Universidad Nacional Autonoma de Mexico

Facultad de Ingenieria

Databases

Final Project
Intelligent Management System for a Stationery Retail Chain

Professor: Eng. Fernando Arreola Franco

Group: 1

Blue Team:

Calzada Lozada Sergio Armando
De la Peña Osorio Lilian
Hernández Ledesma Emiliano
Maldonado Jr. Montoya Gustavo
Moreno Vázquez Esteban
Ramírez Peña Luis Enrique
Núñez Badillo Armando Adair

Submission date: December 6th, 2025

Semester: 2026-1

1 Introduction

The purpose of this document is to analyze, design, and implement a comprehensive database solution for a chain of stationery stores. The proposal covers the entire process, from understanding the problem to building conceptual, logical, and physical models, as well as implementing the required logic in PostgreSQL to guarantee the correct functioning of the system.

The project seeks to address the requirements established by the client, ensuring efficient management of suppliers, customers, inventory, products, and sales. The solution is based on the database design principles reviewed during the course and aims to deliver a system that meets each requirement.

2 Problem Analysis

The requested system must allow the registration and management of information related to suppliers, customers, products, inventory, and sales in a stationery store chain. Currently, this information is not organized, which makes it difficult to operate the business, control stock, and obtain relevant reports.

The analysis of the problem makes it possible to identify needs such as automating inventory reduction, generating totals per sale, obtaining reports by period, controlling low-stock products, and recording each operation in detail. To achieve this, it is necessary to correctly design the database and implement the required functionalities through stored procedures, functions, views, and indexes.

3 Client Requirements

Below is a summary of the requirements established by the client for the first part of the project.

3.1 Information Requirements

- Register supplier data: business name, address, name, and phone numbers.
- Register customer data: RFC, name, address, and at least one email address.
- Manage inventory: barcode, purchase price, photograph, purchase date, and quantity in stock.
- Register product information: brand, description, and price.
- Register sales: sale number with format `VENT-00`, date, and total amount to pay.
- Register sale details: quantity per item and total per item.

3.2 Functional Requirements

- Generate a view that simulates an invoice.
- Calculate total sales and profit for a given date or date range.
- Decrease stock when selling a product and cancel the sale if stock reaches zero.

- Emit an alert when stock is less than three units.
- Automatically update line totals per item and the total of the sale.
- Obtain the names of the products with fewer than three units in stock.
- Calculate the profit of a product based on its barcode.
- Create at least one index, justifying its type and placement.

4 Work Plan

The development of the project is organized into phases to ensure a clear and orderly execution of each activity.

- **Phase 1: Problem analysis.** Review of the project document and identification of requirements.
- **Phase 2: Database design.** Development of the entity–relationship model and relational model.
- **Phase 3: Implementation.** Creation of tables, data insertion, views, functions, triggers, and indexes.
- **Phase 4: Testing and validation.** Verification of component functionality and necessary adjustments.
- **Phase 5: Final document.** Integration of diagrams, code, and explanations in LaTeX.
- **Phase 6: Presentation.** Preparation and rehearsal of the formal presentation.

Task distribution:

- **De la Peña Osorio Lilian:**
  - Development of the table creation script (CREATE TABLE).
  - Implementation of income and revenue queries.
- **Maldonado Jr. Montoya Gustavo:** Responsible for developing the advanced business logic functions required by the client's specifications:
  - Function to calculate profit by barcode.
  - Function to retrieve low-stock products.
  - System consistency validation.
- **Ramírez Peña Luis Enrique:** Responsible for creating the Entity–Relationship (ER) diagram, verifying requirements, and coordinating with the team member in charge of the relational model for better understanding and evaluation.
- **Hernández Ledesma Emiliano:**

  - Devel opment of the invoice-style view (vista_factura).
  - Translation and refinement of project documentation.
  - Responsible for the presentation.

- **Núñez Badillo Armando Adair:** Responsible for implementing inventory-related triggers and functions.

- **Moreno Vázquez Esteban:**
  - Data insertion across all tables.
  - Index creation for performance optimization.

5 Phase 2: Entity–Relationship Model and Relational Model

5.1 Entity–Relationship Model

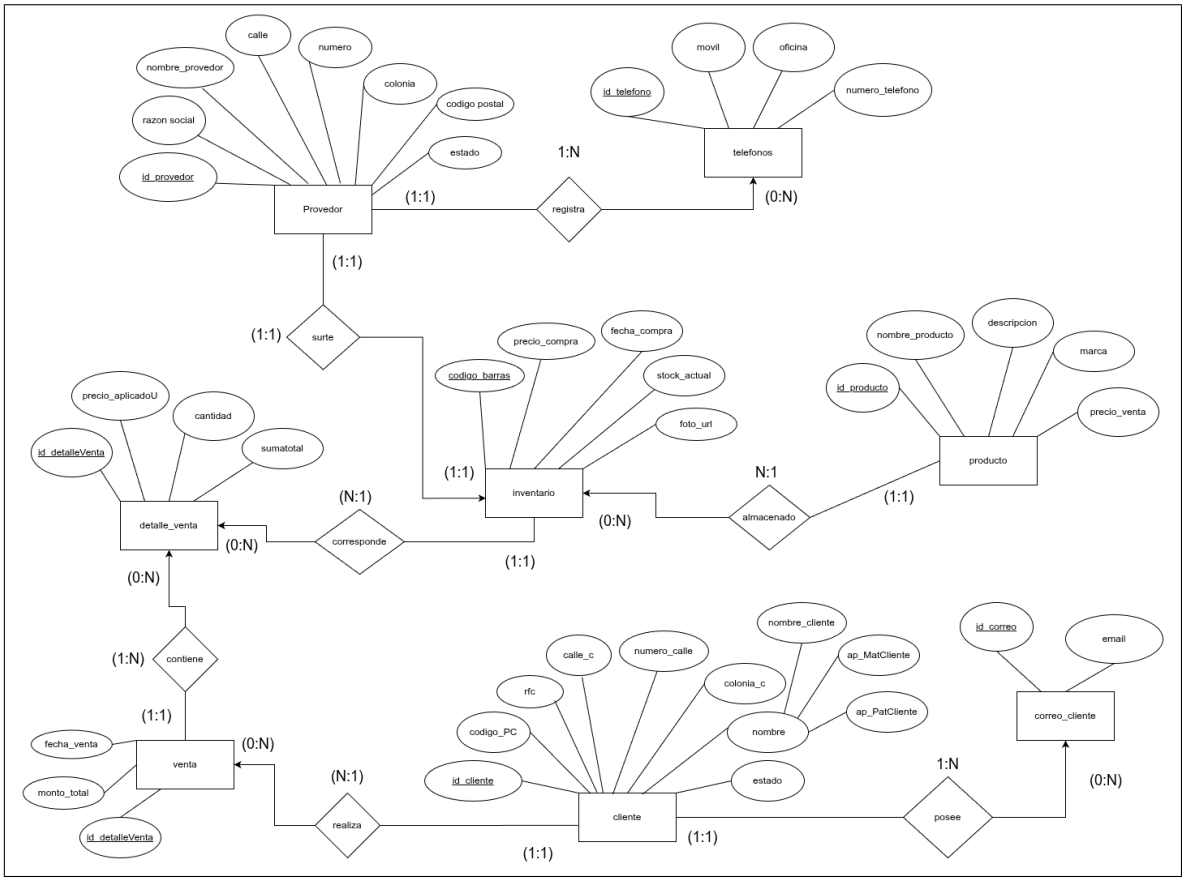

Figure 1: Entity–Relationship Model Diagram

5.2 Relational Model

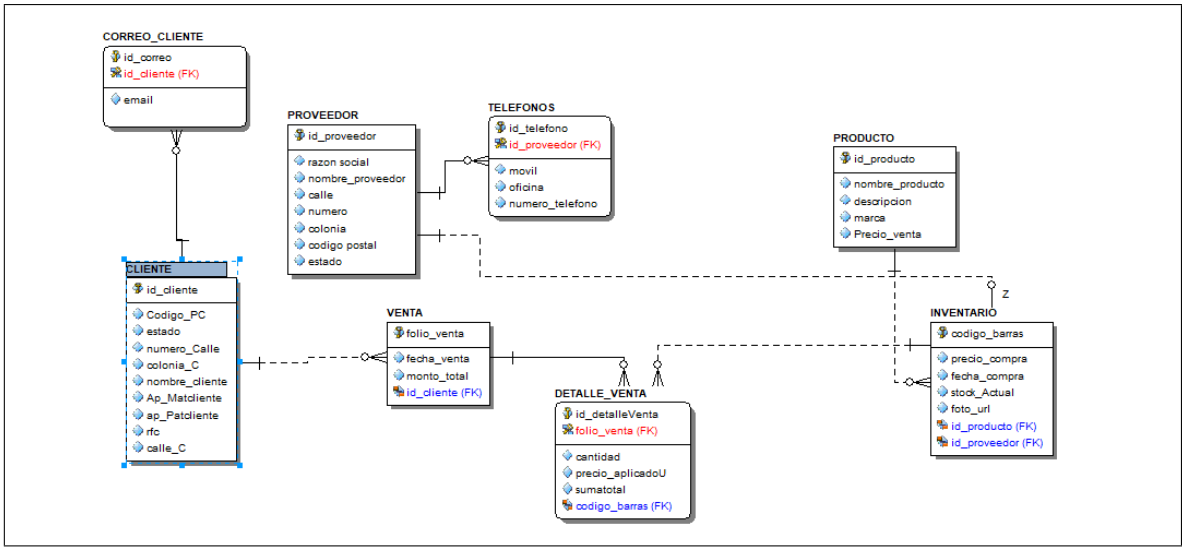

Figure 2: Relational Model Diagram

6 Implementation

6.1 Table Creation

The table creation script was designed based on the proposed relational model, with the objective of guaranteeing data integrity and coherently reflecting the relationships among customers, suppliers, products, inventory, and sales. Below, the most important design decisions are justified.

Choice of Primary Keys

In most tables, the `SERIAL` type was used as the primary key (`id_cliente`, `id_correo`, `id_proveedor`, `id_telefono`, `id_producto`, `id_detalleventa`), since it allows PostgreSQL to generate unique numeric identifiers automatically and conveniently.

In the case of the `inventario` table, the primary key was defined as the `codigo_barras` field. This decision was made because the barcode naturally and uniquely identifies each item in stock, and it is also the attribute used in the sale operation. For the `venta` table, the primary key is `folio_venta`, which corresponds to the user-visible folio with the format `VENT-001`, as requested in the requirements.

Foreign Keys and Referential Integrity

Foreign keys were defined following the relationships in the relational model:

- `correo_cliente.id_cliente` references `cliente(id_cliente)`, allowing a customer to have multiple registered email addresses. The `ON DELETE CASCADE` option was used so that, when

a customer is deleted, their associated emails are automatically removed.

- `telefonos.id_proveedor` references `proveedor(id_proveedor)`, so that each phone record is associated with a valid supplier.
- In `inventario`, the fields `id_producto` and `id_proveedor` reference `producto(id_producto)` and `proveedor(id_proveedor)`, respectively, ensuring that each barcode is linked to an existing product and supplier.
- In `venta`, the field `id_cliente` references `cliente(id_cliente)`, which guarantees that there are no sales without an associated customer.
- In `detalle_venta`, the field `folio_venta` references `venta(folio_venta)` with `ON DELETE CASCADE`, so that when a sale is deleted, its detail records are also removed. The field `codigo_barras` references `inventario(codigo_barras)`, linking each detail line to a specific inventory item.

These foreign keys maintain consistency among tables and prevent orphan records.

`NOT NULL`, `UNIQUE` and `CHECK` Constraints

Adjustment of the Composite Attribute `domicilio`

In the Entity–Relationship Model (ERM), the attribute `domicilio` (address) was defined as a composite attribute made up of five elements: `estado`, `código postal`, `colonia`, `calle` and `número`. In order to respect this conceptual structure and ensure proper normalization, the physical design of the relational model decomposed this attribute into five independent columns in both the `cliente` and `proveedor` tables.

The resulting columns are:

- `estado` – federal entity (state).
- `codigo_postal` – postal code with a five-digit validation.
- `colonia` – neighborhood or district.
- `calle` – street name.
- `numero` – exterior address number.

This decision facilitates the use of the composite attribute in future queries and contributes to the clarity of the model. The postal code format validation was implemented using a `CHECK` constraint that ensures the value matches the five-digit numeric pattern:

```
CHECK (codigo_postal ~ '^[0-9]{5}$')
```

With this adaptation, the physical design faithfully reflects the structure proposed in the ERM.

Certain attributes were declared as `NOT NULL` because they are essential for the system's operations. For example: `nombre`, `estado`, `codigo_postal`, `colonia`, `calle` and `numero` in `cliente`, as well as the same address components in `proveedor`, `precio_venta` in `producto`, and the quantity and price fields in `inventario`, `venta` and `detalle_venta`.

`UNIQUE` constraints were used on `cliente.rfc` and `proveedor.razon_social` to prevent duplicate customers or suppliers with the same tax identification or business name.

`CHECK` constraints were used to enforce business rules such as:

- `precio_venta > 0` and `precio_compra > 0`, ensuring that there are no negative or zero prices.
- `stock_actual >= 0`, preventing negative inventory quantities.
- `cantidad > 0` and `sumtotal >= 0` in `detalle_venta`, ensuring that sales have valid quantities and totals.
- A basic email validation, verifying that the `email` field contains at least the `@` character.

In the `telefonos` table, an additional constraint was added that requires at least one of the phone fields to be non-null, in order to avoid records without any contact information.

Generation of the Sale Folio `VENT-001`

To satisfy the requirement of a sale folio in the format `VENT-001`, the sequence `seq_folio_venta` was defined to generate consecutive numeric values. In the `venta` table, the `folio_venta` field uses this sequence as its default value:

```
'VENT-' || to_char(nextval('seq_folio_venta'), 'FM000')
```

In this way, each new sale automatically receives a unique folio with the prefix `VENT-` and three digits, padded with leading zeros when necessary (for example: `VENT-001`, `VENT-002`, `VENT-010`, etc.). The `to_char` function is used only to format the sequence number, while the increment is handled internally by `nextval`.

Additionally, the `CHECK` constraint `chk_folio_venta_formato` was added, which verifies using a regular expression that all folios comply with the `VENT-[0-9]` pattern, preventing values with a different format from the specified one.

6.2 Data Insertion

Data insertion was carried out with the intention of preserving realism in the stored values. The constraints of each table were taken into account to avoid errors, and it was verified that the data was coherent so that it made sense as a whole. Finally, a sufficient amount of data was added to clearly demonstrate the characteristics of the database.

Important points during data insertion:

- For tables that used foreign keys, it was verified that the selected identifiers existed in the referenced table.
- In the `producto` table, options for balance recharges of $50, $100 and $200 were added. In addition, printing options in black and white and in color were included, so that they could be recorded in the inventory.
- For the `inventario` table, a URL of an image of each product was added.

- The inserts in the `venta` table are consistent with the inserts in the `detalle_venta` table. For each sale, the products, total quantity, and sum of `precio_aplicado` in `detalle_venta` for records with the same folio are equivalent to their respective `monto_total` in `venta`.
- The values for `precio_compra` in the `inventario` table and `precio_venta` in the `producto` table were selected so that their difference corresponds to the profit obtained by the stationery store.

6.3 Invoice-Like View and Revenue Queries

With the goal of facilitating queries of information related to each sale, a view was implemented that integrates, in a single result, the sale data, the customer data, and the sold items. This view functions as a "logical invoice" within the database and prevents the user from having to manually write several joins every time they need to review the details of a transaction. In addition, revenue queries were developed to obtain the total sold on a specific date, the total sold in a date range, and the total profit for the business.

View `vista_factura`

The `vista_factura` view combines the `venta`, `cliente`, `correo_cliente`, `detalle_venta`, `inventario` and `producto` tables. Each row in the view represents one detail line of a sale (that is, a specific product within a folio), including at the same time the customer data and the total sale amount.

In general, the view provides:

- **Sale data**: `folio_venta`, `fecha_venta` and `monto_total`.
- **Customer data**: identifier, full name, RFC, address (`estado`, `codigo_postal`, `colonia`, `calle`, `numero`) and email.
- **Item details**: barcode, product name, brand, description, quantity, applied unit price and subtotal per line.

The final implementation of the view is as follows:

```
CREATE VIEW vista_factura AS
SELECT
    v.folio_venta,
    v.fecha_venta,
    v.monto_total,

    -- Customer data
    c.id_cliente,
    c.nombre AS nombre_cliente,
    c.ap_patcliente,
    c.ap_matcliente,
    c.rfc AS rfc_cliente,
    c.estado AS estado_cliente,
    c.codigo_postal AS cp_cliente,
```

```
    c.colonia AS colonia_cliente,
    c.calle AS calle_cliente,
    c.numero AS numero_cliente,
    cc.email AS email_cliente,

    -- Item details
    dv.id_detalleventa,
    dv.codigo_barras,
    p.nombre AS nombre_producto,
    p.marca AS marca_producto,
    p.descripcion AS descripcion_producto,
    dv.cantidad,
    dv.precio_aplicadou AS precio_unitario,
    dv.sumtotal AS subtotal_linea

FROM venta v
JOIN cliente c
    ON v.id_cliente = c.id_cliente
LEFT JOIN correo_cliente cc
    ON cc.id_cliente = c.id_cliente
JOIN detalle_venta dv
    ON dv.folio_venta = v.folio_venta
JOIN inventario i
    ON dv.codigo_barras = i.codigo_barras
JOIN producto p
    ON i.id_producto = p.id_producto;
```

To query the "invoice" corresponding to a specific folio, it is enough to filter the view, for example:

```
SELECT *
FROM vista_factura
WHERE folio_venta = 'VENT-005'
ORDER BY id_detalleventa;
```

This query returns all detail lines associated with that folio, along with the customer and sale data, allowing the complete invoice to be viewed.

Revenue Queries

Based on the information recorded in `venta`, `detalle_venta` and `inventario`, queries were defined that allow obtaining the total sold and the profit generated in different time periods.

Total Sold and Profit on a Specific Date

```
CREATE OR REPLACE FUNCTION ventaTotalFecha (fechaConsulta DATE)
RETURNS TABLE(fecha date,
```

```
             total_vendido numeric,
             ganancia numeric
            )
AS $$
BEGIN

    RETURN QUERY
    SELECT DATE(v.fecha_venta) AS fecha,
           SUM(dv.sumtotal) AS total_vendido,
           SUM( (dv.precio_aplicadou - i.precio_compra) * dv.cantidad ) AS ganancia
    FROM venta v
    JOIN detalle_venta dv
        ON dv.folio_venta = v.folio_venta
    JOIN inventario i
        ON dv.codigo_barras = i.codigo_barras
    WHERE DATE(v.fecha_venta) = fechaConsulta
    GROUP BY DATE(v.fecha_venta);

END;
$$ LANGUAGE plpgsql;

-- To call the function, we use the command:
SELECT * FROM ventaTotalFecha('2025-11-27');
-- The date must have the format YYYY-MM-DD.
```

Total Sold and Profit Between Two Dates

```
CREATE OR REPLACE FUNCTION ventaTotalEntreFechas (fechaInicioConsulta DATE,
                                                  fechaFinalConsulta DATE DEFAULT NULL)
RETURNS TABLE(fecha date,
             total_vendido numeric,
             ganancia numeric
            )
AS $$
BEGIN

    -- If the final date is NULL, the current date is assigned
    fechaFinalConsulta := COALESCE(fechaFinalConsulta, now());

    RETURN QUERY
    SELECT DATE(v.fecha_venta) AS fecha,
           SUM(dv.sumtotal) AS total_vendido,
           SUM( (dv.precio_aplicadou - i.precio_compra) * dv.cantidad ) AS ganancia
    FROM venta v
    JOIN detalle_venta dv
        ON dv.folio_venta = v.folio_venta
    JOIN inventario i
```

```
        ON dv.codigo_barras = i.codigo_barras
    WHERE DATE(v.fecha_venta)
        BETWEEN fechaInicioConsulta AND fechaFinalConsulta
    GROUP BY DATE(v.fecha_venta)
    ORDER BY fecha;

END;
$$ LANGUAGE plpgsql;

-- Examples of how to call the function:
SELECT * FROM ventaTotalEntreFechas('2025-11-01', '2025-12-08');
SELECT * FROM ventaTotalEntreFechas('2025-11-01');
-- In both cases, dates must use the format YYYY-MM-DD.
```

Total Historical Profit

```
CREATE OR REPLACE FUNCTION gananciaTotal ()
RETURNS TABLE(ganancia numeric)
AS $$
BEGIN

    RETURN QUERY
    SELECT
        SUM( (dv.precio_aplicadou - i.precio_compra) * dv.cantidad ) AS ganancia_total
    FROM detalle_venta dv
    JOIN inventario i
        ON dv.codigo_barras = i.codigo_barras;

END;
$$ LANGUAGE plpgsql;

-- To call the function, we use the command:
SELECT * FROM gananciaTotal();
```

These queries make it possible to determine the revenue generated and the profit obtained by the business, fulfilling the requirements of the project.

6.4 Special Business Functions

The following functions are intended to obtain key information for decision-making in the business. In particular, they make it possible to determine the profit generated by a product based on its barcode, as well as to identify items with low inventory levels together with their supplier information.

Profit per Product This function calculates the profit generated by a specific product using its barcode. To do this, it relates information from the inventory, the product, and the sale details, and determines the total profit considering the applied sale price and the purchase cost.

```
CREATE OR REPLACE FUNCTION utilidadPorProducto (codigoB varchar)
RETURNS TABLE(codigo_barras varchar,
              nombre varchar,
              cantidad_vendido integer,
              utilidad numeric
             )
AS $$
BEGIN

    RETURN QUERY
    SELECT DV.codigo_barras,
           PR.nombre,
           DV.cantidad,
           -- Operation that calculates the product profit
           SUM( (DV.precio_aplicadou - I.precio_compra) * DV.cantidad )
    FROM inventario I
    INNER JOIN producto PR
        ON I.id_producto = PR.id_producto
    INNER JOIN detalle_venta DV
        ON I.codigo_barras = DV.codigo_barras
    WHERE I.codigo_barras = codigoB
    GROUP BY (DV.codigo_barras,
           PR.nombre,
           DV.cantidad);

END;
$$ LANGUAGE plpgsql;

-- The function is called with:
SELECT * FROM utilidadPorProducto('1704175696641');
```

Products with Low Stock This function returns all products whose inventory level is equal to or less than three units. It allows displaying the barcode, product name, purchase date, current stock, and the supplier's contact information. It is useful to anticipate purchases and avoid stockouts.

```
CREATE FUNCTION stockBajo ()
RETURNS TABLE(codigo_barras varchar,
              nombre varchar,
              fecha_compra date,
              stock_actual integer,
              proveedor varchar,
              movil varchar,
              oficina varchar,
              numero_telefono varchar
              )
AS $$
```

```
BEGIN

    RETURN QUERY
    SELECT I.codigo_barras,
           PR.nombre,
           I.fecha_compra,
           I.stock_actual,
           P.nombre,
           T.movil,
           T.oficina,
           T.numero_telefono
    FROM inventario I
    INNER JOIN proveedor P
        ON I.id_proveedor = P.id_proveedor
    INNER JOIN telefonos T
        ON I.id_proveedor = T.id_proveedor
    INNER JOIN producto PR
        ON I.id_producto = PR.id_producto
    WHERE I.stock_actual <= 3;

END;
$$
LANGUAGE plpgsql;

-- The function is called without parameters:
SELECT * FROM stockBajo();
```

6.5 Indexes

Two indexes were created; in both cases a **B-Tree** index was chosen. This is a balanced tree that allows locating records without scanning the entire table. This type of tree seeks to maintain the smallest possible number of levels, thus reducing access time compared to trees with a larger number of levels. The B-Tree maintains data in sorted order, so searches are performed with time complexity **O(log n)**, which is ideal for the cases where it was used.

- In the first case, an index was created in the `inventario` table on the `stock_actual` column with the purpose of enabling fast searches based on available stock. For example, if the user wanted to know which products are closest to being sold out, using `ORDER BY` with an index would be much quicker, especially with a large inventory.
- In the second case, an index was also created in the `inventario` table, but on the `id_proveedor` column, with the purpose of optimizing join queries between inventory and supplier. For example, if the user wanted to know which products are provided by which providers, a join operation would be faster because it searches through both tables for matching values.

6.6 Triggers and functions for the inventory

The aim of this section is to automate the management of the inventory and the sales totals, in accordance with the requirements of the specification:

- Subtract from the stock the quantity sold of each product.
- Abort the transaction when the stock reaches zero or is insufficient.
- Generate an alert when the stock of a product is less than 3 units.
- Update the total amount to be paid per item and the total amount of the entire sale.

To achieve this, two triggers were defined on the `detalle_venta` table, supported by functions written in `PL/pgSQL`.

BEFORE trigger on `detalle_venta`

The first trigger, `trg_detalle_venta_before`, is executed *before* inserting a row into the `detalle_venta` table. This trigger invokes the function `fn_detalle_venta_before()`, which performs the following actions:

1. It computes the total per item (`sumtotal`) by multiplying the quantity sold by the applied price:
$$\texttt{sumtotal} = \texttt{cantidad} \times \texttt{precio\_aplicadou}.$$
2. It queries the value of `stock_actual` in the `inventario` table using the product's `codigo_barras`.
3. If there is no inventory record for that barcode, an exception is raised and the operation is cancelled.
4. If `stock_actual` is less than or equal to zero, an exception is raised indicating that there is no stock available.
5. If the requested quantity is greater than the available stock, a "stock insufficient" exception is raised.
6. Finally, if the available stock minus the quantity sold would be exactly zero, the transaction is aborted in order to comply with the business rule stating that the sale must be cancelled when the stock reaches zero.

With this trigger we guarantee that:

- The total per item is computed automatically.
- Sales are never recorded with null or insufficient stock.

AFTER trigger on `detalle_venta`

The second trigger, `trg_detalle_venta_after`, is executed *after* inserting a row into the `detalle_venta` table. This trigger calls the function `fn_detalle_venta_after()`, which is responsible for:

1. Subtracting from the inventory the quantity sold, updating the `stock_actual` field of the `inventario` table.
2. Retrieving the new value of `stock_actual`. If the result is less than 3, a message is issued via `RAISE NOTICE`, acting as a low-stock alert.
3. Recomputing the total amount of the complete sale. To do so, all the `sumtotal` values of the rows associated with the same `folio_venta` in the `detalle_venta` table are summed, and this result is stored in the `monto_total` field of the `venta` table.

This second trigger allows us to:

- Keep our inventory synchronized with the sales made.
- Detect at an early stage those products for which only a very small number of units remain in stock.
- Ensure that the total of the sale (`monto_total`) is always up to date, even when several items are added to the same sale.

Taken together, both triggers automatically implement the previously established business rules that we use for managing and controlling our inventory, as well as for computing the amounts in the sales system.

7 Personal Conclusions

1. **De la Peña Osorio Lilian:** In my opinion, this project represented an opportunity to apply the concepts studied during the course. From the design of the Relational Model and the Entity–Relationship Model, I was able to understand how to clearly represent the key entities of the business (customers, suppliers, products, inventory, and sales) and their respective relationships. Taking into account composite attributes such as the address, and deciding how to decompose them in the relational model, helped me better understand the importance of normalization and sound conceptual design before writing a single line of SQL code.

   The development of the table creation script in PostgreSQL was a valuable exercise for translating the logical design into a concrete implementation, paying attention to details such as primary keys, foreign keys, `NOT NULL`, `UNIQUE` and `CHECK` constraints, as well as the format of the sale folio `VENT-001`. Finally, building the invoice-like view and the revenue queries allowed me to see how the database not only stores information but also facilitates the generation of useful reports for the business, such as the total sold per date or the profit obtained in a period. Likewise, with the help of the team, it was easier to complement each part of the project, achieving a favorable result that fits the requested requirements.

2. **Maldonado Jr. Montoya Gustavo:** In this project I was able to observe and confirm the importance of everything covered during the semester, from the Entity–Relationship Model to its application in SQL coding. Now that we have reached this point, I can say that the ER model is the most important and first thing we must review and create for a database, since it is the basis of the project structure. With it we can understand the relationships between tables and their behavior, we can identify the attributes that make up the entities, their roles (whether they are primary keys, foreign keys, whether they can contain `NULL` values or must contain something `NOT NULL`, whether they are calculated values or multivalued attributes), and which of them are composite. With this, we can form a more specific idea to obtain the relational model, which is already very similar to the tables we use in DDL.

   Although the ER model is very important, so are the functions and triggers that we use to query and maintain information consistency. As we saw in the project, there may be cases where a user requests more items than we have registered in inventory, and in that case we cannot offer something we do not have; therefore, a trigger is used to warn about this problem.

   In our project we were only able to insert a limited amount of data, and I was able to see that it worked well. However, I would like to see in this or another project how a query behaves with thousands of records: its response time, possible errors that may occur, and how those queries can be improved and optimized, etc.

3. **Hernández Ledesma Emiliano:** Throughout this project, I deepened my understanding of how essential proper database design is for building reliable and functional systems. Starting from the Entity–Relationship Model, I learned the importance of correctly identifying entities, attributes, and relationships so that the logical model truly reflects real-world processes. When translating this design into SQL, I strengthened my ability to work with constraints, foreign keys, triggers, and validation mechanisms that preserve data integrity.

   Developing functions and views, especially those related to invoice generation and revenue/profit queries, showed me how SQL can transform raw data into useful information for business decision-making. In addition, collaborating with my teammates enriched the experience, as each person contributed perspectives and knowledge that helped improve the final result. This project reaffirmed for me the relevance and power of database systems in solving practical, real-world problems.

4. **Ramírez Peña Luis Enrique:** The development of this project represents an integral exercise that allows us to apply, in a practical manner, the knowledge acquired throughout the course, confronting a realistic problem commonly found in the workplace. As in Software Engineering, the first step involved thoroughly understanding the business, identifying its needs, and gathering the requirements based on the information provided. From there, as a team, we were able to define tasks—some sequential and others parallel—that progressively led us toward a functional solution.

   This experience is valuable because it closely simulates the dynamics of a professional environment, where proper database design is essential to ensure consistency and the successful development of the project. From the creation of the ERD and relational model to the implementation in code, each stage required analysis and justification of the decisions made. Overall, this project strengthened our ability to propose technical solutions to real problems, especially considering that, in many cases, clients or employers do not have a technical

background and rely on our ability to translate their needs into efficient and well-designed systems.

5. **Moreno Vázquez Esteban:** During this project we were able to correctly develop a management system based on the analysis of the proposed problem. To achieve this, we used the knowledge acquired throughout the semester, from the first topic to the last. Personally, the exercises and assignments completed during the semester, along with the laboratory practices, made the transition from theoretical concepts to practical application easier. Even so, there is a big difference between doing an exercise and designing a project with real usefulness, since many details must be taken into account for it to work properly.

   I believe that teamwork was fundamental, as it made it easier to complete many tasks, completing this project individually would have taken much longer. At the same time, there were certain challenges, such as integrating everyone's work, because we had to be careful to ensure everything functioned correctly. It was necessary to understand what my teammates had done in order to move forward. In this sense, having clear work documentation was very important, as it made it easier to unify everything.

6. **Calzada Lozada Sergio Armando:** Throughout the development of this project, we applied everything we learned during the semester. When defining our database, we identified each of the entities and the attributes associated with each one. The part I worked on was the MR model using the ER Studio tool; this model is normalized and meets all the required characteristics. Each member of the team had to complete their part of the project successfully, and the modeling depended on each team member, as each part relied on another. Teamwork was very important in completing the project. Each of my teammates carried out their assigned activities correctly, the objectives were successfully achieved, and the knowledge we gained was properly applied in every part of the project.

7. **Nunez Badillo Armando Adair:** In this project we developed an inventory and sales management system on top of PostgreSQL using `PL/pgSQL`. A key design choice was to enforce the main business rules (stock control, validation of sales, and total calculations) directly in the database layer, so that the system remains consistent regardless of how the client application is implemented.

   My main contribution was the implementation of the triggers and their supporting functions on the `detalle_venta` table. Through the `BEFORE` and `AFTER` triggers, I worked on validating that there is enough stock before each sale, calculating the total per item, updating the inventory automatically, and raising alerts when stock is running low. This work helped me better understand how to model business rules inside the database, how `BEFORE/AFTER` triggers behave, and how to use exceptions and transactions to protect data integrity in a real system.