
razon_social varchar(100) not null
);

CREATE OR REPLACE FUNCTION process_telefono() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = ’DELETE’) THEN
INSERT INTO auditoria_telefono SELECT ’D’, now(), user, OLD.*;
ELSIF (TG_OP = ’UPDATE’) THEN
INSERT INTO auditoria_telefono SELECT ’U’, now(), user, OLD.*;
ELSIF (TG_OP = ’INSERT’) THEN
INSERT INTO auditoria_telefono SELECT ’I’, now(), user, NEW.*;
END IF;
RETURN NULL;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER TELEFONO_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON telefono
FOR EACH ROW EXECUTE FUNCTION process_telefono();
```

Figura 48. Telefono Audit

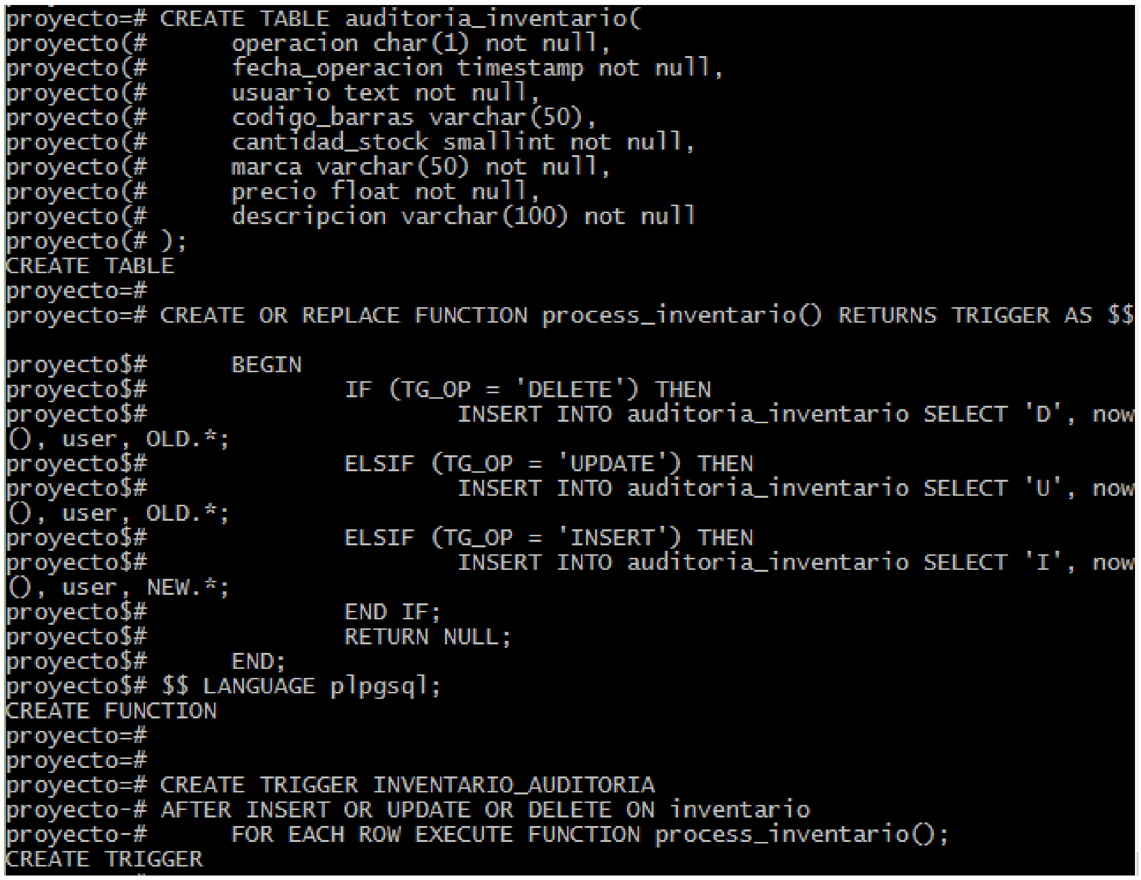


```
proyecto=# CREATE TABLE auditoria_telefono(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      telefono int,
proyecto(#      razon_social varchar(100) not null
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_telefono() RETURNS TRIGGER AS $$
proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_telefono SELECT 'D', now()
, user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_telefono SELECT 'U', now()
, user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_telefono SELECT 'I', now()
, user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER TELEFONO_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON telefono
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_telefono();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_inventario(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
codigo_barras varchar(50),
cantidad_stock smallint not null,
marca varchar(50) not null,
precio float not null,
descripcion varchar(100) not null
);

CREATE OR REPLACE FUNCTION process_inventario() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = ’DELETE’) THEN
INSERT INTO auditoria_inventario SELECT ’D’, now(), user, OLD.*;
```

```
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_inventario SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_inventario SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER INVENTARIO_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON inventario
FOR EACH ROW EXECUTE FUNCTION process_inventario();
```

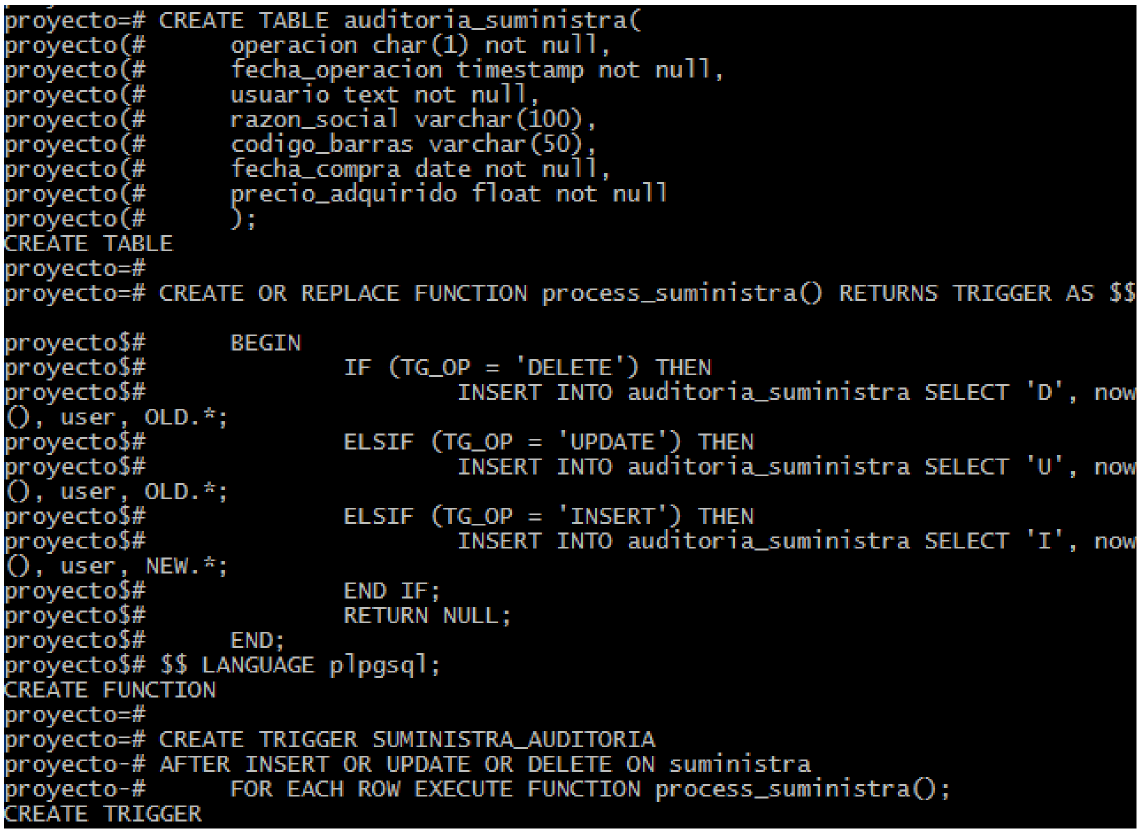
Figura 49. Inventario Audit



```
proyecto=# CREATE TABLE auditoria_inventario(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      codigo_barras varchar(50),
proyecto(#      cantidad_stock smallint not null,
proyecto(#      marca varchar(50) not null,
proyecto(#      precio float not null,
proyecto(#      descripcion varchar(100) not null
proyecto(# );
CREATE TABLE
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_inventario() RETURNS TRIGGER AS $$

proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_inventario SELECT 'D', now
(), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_inventario SELECT 'U', now
(), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_inventario SELECT 'I', now
(), user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=#
proyecto=# CREATE TRIGGER INVENTARIO_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON inventario
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_inventario();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_suministra(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
razon_social varchar(100),
codigo_barras varchar(50),
fecha_compra date not null,
precio_adquirido float not null
);

CREATE OR REPLACE FUNCTION process_suministra() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN
INSERT INTO auditoria_suministra SELECT 'D', now(), user, OLD.*;
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_suministra SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_suministra SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER SUMINISTRA_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON suministra
FOR EACH ROW EXECUTE FUNCTION process_suministra();
```

Figura 50. Suministra Audit

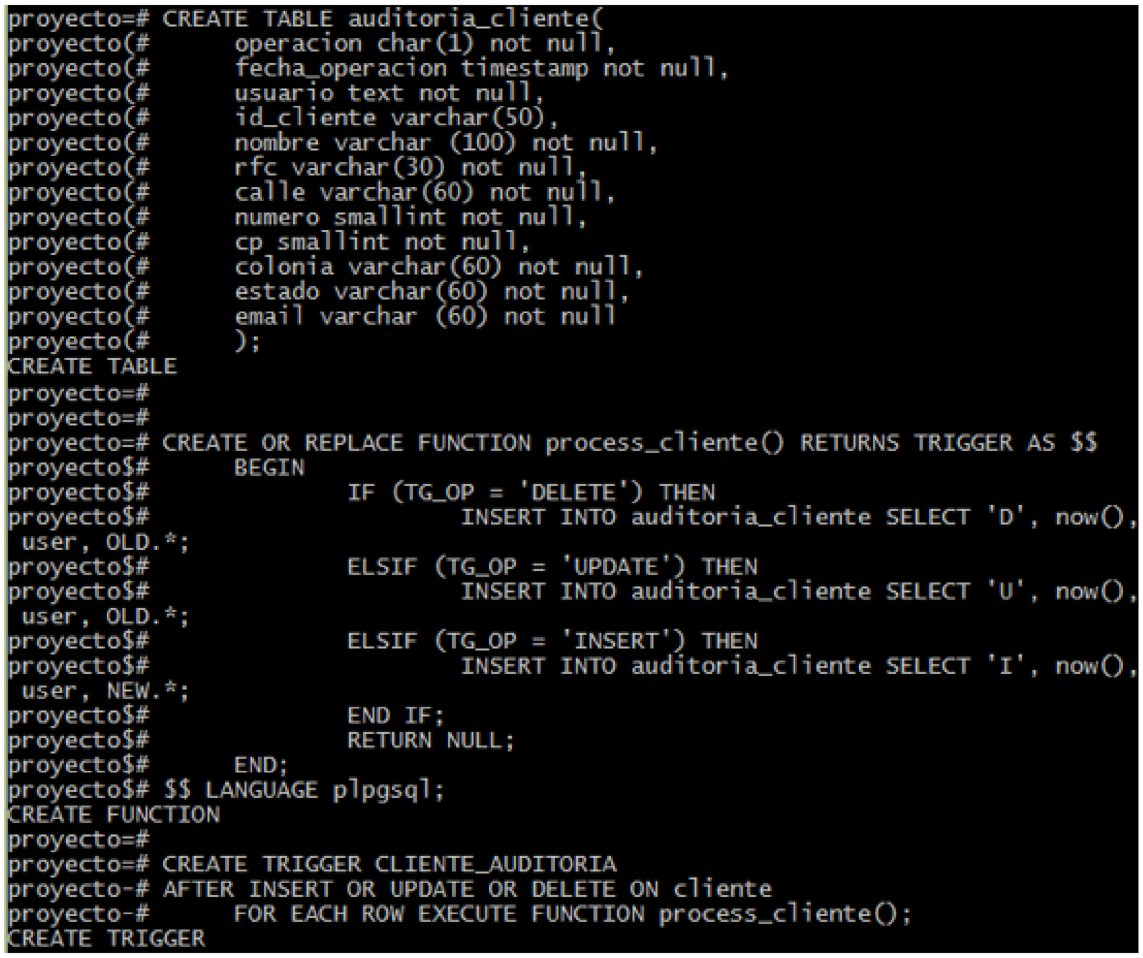


```
proyecto=# CREATE TABLE auditoria_suministra(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      razon_social varchar(100),
proyecto(#      codigo_barras varchar(50),
proyecto(#      fecha_compra date not null,
proyecto(#      precio_adquirido float not null
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_suministra() RETURNS TRIGGER AS $$

proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_suministra SELECT 'D', now
(), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_suministra SELECT 'U', now
(), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_suministra SELECT 'I', now
(), user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER SUMINISTRA_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON suministra
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_suministra();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_cliente(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
id_cliente varchar(50),
nombre varchar (100) not null,
rfc varchar(30) not null,
calle varchar(60) not null,
numero smallint not null,
cp smallint not null,
colonia varchar(60) not null,
estado varchar(60) not null,
email varchar (60) not null
);


CREATE OR REPLACE FUNCTION process_cliente() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = ’DELETE’) THEN
INSERT INTO auditoria_cliente SELECT ’D’, now(), user, OLD.*;
```

```
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_cliente SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_cliente SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER CLIENTE_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON cliente
FOR EACH ROW EXECUTE FUNCTION process_cliente();
```

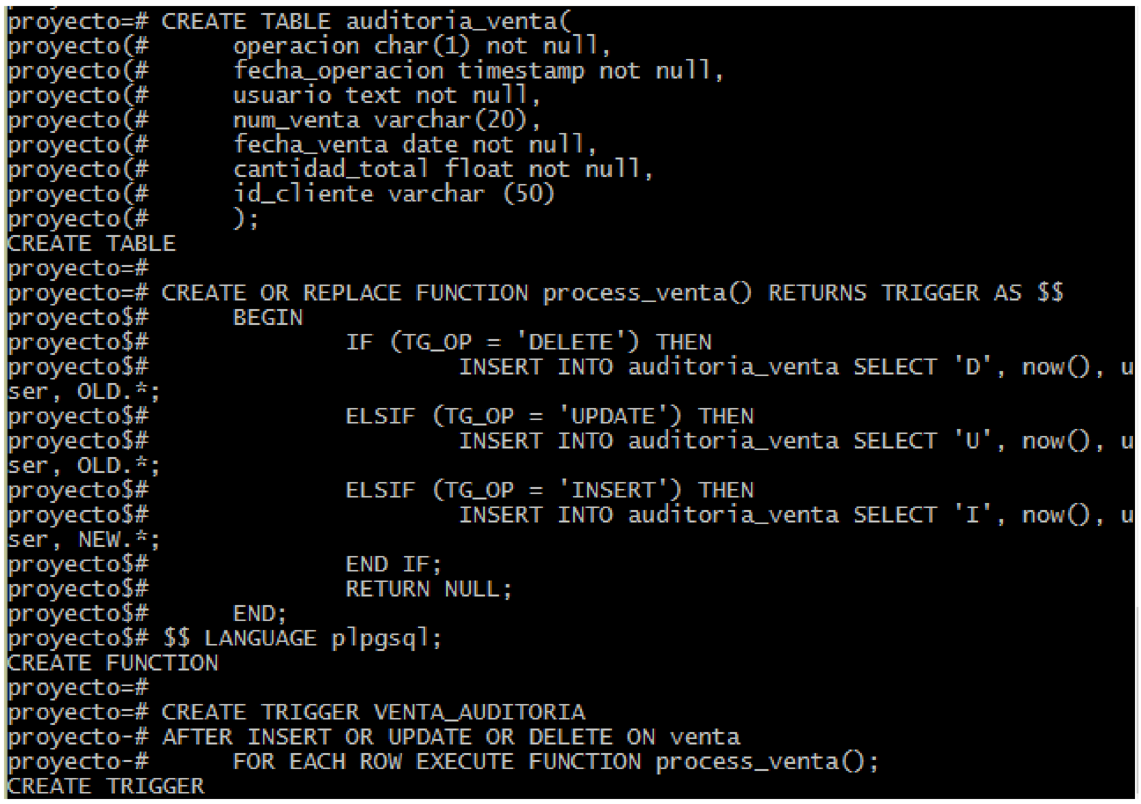
Figura 51. Cliente Audit



```
proyecto=# CREATE TABLE auditoria_cliente(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      id_cliente varchar(50),
proyecto(#      nombre varchar (100) not null,
proyecto(#      rfc varchar(30) not null,
proyecto(#      calle varchar(60) not null,
proyecto(#      numero smallint not null,
proyecto(#      cp smallint not null,
proyecto(#      colonia varchar(60) not null,
proyecto(#      estado varchar(60) not null,
proyecto(#      email varchar (60) not null
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_cliente() RETURNS TRIGGER AS $$
proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_cliente SELECT 'D', now(),
 user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_cliente SELECT 'U', now(),
 user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_cliente SELECT 'I', now(),
 user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER CLIENTE_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON cliente
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_cliente();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_venta(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
num_venta varchar(20),
fecha_venta date not null,
cantidad_total float not null,
id_cliente varchar (50)
);

CREATE OR REPLACE FUNCTION process_venta() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN
INSERT INTO auditoria_venta SELECT 'D', now(), user, OLD.*;
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_venta SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_venta SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER VENTA_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON venta
FOR EACH ROW EXECUTE FUNCTION process_venta();
```

Figura 52. Venta Audit

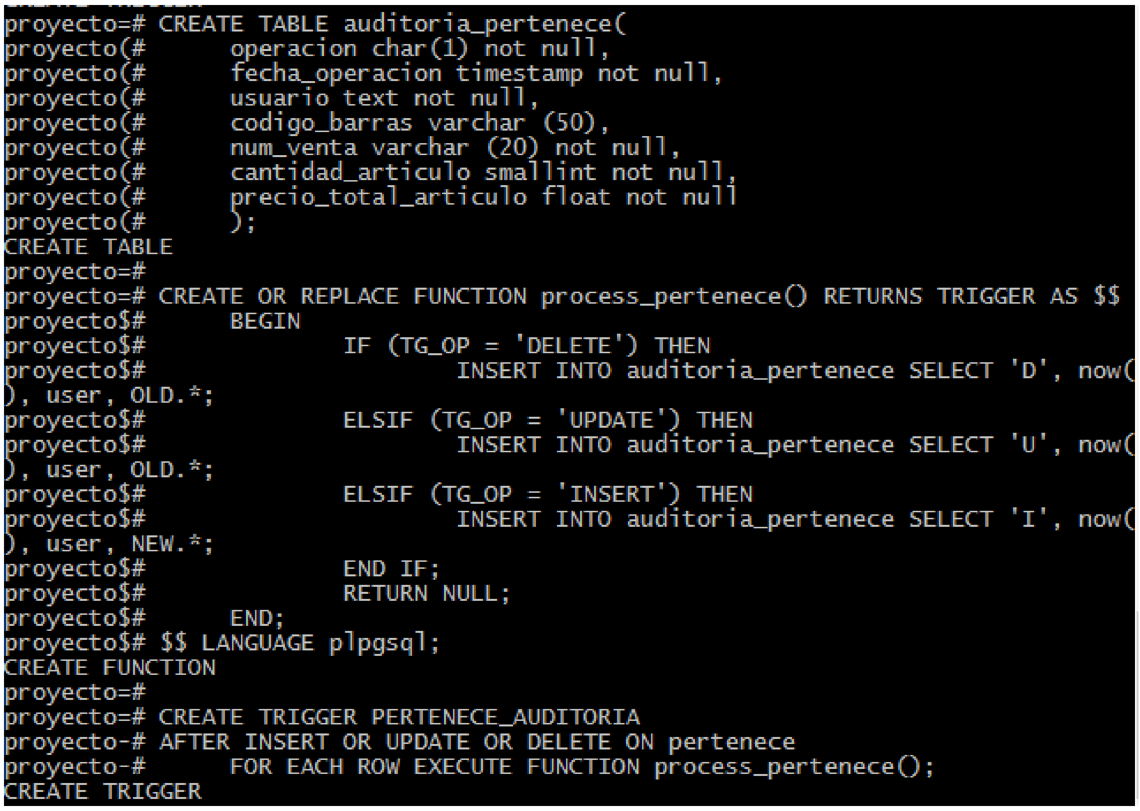


```
proyecto=# CREATE TABLE auditoria_venta(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      num_venta varchar(20),
proyecto(#      fecha_venta date not null,
proyecto(#      cantidad_total float not null,
proyecto(#      id_cliente varchar (50)
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_venta() RETURNS TRIGGER AS $$
proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_venta SELECT 'D', now(), u
ser, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_venta SELECT 'U', now(), u
ser, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_venta SELECT 'I', now(), u
ser, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER VENTA_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON venta
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_venta();
CREATE TRIGGER
```

```
CREATE TABLE auditoria_pertenece(
operacion char(1) not null,
fecha_operacion timestamp not null,
usuario text not null,
codigo_barras varchar (50),
num_venta varchar (20) not null,
cantidad_articulo smallint not null,
precio_total_articulo float not null
);

CREATE OR REPLACE FUNCTION process_pertenece() RETURNS TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN
INSERT INTO auditoria_pertenece SELECT 'D', now(), user, OLD.*;
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO auditoria_pertenece SELECT 'U', now(), user, OLD.*;
ELSIF (TG_OP = 'INSERT') THEN
INSERT INTO auditoria_pertenece SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER PERTENECE_AUDITORIA
AFTER INSERT OR UPDATE OR DELETE ON pertenece
FOR EACH ROW EXECUTE FUNCTION process_pertenece();
```

Figura 53. Pertenece Audit



```
proyecto=# CREATE TABLE auditoria_pertenece(
proyecto(#      operacion char(1) not null,
proyecto(#      fecha_operacion timestamp not null,
proyecto(#      usuario text not null,
proyecto(#      codigo_barras varchar (50),
proyecto(#      num_venta varchar (20) not null,
proyecto(#      cantidad_articulo smallint not null,
proyecto(#      precio_total_articulo float not null
proyecto(#      );
CREATE TABLE
proyecto=#
proyecto=# CREATE OR REPLACE FUNCTION process_pertenece() RETURNS TRIGGER AS $$
proyecto$#      BEGIN
proyecto$#              IF (TG_OP = 'DELETE') THEN
proyecto$#                      INSERT INTO auditoria_pertenece SELECT 'D', now(
), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'UPDATE') THEN
proyecto$#                      INSERT INTO auditoria_pertenece SELECT 'U', now(
), user, OLD.*;
proyecto$#              ELSIF (TG_OP = 'INSERT') THEN
proyecto$#                      INSERT INTO auditoria_pertenece SELECT 'I', now(
), user, NEW.*;
proyecto$#              END IF;
proyecto$#              RETURN NULL;
proyecto$#      END;
proyecto$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
proyecto=#
proyecto=# CREATE TRIGGER PERTENECE_AUDITORIA
proyecto-# AFTER INSERT OR UPDATE OR DELETE ON pertenece
proyecto-#      FOR EACH ROW EXECUTE FUNCTION process_pertenece();
CREATE TRIGGER
```

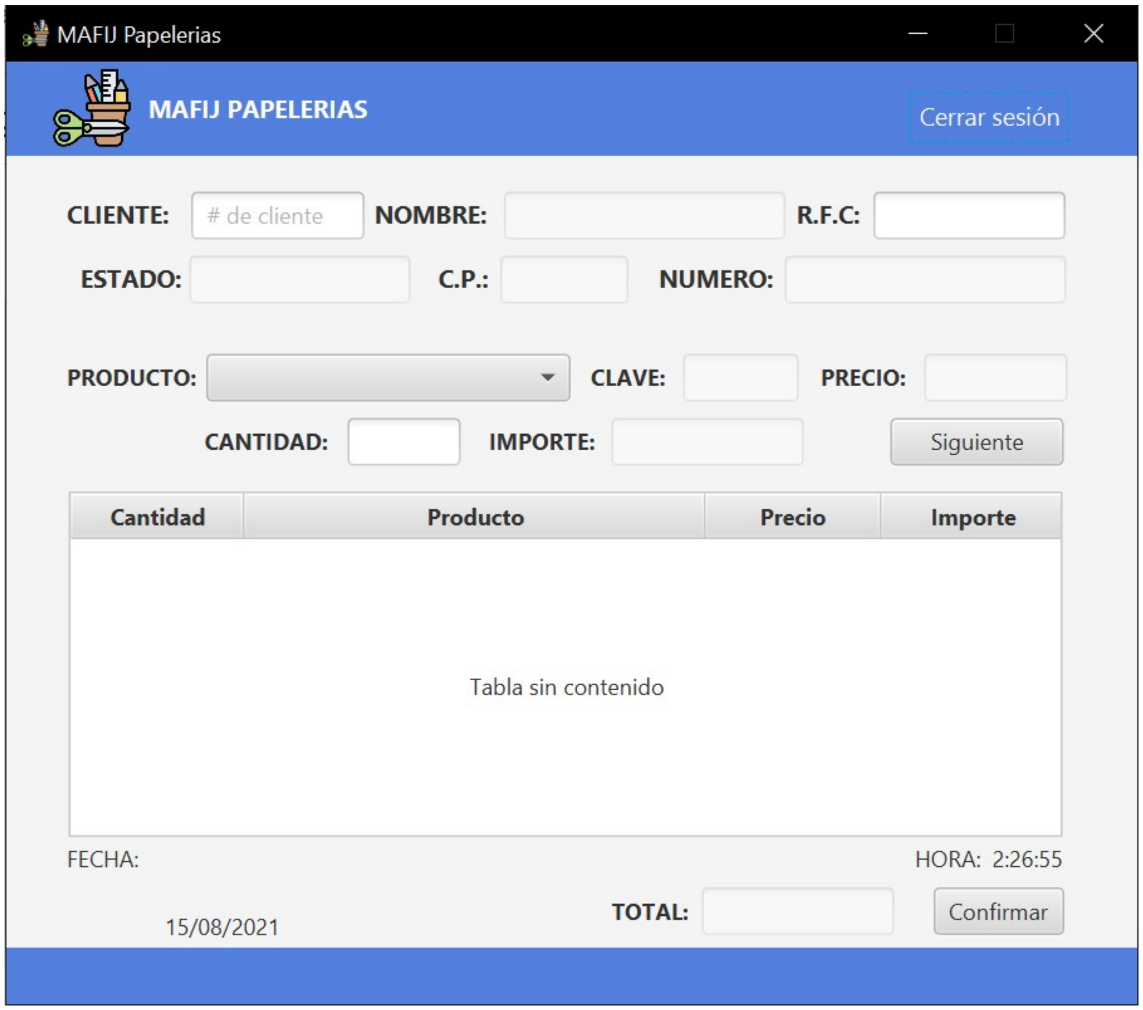
5. Presentation

The presentation we made, is the graphic interface we created for a stationary store with the name ”MAFIG”, they are acting as our client, it was made in Java language so it can be easily connected to the PostGreSQL Database.

5.1. Main Menu

When we enter to the main page of the store, we can see the most important information about the client, and the information about the product the client want to buy, being showed by a table that will sum all the different products that will be purchased.

Figura 54. Main Menu for a Sale



Cantidad	Producto	Precio	Importe

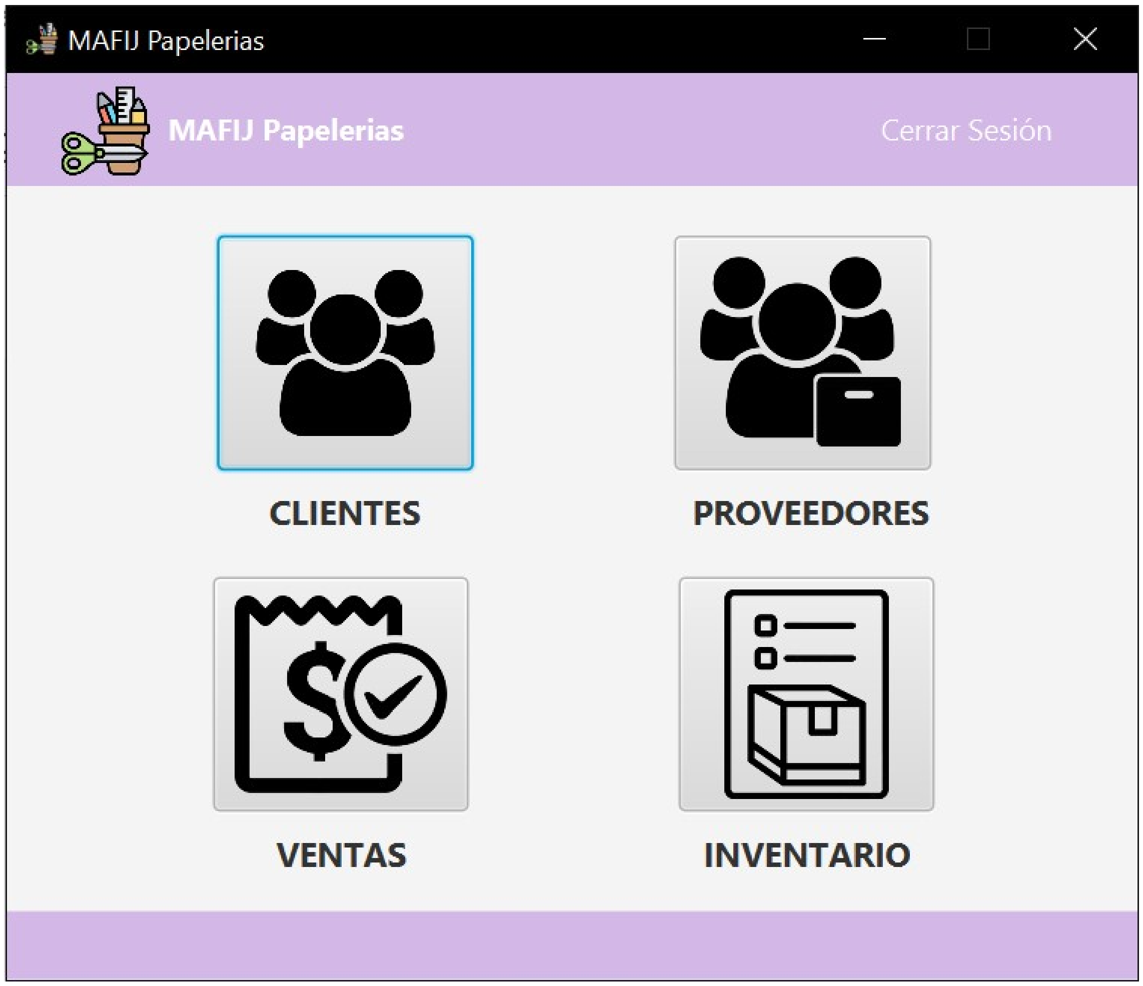
Tabla sin contenido

FECHA: 15/08/2021

HORA: 2:26:55

The administrator of the database system will also have a menu to navigate with the principal information about the clients, suppliers, sales and inventory the stationary store has, so the information that needs to be checked or clarified is easy accessible without any problem.

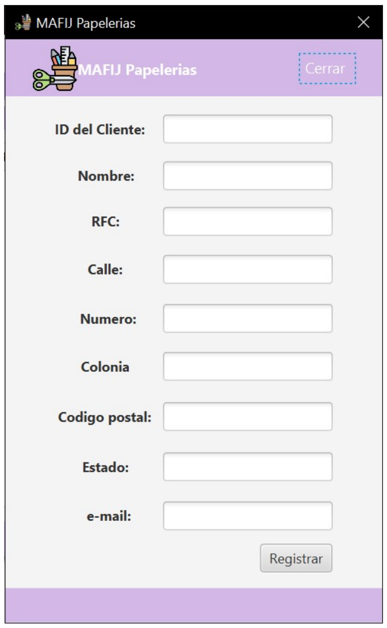
Figura 55. Main Menu for the Admin



5.2. Adding

When we have a new client to our store, our duty is to add it to the database system as fast and easy as possible, so we made a window where the client can provide us with the necessary information, in order to purchase all the items that are needed.

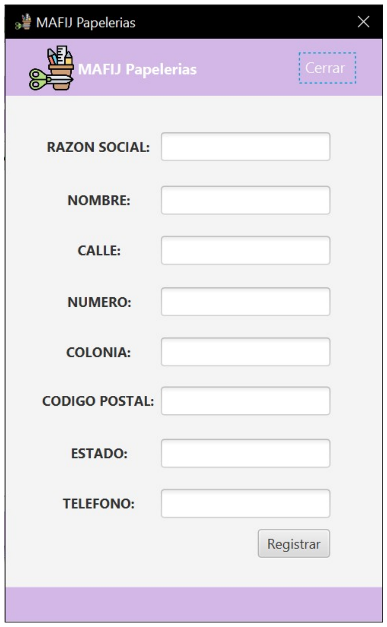
Figura 56. Adding a Client


MAFIJ Papelerias

ID del Cliente:	
Nombre:	
RFC:	
Calle:	
Numero:	
Colonia	
Codigo postal:	
Estado:	
e-mail:	

When we want to add a Supplier that want us to sale their products, we also need them to be in our database system, where the necessary information has to be recorded, so it is easy to know which products we buy from them and what kind of products are these.

Figura 57. Adding a Supplier


MAFIJ Papelerias

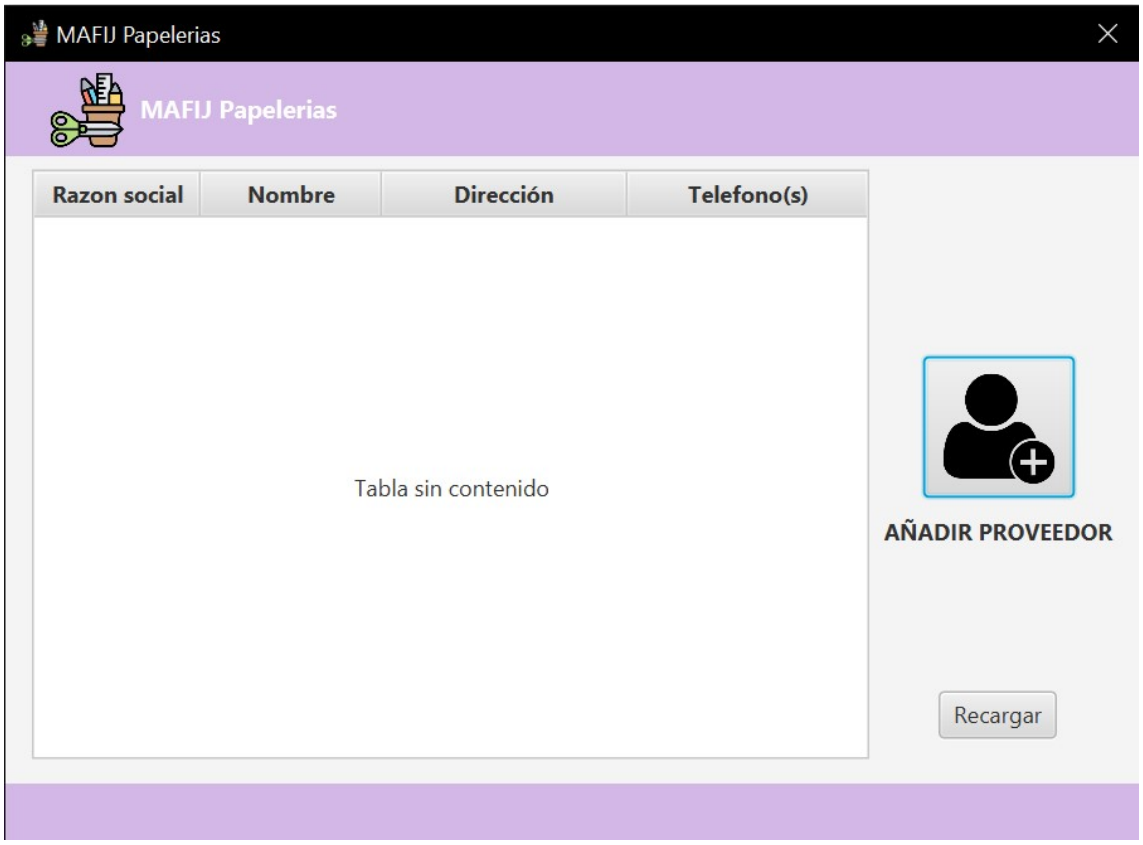
RAZON SOCIAL:	
NOMBRE:	
CALLE:	
NUMERO:	
COLONIA:	
CODIGO POSTAL:	
ESTADO:	
TELEFONO:	

5.3. Viewing Information

It is very important to everyone of us to see all the information we have in the database system, not only about our sales, but also about the clients, suppliers and inventory, so we can always have the information up to date, and in case we make a change, does not matter if it is a small one or a big one, we can see all the information being treated.

The different windows are shown like these:

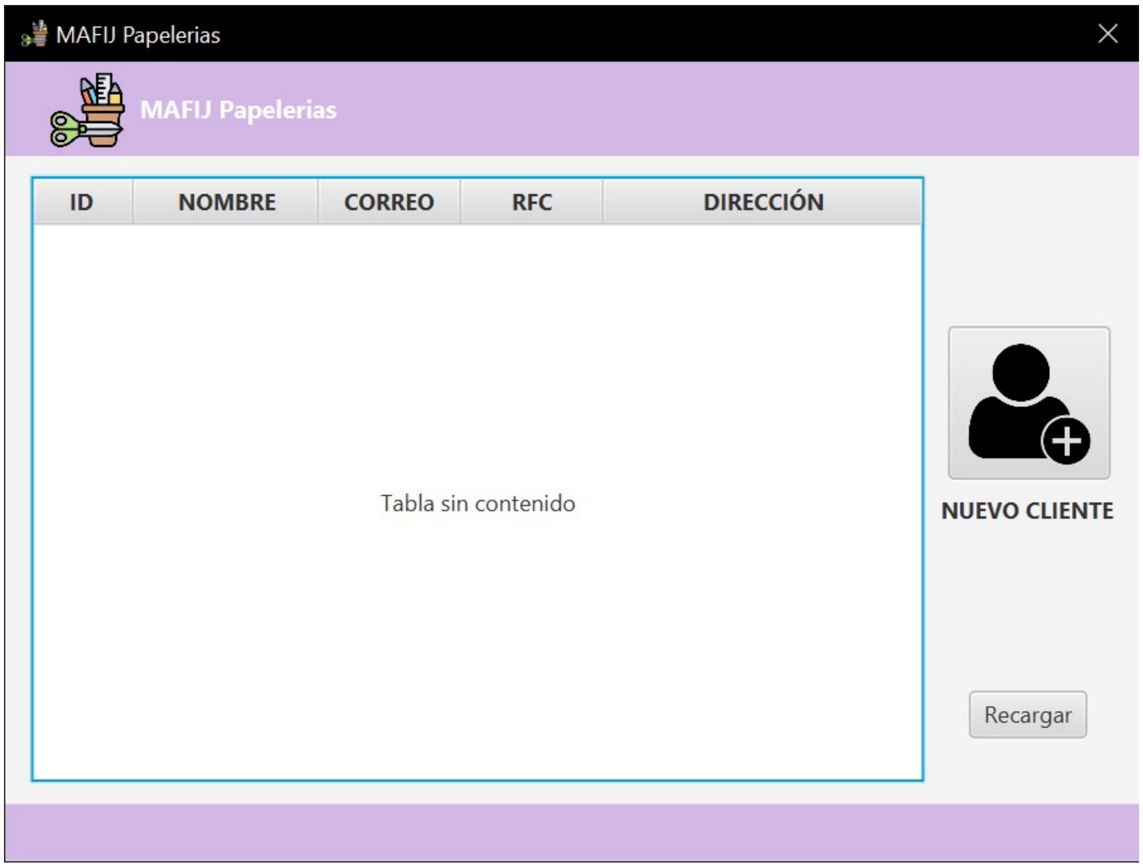
Figura 58. Supplier's Information



Razon social	Nombre	Dirección	Telefono(s)

Tabla sin contenido

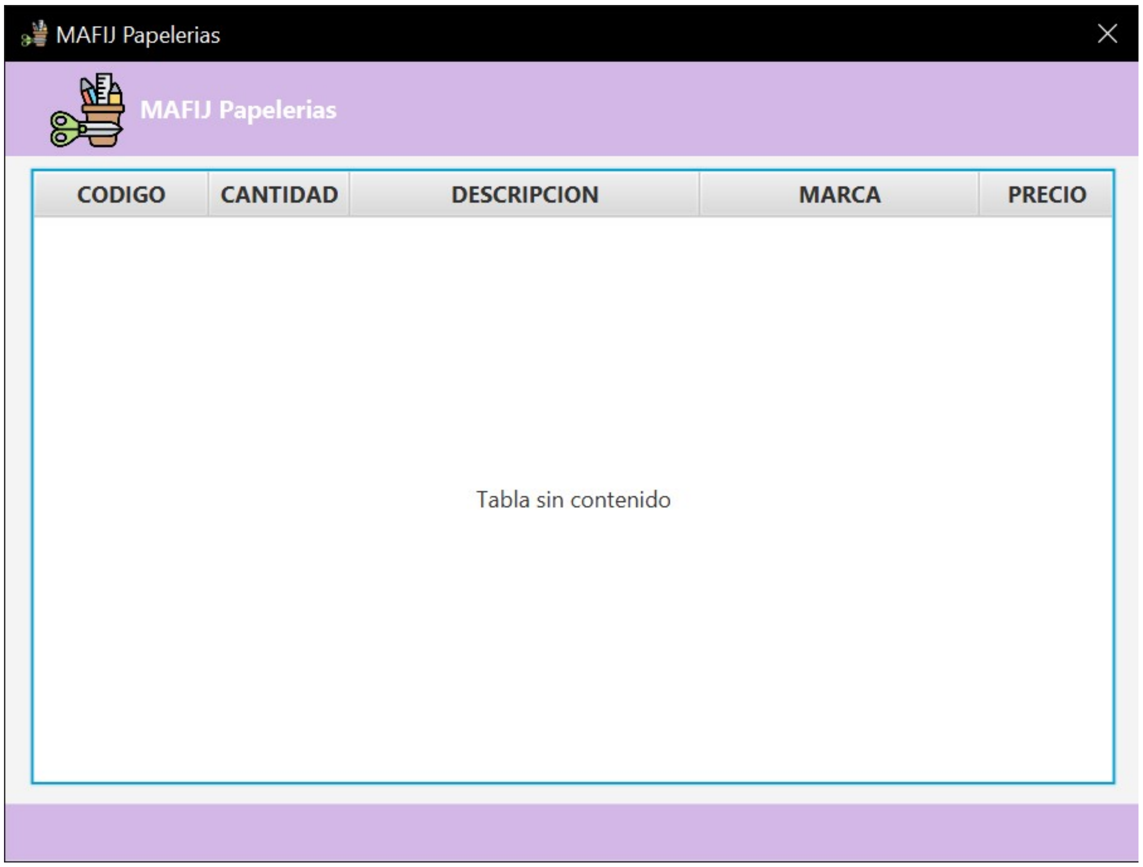
Figura 59. Client’s Information



ID	NOMBRE	CORREO	RFC	DIRECCIÓN

Tabla sin contenido

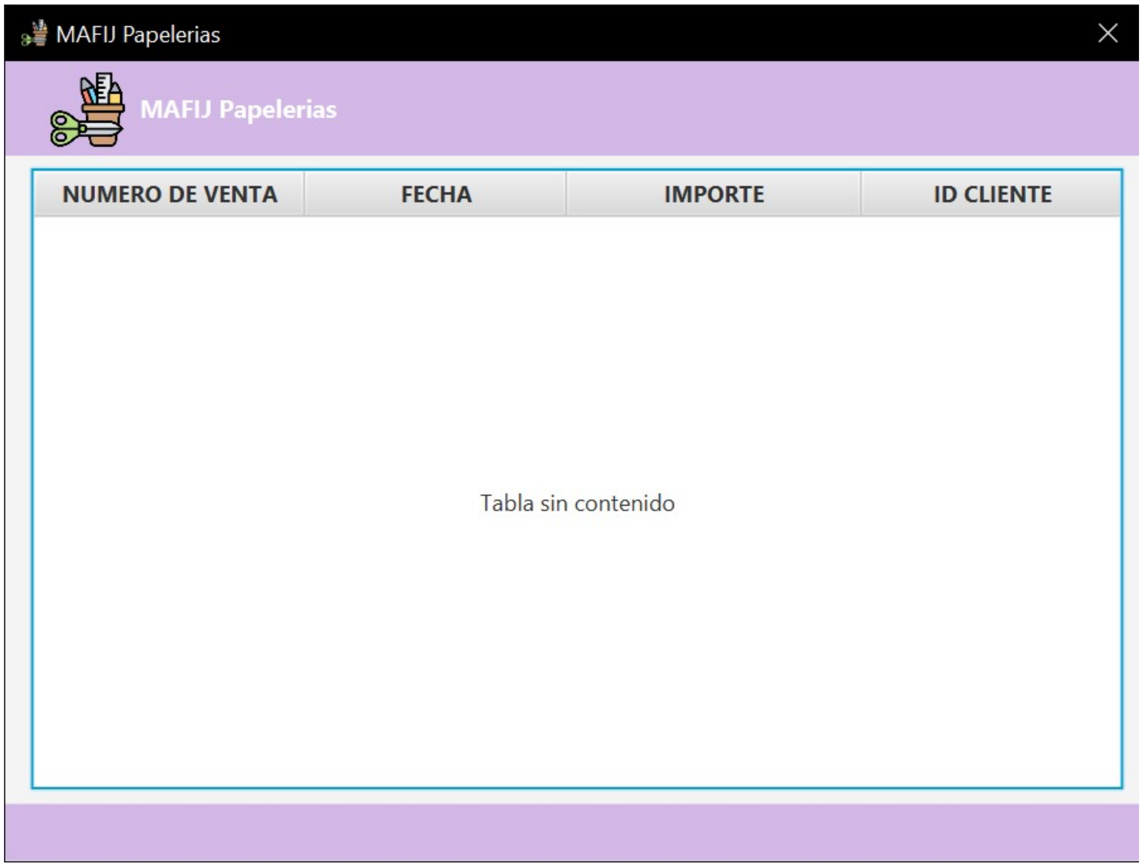
Figura 60. Inventory's Information



CODIGO	CANTIDAD	DESCRIPCION	MARCA	PRECIO

Tabla sin contenido

Figura 61. Sale's Information



NUMERO DE VENTA	FECHA	IMPORTE	ID CLIENTE

Tabla sin contenido

6. Conclusions

6.1. Marín Márquez Jonathan

I learned in this proyect about the correct management of my time, so I can divide my different tasks and homework in a way to work efficiently as a team. Also I learnt a lot about the design and implementation of a database system working with PosGreSQL.

What was the most difficult part for me, was the correct way to connect the graphic interface to a remote database system. Doing some research and trying different ways to solve this problem I got into a different solution.

6.2. Ramírez Flores Eslavica Monserrat

The objectives of the project were met since the concepts seen in theory were tested, from the abstraction of the problem, the development and architecture of the database and its respective execution in a handler that was postgreSQL.

I reinforced my previous knowledge of management and software projects for the documentation that was requested, I learned the meaning of organization to carry out a teamwork plan, so that in the future I will have good practices to contribute collaboratively and have good results in a work environment.

6.3. Rea Aparicio Angel David

This project teached me about working in an organized team and how the effort of all the team can achieve a good project.

It was not easy to make some translations of the documentation, due to the fact of redundancy words in Spanish but it was a great opportunity to remember how to work on a formal document.
I comprehend the process of the creation of a database system and the importance of a good analysis in all the aspects of the project, also I noticed a very important detail about the database systems, the fact that making them and give information to them is extremely slow and I think, that is the motive why these systems improve every year.

6.4. Soriano Bonilla Carlos Ivan

In the project I learnt how to make a database system from zero, at the beginning we started the analysis of the problem and then the project requirements with the correct understanding to design the database system, these are the entity-relation diagram and the relational model as well as their normalization process. All the concepts we used in the project where used to make the different sentences of the SQL language using PostGreSQL.

What we saw in class was implemented in this project, I understood in a better way when we have a direct problem to be analized and try to do a good solution within our perspective. During the porject there where some problems at the design process with the relationships and other troubles with the different sentences of PostGreSQL due to a bad syntaxis.

6.5. Valenzuela García de León Fernando Rodrigo

In this project I learn a lot about working in a team, and how to make a good management of my teammates' abilities so we could make the project as easy as possible for all of us, also I had to make extra time to understand how the PDF creator language that is LATEX, so I could make the documentation as proper as it has to be, also to have knowledge about each part of the teammates work, so I can extract the important information to make a good documentation about it.

The most difficult part about this, was learning how to use LATEX with all the functions about inserting images, different sections and how to make a good presentation for it, also, when a teammate gave me information I had to investigate a bit about the different functions and sentences they used so I can explain them and also I had test my English skills so I could translate what they wanted me to write as properly as I can, making this project one that gave me a lot of experiences in different areas.