UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

Proyecto final
Bases de datos

Semestre 2025-2

Nava Benítez David Emilio
Tavera Castillo David
Medina Guzmán Santiago
Santiago Estrada Samantha
Rodríguez Zacarías Iván
Jiménez Elizalde Josué

Profesor:
Arreola Franco Fernando
Grupo: 1

26 de mayo de 2025

Índice

Índice de figuras

1. Introducción

En el entorno empresarial contemporáneo, la transformación digital ha dejado de ser una ventaja competitiva para convertirse en una necesidad estratégica. Esta transición es especialmente crucial para las pequeñas y medianas empresas que buscan optimizar sus procesos, centralizar su información y ofrecer un mejor servicio a sus clientes. En este contexto, el presente proyecto tiene como propósito desarrollar una solución tecnológica orientada a la digitalización de los procesos internos de una mueblería, particularmente en lo relacionado con el control de inventario, ventas, proveedores, clientes y recursos humanos.

La problemática que se aborda radica en el manejo manual y físico de registros que, además de ser propenso a errores, dificulta la visibilidad global de la operación de la empresa. La ausencia de una base de datos centralizada y estructurada limita la capacidad de análisis, seguimiento y toma de decisiones oportunas. Por tanto, se plantea como objetivo general diseñar e implementar una base de datos relacional en PostgreSQL que permita almacenar, gestionar y consultar de manera eficiente la información crítica del negocio, asegurando su integridad, consistencia y disponibilidad.

La solución se construirá a partir del análisis detallado de los requerimientos funcionales proporcionados, abarcando aspectos como la gestión de artículos y sus proveedores, la trazabilidad de las ventas, la generación automatizada de tickets y reportes, así como la administración de los empleados y las sucursales. Además, se contempla la incorporación de mecanismos de seguridad y validación mediante la implementación de restricciones, vistas, funciones, triggers y procedimientos almacenados. Estos elementos permitirán automatizar tareas, validar operaciones y mantener la coherencia de los datos.

Desde el punto de vista académico, este proyecto representa una oportunidad para aplicar de manera práctica los conocimientos adquiridos durante el curso de Bases de Datos, incluyendo el diseño conceptual, lógico y físico, así como el uso de lenguaje SQL en su vertiente de definición y manipulación de datos. Se busca, además, fomentar el trabajo en equipo, la organización de actividades y la documentación formal del proceso de desarrollo, siguiendo principios de la administración de proyectos de software adoptado por nosotros a traves de iniciativa grupal.

2. Plan de Trabajo

2.1. Acta Constitutiva

2.1.1. Información general del proyecto:

Nombre del Proyecto	Sistema de Gestión de Operaciones para Mueblería
Project Manager	David Tavera
Correo Electrónico	—
Teléfono	+52 123 456 7890
Unidades Organizativas	Departamento de Bases de Datos
Roles y Responsables	▪ **Analista de Requerimientos:** David Nava ▪ **Diseñador de Base de Datos:** Samantha Santiago ▪ **Desarrollador de Base:** David Tavera ▪ **Desarrollador Backend:** Ivan Rodriguez ▪ **Desarrollador Frontend:** Josué Jiménez ▪ **Tester:** Santiago Medina
Fecha Prevista de Inicio	21 de abril de 2025
Fecha Prevista de Finalización	24 de mayo de 2025

2.1.2. Propósito del proyecto:

Diseñar e implementar una base de datos relacional que permita digitalizar la operación de una mueblería, centralizando la información de artículos, ventas, empleados, clientes y sucursales para mejorar la eficiencia operativa y la toma de decisiones. Se incluye un dashboard visual que permita el análisis de los ingresos mensuales y otros

indicadores de negocio.

2.1.3. Alcance:

El alcance de este proyecto incluye el diseño e implementación de una base de datos en PostgreSQL para una mueblería, con el fin de digitalizar sus operaciones comerciales. Esto abarca la gestión de artículos, proveedores, clientes, empleados, ventas y sucursales, así como la creación de procedimientos automáticos, restricciones de integridad y un dashboard visual que muestre indicadores clave, tales como ingresos mensuales por sucursal.

2.1.4. Entregables principales:

- Modelo entidad-relación.
- Modelo Relacional.
- Script SQL con la creación de la base de datos y su correspondiente tabla.
- Script para el agregado de información.
- Script de la información a nivel Base de Datos.
- Dashboard.
- Presentación para la exposición del proyecto.
- Documento con justificación técnica, evidencia de funcionamiento y conclusiones (el presente documento).

2.1.5. Supuestos:

El sistema se centrará únicamente en la gestión de información y los datos de prueba serán proporcionados o creados por el equipo de desarrollo.

2.1.6. Restricciones:

- Se usará Draw.io y SQL Developer para el diseño conceptual y lógico, respectivamente, de la base de datos.
- Se debe usar PostgreSQL como sistema gestor.
- El desarrollo debe completarse en un periodo de 5 semanas.

Firmas de Aprobación

Nombre	**Rol**	**Firma electrónica**
David Tavera	Project Manager	David Tavera
David Nava	Analista de Requerimientos	David Nava
Samantha Santiago	Diseñadora de Base de Datos	Samantha Santiago
Ivan Rodriguez	Desarrollador Backend	Ivan Rodriguez
Josué Jiménez	Desarrollador Frontend	Josué Jiménez
Santiago Medina	Tester	Santiago Medina

Fecha de aprobación: [22/04/2025]

2.2. Plan de Actividades

Actividad	**Descripción**	**Predecesor**	**Duración [Horas]**
A1	Levantamiento de requerimientos	-	4
A2	Diseño conceptual de la base de datos	A1	2
A3	Diseño lógico de la base de datos	A2	3
A4	Revisión del diseño y reglas de negocio	A2, A3	2
A5	Reunión para aprobación de diseño	A1, A2, A3, A4	2
A6	Definición de estructura de la base de datos	A5	4
A7	Desarrollo de Inserts	A6	2
A8	Reunión de avance	Todas las anteriores	2
A9	Desarrollo de funciones	A8	12
A10	Desarrollo de Triggers	A8	4
A11	Reunión de avance	Todas las anteriores	2
A12	Revisión de detalles y correcciones	A11	12
A13	Reunión para aprobación	A12	3
A14	Creación del Dashboard	A13	6
A15	Desarrollo Frontend	A13	15
A16	Testing	A15	8
A17	Revisión y corrección de errores encontrados en fase previa	A8	2
A18	Reunión para aprobación final del sistema	A8	4
B1	Realización de entregables y documento final	-	Documental

2.3. Asignación de Responsabilidades

Nombre	**Rol**	**Actividad asignada**
David Tavera	Project Manager	A6, A9, A10 y Reuniones
David Nava	Analista de requerimientos	A1, A2, A3, B1
Samantha Santiago	Diseñadora	A2, A3
Ivan Rodriguez	Desarrollador	A6, A7, A10
Josue Jimenez	Desarrollador	A3, A15, B1
Santiago Medina	Tester	A14, A16, B1

Explicitamente, todos contribuyen en la actividad *B1* al realizar cada uno de los entregables previamente enlistados; así como contribuir en las reuniones de avance, aprobación y corrección.

Cabe destacar que esta asignación de responsabilidades no exime el hecho de realizar trabajo más allá de sus encargos aqui asignados. Por lo tanto el equipo completo estará involucrado en todas las fases sin importar su asignación, pero teniendo en cuenta que su responsabilidad principal es la aqui señalada.

3. Diseño

El diseño de la base de datos se estructuró en diversas fases, comenzando con la identificación de entidades y relaciones a partir de los requerimientos funcionales proporcionados. Posteriormente, se elaboraron modelos conceptuales, lógicos y físicos, los cuales se describen a continuación.

3.1. Análisis de Requerimientos

El análisis de requerimientos tuvo como objetivo identificar las entidades, atributos y relaciones clave a partir de la descripción funcional proporcionada para la digitalización de los procesos de una mueblería. A través de esta etapa, se examinaron cuidadosamente las necesidades del sistema, considerando tanto la gestión operativa (ventas, stock, empleados, proveedores) como los aspectos administrativos (clientes, facturación, sucursales).

Se realizó una lectura detallada de los requerimientos funcionales, permitiendo abstraer los siguientes elementos principales:

- Entidades centrales como **Artículo**, **Venta**, **Cliente**, **Empleado**, **Proveedor** y **Sucursal**.
- Atributos esenciales para cada entidad, incluyendo datos como códigos de identificación, información fiscal, datos personales, precios, fechas y cantidades.
- Relaciones relevantes, tales como la asociación entre artículos y proveedores , ventas realizadas por cajeros y vendedores, así como la jerarquía organizacional de los empleados.
- Reglas de negocio explícitas, como la validación de existencia de stock antes de concretar una venta, o la restricción de que los empleados solo trabajen en una sucursal.

De la misma forma, se extraen los requerimientos funcionales y no funcionales que se tendrá que implementar principalmente dentro del diseño físico; tales como:

1. Actualización de totales de articulos para una articulo, una venta y la cantidad de articulos a partir de haberse realizado una venta.
2. Validación de disponibilidad.
3. Lista de articulos no disponibles y/o con stock menor a tres.

4. Generación de ticket de venta incluyendo su folio para facuración.

5. Validación de permisos para el sistema (vendedor y cajero) así como pertenencia a la misma sucursal.

6. Desplegar la jerarquia organizacional al realizar una busqueda a partir de un empleado.

Este análisis sirve como base para la construcción del modelo entidad-relación (MER) y el posterior modelo relacional (MR), asegurando que la base de datos represente fielmente la operación del negocio y permita la integridad, consistencia y trazabilidad de la información.

3.2. Diseño conceptual

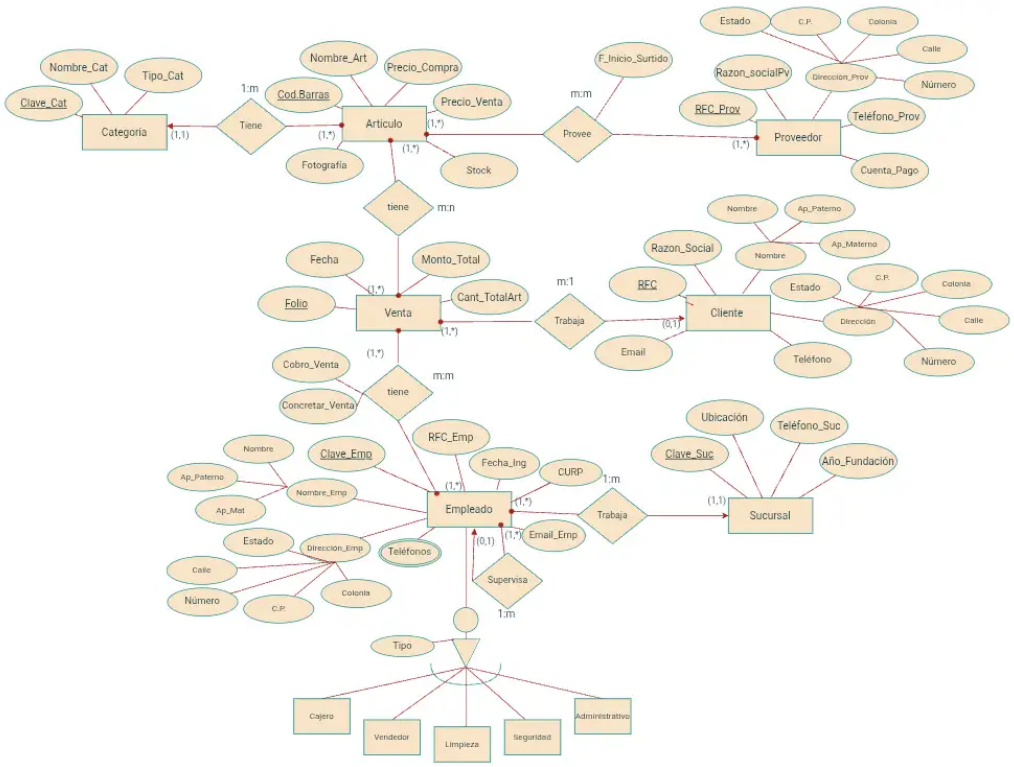

Figura 1: MER

Siguiendo los requerimientos ofrecidos por parte del cliente y haciendo un análisis exhaustivo para determinar las reglas de negocio y no omitir partes fundamentales de la base de datos; se diseñó a tráves de Draw.io el modelo entidad - relación como un primer paso hacia el cumplimiento de los requisitos solicitados.
Entidades:

- Empleado
- Sucursal
- Venta
- Cliente
- Proveedor
- Artículo
- Categoría

Mientras que las siguientes entidades se identificaron a partir de una generalización en consecuencia de los requerimientos y para un modelado eficaz:

- Cajero
- Vendedor
- Limpieza
- Seguridad
- Administrativo

Cada una de estas entidades estan relacionados con una u otra, determinadas a partir de los requerimientos.
Se presenta el modelo entidad - relación en la Figura 1.

Se tienen las siguientes relaciones entre entidades:

- Categoría con Artículo (1:M)
- Artículo con Venta (M:M)
- Venta con Cliente (M:1)
- Empleado con Sucursal (1:M)
- Artículo con Proveedor (M:M)

Se modeló una relación recusiva en la entidad Empleado para el supervisor directo de cada empleado:

- Empleado con Empleado (Supervisor) (1:M)

Finalmente, destacar que se usa una exclusión en la generalización para limitar el rol del Empleado que debe pertenecer a uno y solo uno de ellos.

3.3. Diseño Lógico

Durante este diseño, usando la herramienta SQL Developer para la construccion del Modelo Relacional, tomamos como base la fase anterior para poder transformar el MER obtenido a un MR que satisfaga con los requerimientos. Por lo que se usaron técnicas para poder realizar correctamente la transformación.

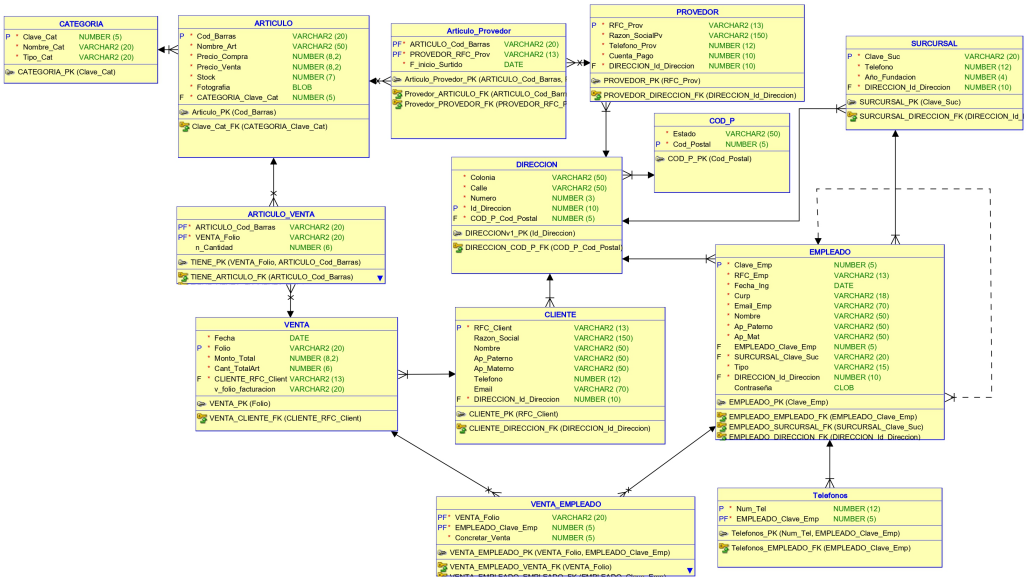

Figura 2: MR

Como se puede observar en la Figura 2 se representan cada una de las entidades modeladas en la Figura 1 tranformadas a relaciones con sus correspodientes atributos y definiendo el dominio de todas y cada una de las relaciones. Así también, se define la llave primaria (PK) guiandonos en las claves principales de las entidades fuertes modeladas en el Modelo Entidad - Relación.

Para las relaciones M:M contenidas en la Figura 1 se hizo su correcta transformación a una relación intermedia entre las entidades, satisfaciendo que la nueva relación tendrá como PK las PK's de las en-

tidades que une (que a su vez son Foreign Key), más los atributos que la relación tenga. Por esta situación, se tienen las siguientes Relaciones:

- ARTICULO_VENTA
- ARTICULO_PROVEEDOR
- VENTA_EMPLEADO

Que satisfacen los requerimientos de la transformación.

Para el caso del atributo multivaluado que se pueden identificar en la Figura 1 y que es parte de la entidad EMPLEADO, se crea una nueva relación. A nivel general, la PK de la nueva tabla será una PK compuesta formada por la PK de la tabla, en este caso EMPLEADO y por el atributo multivalorado, en este caso, teléfono.

En un principio, teniamos una transformación parecida a la del atributo multivaluado anterior para Direccion (atributo compuesto en la Figura 1) en cada una de las tablas siguientes: EMPLEADO, CLIENTE y PROVEEDOR. Esto no fue eficiente por lo que se optó por una sola tabla llamada DIRECCION y que este propagada como FK en cada una de las tablas ya antes mencionadas.

Dentro de nuestro proceso de tranformación, al realizAr la normalización correspondiente nos percatamos de un grupo de repetición en el codigo postal, ya que diferentes codigos postales iban a tener el mismo estado, por lo que la tabla DIRECCION no cumplia con la primera forma normal (1FN). Para evitar este problema se creo una nueva tabla llamada COD_P que se va a propagar como FK hacia DIRECCION.

Para la transformación de la entidad EMPLEADO, se realizo los mismos pasos para la entidades anteriores pero al tener una relación recursiva, que se puede observar en la Figura 1, para el Supervisor directo (que a fin de cuentas es un empleado), se propago la PK de

empleado como un FK cuidando que no puede haber atributos que se llamen igual.

Finalmente, para resolver la generalización y la exclusión de esta misma, se optó por un atributo llamado *Tipo* que funciona para saber el tipo de empleado:

- Cajero
- Vendedor
- Limpieza
- Seguridad
- Administrativo

3.4. Diseño Físico

Para esta fase del diseño, tomamos ya todo lo que hemos realizados en las anteriores fases para proceder a implementarlo en PostgreSQL. Con ello se realizaron los scripts correspondientes para definir las relaciones provenientes del MR asi como sus atributos. Así también, se establecieron los Constraints pertinentes para poder modelar lo realizado en el MR.

3.4.1. Convenciones generales en el diseño físico

En el proceso de diseño físico de la base de datos, se han adoptado varias convenciones técnicas con el objetivo de asegurar consistencia, eficiencia y precisión en el manejo de los datos. Estas convenciones son aplicadas transversalmente a todas las tablas del sistema, y su comprensión evita la redundancia en las explicaciones posteriores.

- **Tipos de datos alfanuméricos (`varchar`)**: Se utiliza el tipo `character varying` con un límite de longitud específico para campos como nombres, códigos o descripciones. Esto permite almacenar cadenas de longitud variable, optimizando espacio y permitiendo la inclusión de datos alfanuméricos como códigos de barras o claves externas.
- **Valores monetarios (`numeric(8,2)`)**: Para representar precios o montos financieros, se emplea el tipo `numeric(8,2)`, que ofrece una alta precisión al evitar errores de redondeo asociados con los tipos `float` o `real`. Esta elección es clave en aplicaciones donde la exactitud financiera es prioritaria.
- **Cantidad en inventario o conteos (`numeric` sin decimales)**: Para campos que representan unidades físicas o conteos (como el stock disponible), se opta por tipos numéricos enteros (`numeric(x,0)`), ya que no se manejan fracciones en este tipo de información.

- **Fecha** (`DATE`): Se utiliza el tipo `DATE` para almacenar fechas sin información de hora, adecuado para campos como fechas de nacimiento, fechas de registro o fechas de eventos. Este tipo permite realizar operaciones y comparaciones temporales de forma eficiente.
- **Texto con colación específica** (`TEXT COLLATE`): El tipo `TEXT` permite almacenar cadenas de longitud variable sin límite predefinido. La cláusula `COLLATE` especifica la colación a usar para comparar y ordenar texto, en este caso `pg_catalog."default"`, que es la colación predeterminada del sistema. Esto asegura que las operaciones sobre el texto sigan reglas específicas de ordenamiento y comparación según la configuración regional.
- **Claves primarias** (`PRIMARY KEY`): Toda tabla relevante en el sistema contiene al menos una clave primaria, definida generalmente sobre un campo único, no nulo, que garantiza la identificación unívoca de cada registro. Esta restricción es fundamental para mantener la integridad interna de cada entidad.
- **Integridad referencial y claves foráneas** (`FOREIGN KEY`): Las relaciones entre tablas se establecen mediante claves foráneas que apuntan a claves primarias de otras tablas. Estas restricciones garantizan que los datos estén relacionados correctamente y evitan registros huérfanos.
- **Acciones referenciales** (`ON DELETE / ON UPDATE`):
  - Se utiliza `ON UPDATE NO ACTION` para impedir la modificación de claves primarias referenciadas.
  - En algunos casos, se define `ON DELETE NO ACTION` para evitar eliminaciones que afecten integridad.

- Cuando se requiere eliminar en cascada los registros dependientes, se emplea `ON DELETE CASCADE`, facilitando el mantenimiento automático de la integridad entre entidades relacionadas.

- **Imágenes o datos binarios** (`bytea`): En los casos donde es necesario almacenar imágenes o archivos directamente en la base de datos (como fotografías de productos), se emplea el tipo `bytea`, que permite guardar datos binarios. Aunque en sistemas productivos puede ser más eficiente almacenar solo la ruta del archivo, para este proyecto se opta por esta solución para mantener independencia del sistema de archivos externo.

- **Restricciones de nulidad** (`NOT NULL`): Se aplican de forma estricta a los campos que representan datos obligatorios, lo cual previene la inserción de registros incompletos y refuerza la integridad semántica de la base de datos.

Estas convenciones generales se aplican como principios base para todas las tablas del sistema.

3.4.2. Creación de Tablas

En esta fase, se define la estructura de las tablas que conformaran la definición de nuestra base de datos. De manera general, se definen los atributos y tipos de datos de cada una de las tablas, asi como sus restricciones y consideraciones adicionales a tráves de PostgreSQL.
Tabla ARTICULO:

```
1     CREATE TABLE IF NOT EXISTS public.articulo
2 (
3     v_cod_barras character varying(20) COLLATE pg_catalog."
        default" NOT NULL,
4     v_nombre_art character varying(50) COLLATE pg_catalog."
        default" NOT NULL,
```

```
    n_precio_compra numeric(8,2) NOT NULL,
    n_precio_venta numeric(8,2) NOT NULL,
    n_stock numeric(7,0) NOT NULL,
    b_fotografia bytea NOT NULL,
    n_categoria_clave_cat numeric(5,0) NOT NULL,
    CONSTRAINT articulo_pk PRIMARY KEY (v_cod_barras),
    CONSTRAINT clave_cat_fk FOREIGN KEY (n_categoria_clave_cat
       )
        REFERENCES public.categoria (n_clave_cat) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla anterior almacena los datos básicos de los productos de la mueblería, identificados de forma única por su código de barras (`v_cod_barras`). Incluye atributos para nombre, precios de compra y venta, stock disponible y fotografía del artículo. Además, establece una relación con la tabla **categoria** a través de la clave foránea `n_categoria_clave_cat`, garantizando que cada artículo pertenezca a una categoría válida. Se definen restricciones para mantener la integridad de los datos, como la clave primaria y la restricción referencial con acciones `NO ACTION` para actualización y eliminación.

Tabla ARTICULO_PROVEEDOR:

```
    CREATE TABLE IF NOT EXISTS public.articulo_provedor
(
    v_articulo_cod_barras character varying(20) COLLATE
       pg_catalog."default" NOT NULL,
    v_provedor_rfc_prov character varying(13) COLLATE
       pg_catalog."default" NOT NULL,
    d_f_inicio_surtido date NOT NULL,
    CONSTRAINT articulo_provedor_pk PRIMARY KEY (
       v_articulo_cod_barras, v_provedor_rfc_prov),
    CONSTRAINT provedor_articulo_fk FOREIGN KEY (
       v_articulo_cod_barras)
        REFERENCES public.articulo (v_cod_barras) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT provedor_provedor_fk FOREIGN KEY (
       v_provedor_rfc_prov)
        REFERENCES public.provedor (v_rfc_prov) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE
)
```

La tabla representa la relación muchos a muchos entre los artículos y sus proveedores, identificada por la clave primaria compuesta por el código de barras del artículo y el RFC del proveedor. Incluye además la fecha de inicio del suministro (`d_f_inicio_surtido`). Las claves foráneas garantizan la integridad referencial con las tablas **articulo** y **provedor**.

Tabla ARTICULO_VENTA:

```
    CREATE TABLE IF NOT EXISTS public.articulo_venta
(
    v_articulo_cod_barras character varying(20) COLLATE
       pg_catalog."default" NOT NULL,
    v_venta_folio character varying(20) COLLATE pg_catalog."
       default" NOT NULL,
    n_cantidad numeric(6,0) NOT NULL,
    CONSTRAINT tiene_articulo_fk FOREIGN KEY (
       v_articulo_cod_barras)
        REFERENCES public.articulo (v_cod_barras) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT tiene_venta_fk FOREIGN KEY (v_venta_folio)
        REFERENCES public.venta (v_folio) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE
)
```

La tabla modela la relación entre artículos y ventas, especificando la cantidad de cada artículo vendido. Está identificada por las claves foráneas que vinculan con las tablas **articulo** y **venta**, asegurando integridad referencial con `ON DELETE CASCADE`. Tabla CATEGORIA:

```
  CREATE TABLE IF NOT EXISTS public.categoria
(
    n_clave_cat numeric(5,0) NOT NULL,
    v_nombre_cat character varying(20) COLLATE pg_catalog."
       default" NOT NULL,
    v_tipo_cat character varying(20) COLLATE pg_catalog."
       default" NOT NULL,
    CONSTRAINT categoria_pk PRIMARY KEY (n_clave_cat)
)
```

La tabla almacena las categorías para clasificar los artículos. Está identificada por una clave primaria numérica `n_clave_cat`, y contiene el nombre y tipo de categoría, ambos almacenados como cadenas de texto con colación por defecto. Esta tabla facilita la organización y filtrado de productos.

Tabla CLIENTE:

```
    CREATE TABLE IF NOT EXISTS public.cliente
(
    v_rfc_client character varying(13) COLLATE pg_catalog."default" NOT NULL,
    v_razon_social character varying(150) COLLATE pg_catalog."default",
    v_nombre character varying(50) COLLATE pg_catalog."default",
    v_ap_paterno character varying(50) COLLATE pg_catalog."default",
    v_ap_materno character varying(50) COLLATE pg_catalog."default",
    n_telefono numeric(12,0),
    v_email character varying(70) COLLATE pg_catalog."default",
    n_direccion_id_direccion numeric(10,0),
    "t_contrasena" text COLLATE pg_catalog."default",
    CONSTRAINT cliente_pk PRIMARY KEY (v_rfc_client),
    CONSTRAINT cliente_direccion_fk FOREIGN KEY (n_direccion_id_direccion)
        REFERENCES public.direccion (n_id_direccion) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla registra los datos principales de los clientes, identificados por su RFC único como clave primaria. Incluye campos para razón social, nombre y apellidos, teléfono, correo electrónico y una contraseña almacenada como texto con colación por defecto. La dirección se referencia mediante una clave foránea hacia la tabla `direccion`, asegurando integridad referencial sin permitir modificaciones o eliminaciones automáticas.

Tabla COD_P:

```
    CREATE TABLE IF NOT EXISTS public.cod_p
(
    v_estado character varying(50) COLLATE pg_catalog."default
       " NOT NULL,
    n_cod_postal numeric(5,0) NOT NULL,
    CONSTRAINT cod_p_pk PRIMARY KEY (n_cod_postal)
)
```

La tabla almacena códigos postales únicos, identificados por `n_cod_postal` como clave primaria. Cada código postal está asociado a un estado específico, representado como una cadena de hasta 50 caracteres con colación por defecto. Tabla DIRECCION:

```
    CREATE TABLE IF NOT EXISTS public.direccion
(
    v_colonia character varying(50) COLLATE pg_catalog."
       default" NOT NULL,
    v_calle character varying(50) COLLATE pg_catalog."default"
        NOT NULL,
    n_numero numeric(3,0) NOT NULL,
    n_id_direccion numeric(10,0) NOT NULL,
    n_cod_p_cod_postal numeric(5,0) NOT NULL,
    CONSTRAINT direccionv1_pk PRIMARY KEY (n_id_direccion),
    CONSTRAINT direccion_cod_p_fk FOREIGN KEY (
       n_cod_p_cod_postal)
        REFERENCES public.cod_p (n_cod_postal) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla almacena la información detallada de una ubicación física, compuesta por colonia, calle y número, todos obligatorios. Cada dirección está identificada de forma única por `n_id_direccion`, que actúa como clave primaria. Además, establece una relación foránea con la tabla `cod_p`, a través de `n_cod_p_cod_postal`, asegurando que cada dirección esté asociada a un código postal válido. Tabla EMPLEADO:

```
    CREATE TABLE IF NOT EXISTS public.empleado
(
    n_clave_emp numeric(5,0) NOT NULL,
```

```
    v_rfc_emp character varying(13) COLLATE pg_catalog."default" NOT NULL,
    d_fecha_ing date NOT NULL,
    v_curp character varying(18) COLLATE pg_catalog."default" NOT NULL,
    v_email_emp character varying(70) COLLATE pg_catalog."default" NOT NULL,
    v_nombre character varying(50) COLLATE pg_catalog."default" NOT NULL,
    v_ap_paterno character varying(50) COLLATE pg_catalog."default" NOT NULL,
    v_ap_mat character varying(50) COLLATE pg_catalog."default" NOT NULL,
    n_empleado_clave_emp numeric(5,0),
    v_sucursal_clave_suc character varying(20) COLLATE pg_catalog."default" NOT NULL,
    v_tipo character varying(15) COLLATE pg_catalog."default" NOT NULL,
    n_direccion_id_direccion numeric(10,0) NOT NULL,
    "t_contrasena" text COLLATE pg_catalog."default",
    CONSTRAINT empleado_pk PRIMARY KEY (n_clave_emp),
    CONSTRAINT empleado_direccion_fk FOREIGN KEY (n_direccion_id_direccion)
        REFERENCES public.direccion (n_id_direccion) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT empleado_empleado_fk FOREIGN KEY (n_empleado_clave_emp)
        REFERENCES public.empleado (n_clave_emp) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT empleado_sucursal_fk FOREIGN KEY (v_sucursal_clave_suc)
        REFERENCES public.sucursal (v_clave_suc) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla contiene la información esencial del personal de la mueblería, como identifica-

dores fiscales (RFC, CURP), fecha de ingreso y datos personales. La clave primaria es `n_clave_emp`. Se establecen claves foráneas hacia `direccion`, `sucursal` y hacia sí misma (`empleado`), permitiendo registrar jerarquías internas entre empleados. El campo `t_contrasena` se maneja en texto, y se incluye una estructura relacional robusta que garantiza la integridad de las asociaciones laborales. Tabla PROVEEDOR:

```
    CREATE TABLE IF NOT EXISTS public.provedor
(
    v_rfc_prov character varying(13) COLLATE pg_catalog."
       default" NOT NULL,
    v_razon_socialpv character varying(150) COLLATE pg_catalog
       ."default" NOT NULL,
    n_telefono_prov numeric(12,0) NOT NULL,
    n_cuenta_pago numeric(10,0) NOT NULL,
    n_direccion_id_direccion numeric(10,0) NOT NULL,
    "t_contrasena" text COLLATE pg_catalog."default",
    CONSTRAINT provedor_pk PRIMARY KEY (v_rfc_prov),
    CONSTRAINT provedor_direccion_fk FOREIGN KEY (
       n_direccion_id_direccion)
        REFERENCES public.direccion (n_id_direccion) MATCH
           SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla almacena los datos fiscales y de contacto de los proveedores registrados, incluyendo RFC, razón social, número telefónico y cuenta bancaria. Se establece una clave primaria en `v_rfc_prov` para garantizar la unicidad. Además, se relaciona mediante clave foránea con la tabla `direccion`, asegurando la ubicación física del proveedor. El campo `t_contrasena` permite gestionar accesos individuales. Tabla SUCURSAL:

```
    CREATE TABLE IF NOT EXISTS public.sucursal
(
    v_clave_suc character varying(20) COLLATE pg_catalog."
       default" NOT NULL,
    n_telefono numeric(12,0) NOT NULL,
    "n_ano_fundacion" numeric(4,0) NOT NULL,
    n_direccion_id_direccion numeric(10,0) NOT NULL,
    CONSTRAINT sucursal_pk PRIMARY KEY (v_clave_suc),
```

```
    CONSTRAINT sucursal_direccion_fk FOREIGN KEY (
       n_direccion_id_direccion)
        REFERENCES public.direccion (n_id_direccion) MATCH
           SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla contiene la información esencial de cada sede física de la empresa, como su clave identificadora, teléfono de contacto y año de fundación. Se define `v_clave_suc` como clave primaria. La relación con la tabla `direccion` se establece mediante una clave foránea, permitiendo ubicar cada sucursal geográficamente.

Tabla TELEFONOS:

```
    CREATE TABLE IF NOT EXISTS public.telefonos
(
    n_num_tel numeric(12,0) NOT NULL,
    n_empleado_clave_emp numeric(5,0) NOT NULL,
    CONSTRAINT telefonos_pk PRIMARY KEY (n_num_tel,
       n_empleado_clave_emp),
    CONSTRAINT telefonos_empleado_fk FOREIGN KEY (
       n_empleado_clave_emp)
        REFERENCES public.empleado (n_clave_emp) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla permite registrar múltiples números telefónicos asociados a empleados. Se utiliza una clave primaria compuesta por `n_num_tel` y `n_empleado_clave_emp` para garantizar la unicidad de cada número por empleado. La relación con la tabla `empleado` asegura que cada número esté vinculado a un registro válido.

Tabla VENTA:

```
    CREATE TABLE IF NOT EXISTS public.venta
(
    d_fecha date NOT NULL,
    v_folio character varying(20) COLLATE pg_catalog."default"
        NOT NULL,
    n_monto_total numeric(8,2) NOT NULL,
    n_cant_totalart numeric(6,0) NOT NULL,
```

```
    v_cliente_rfc_client character varying(13) COLLATE
       pg_catalog."default" NOT NULL,
    v_folio_facturacion character varying(20) COLLATE
       pg_catalog."default",
    CONSTRAINT venta_pk PRIMARY KEY (v_folio),
    CONSTRAINT venta_cliente_fk FOREIGN KEY (
       v_cliente_rfc_client)
        REFERENCES public.cliente (v_rfc_client) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

La tabla registra las transacciones realizadas por los clientes. El campo `v_folio` actúa como clave primaria para identificar cada venta de forma única. Se incluye una clave foránea que vincula cada venta con un cliente, lo que garantiza la existencia de un registro correspondiente en la tabla `cliente`. También se registra la fecha, el monto total, la cantidad total de artículos vendidos y un folio de facturación opcional.

Tabla VENTA_EMPLEADO:

```
    CREATE TABLE IF NOT EXISTS public.venta_empleado
(
    v_venta_folio character varying(20) COLLATE pg_catalog."
       default" NOT NULL,
    n_empleado_clave_emp numeric(5,0) NOT NULL,
    CONSTRAINT venta_empleado_pk PRIMARY KEY (v_venta_folio,
       n_empleado_clave_emp),
    CONSTRAINT venta_empleado_empleado_fk FOREIGN KEY (
       n_empleado_clave_emp)
        REFERENCES public.empleado (n_clave_emp) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT venta_empleado_venta_fk FOREIGN KEY (
       v_venta_folio)
        REFERENCES public.venta (v_folio) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE
)
```

La tabla representa la relación entre ventas y empleados, indicando qué empleado realizó una venta. La clave primaria compuesta por `v_venta_folio` y `n_empleado_clave_emp` garantiza que no se repitan asignaciones. Las claves foráneas aseguran que los datos referencien ventas y empleados válidos. Se utiliza `ON DELETE CASCADE` para eliminar automáticamente las relaciones si se elimina la venta o el empleado correspondiente.

Cabe destacar algunas consideraciones adicionales como el uso de la cláusula `IF NOT EXISTS` para evitar errores si la tabla ya ha sido creada previamente, lo que resulta bastante útil.

Así también, el uso de un prefijo en los nombres de las columnas (v_, n_, b_,...) indicando el tipo de dato y que es una convención utilizada para facilitar la lectura y el mantenimiento del código. Por lo tanto, con estas definiciones de estuctura la base de datos conforme a los establecido en el programa.

INDICE

Para optimizar el rendimiento de las consultas relacionadas con la tabla `articulo`, se creó el índice `idx_articulo_clave_cat` sobre la columna `n_categoria_clave_cat`, utilizando el siguiente indice:

```
    CREATE INDEX idx_articulo_clave_cat
```

```
2     ON articulo (n_categoria_clave_cat);
```

Este índice permite acelerar significativamente las operaciones de búsqueda y filtrado en las que se involucra dicha columna, especialmente en consultas que realizan uniones (*joins*) o filtros con la tabla `categoria`.

El mecanismo utilizado internamente para este tipo de índice es comúnmente un árbol B (B-Tree) balanceado, lo cual permite organizar y acceder a los registros de forma más eficiente. En lugar de escanear secuencialmente toda la tabla para encontrar coincidencias, el sistema de gestión de bases de datos (en este caso PostgreSQL) recurre al índice para localizar rápidamente las filas que cumplen con el criterio, reduciendo así el tiempo de respuesta de las consultas.

Este tipo de optimización es particularmente útil en sistemas donde se manejan grandes volúmenes de información y se requiere una respuesta ágil en tiempo real, como en sistemas de punto de venta o inventarios, donde la categorización de artículos es clave para la organización y recuperación de datos.

4. Implementación

Para esta fase, una vez ya definida la estructura de nuestra base de datos e identificadas y creadas las tablas con las que vamos a trabajar, los atrbituos de cada una de ellas, sus tipos de datos y sus correspondientes restricciones y restricciones de integridad referencial, procedemos a la abstraccion y desarrollo de código SQL necesario para cumplir con los requerimientos de funcionamiento de este proyecto.

Para poder llegar al cumplimiento de lo requerido se implemento lo siguiente:

- Funciones
- Triggers

4.1. Funciones

Funciones PL/pgSQL

A lo largo del desarrollo del proyecto, se implementaron diversas funciones utilizando el lenguaje `PL/pgSQL` con el objetivo de extender las capacidades de la base de datos y automatizar procesos clave en la lógica del sistema. Estas funciones están diseñadas para encapsular operaciones complejas, mejorar la eficiencia de las consultas y garantizar la integridad de los datos mediante validaciones y reglas definidas a nivel del servidor.

Cada función cumple un propósito específico dentro del modelo de negocio, tales como la obtención de detalles estructurados para generación de tickets de venta, la validación de credenciales y roles del personal, la verificación de relaciones entre empleados de distintas áreas, y la recuperación de jerarquías dentro de la organización. Además, se han desarrollado funciones para facilitar el acceso a información relevante, como el inventario de artículos y sus respectivas categorías.

A continuación, se describen cada una de las funciones desarrolladas, incluyendo su lógica, parámetros de entrada, tipo de retorno y su propósito dentro del sistema.

4.1.1. Convenciones generales de funciones almacenadass

La mayoria de las funciones almacenadas descritas en este documento comparten las siguientes características generales, a menos que se indique lo contrario:

- Están escritas en el lenguaje **PL/pgSQL**.
- Utilizan la cláusula `CREATE OR REPLACE FUNCTION`, lo que permite su redefinición sin eliminar versiones previas.
- Se emplea la directiva `VOLATILE`, ya que los resultados pueden variar entre ejecuciones, dependiendo del estado actual de los datos.
- La opción `PARALLEL UNSAFE` está establecida para evitar problemas en entornos de ejecución paralela.
- Se utiliza la cláusula `RETURNS TABLE` para retornar múltiples columnas directamente.
- La estructura de cada función se organiza dentro de un bloque `BEGIN ... END`, donde se encapsula la lógica principal.
- El parámetro `COST` se establece en `100` como valor por defecto.
- Las funciones pueden incluir lógica condicional mediante `CASE`, filtros con `WHERE`, y consultas con `RETURN QUERY` y `SELECT`.
- Las funciones tienen manejo de errores: contiene una sección `EXCEPTION` que captura cualquier error y lanza un mensaje con la causa del problema.

4.1.2. Definición y detalles

FILTRAR ARTICULOS INSUFICIENTES

```
1     CREATE OR REPLACE FUNCTION public.
         sp_filtrar_articulos_insuficientes(
2   )
3     RETURNS TABLE(v_cod_barras character varying, v_nombre_art
          character varying, n_precio_compra numeric,
         n_precio_venta numeric, n_stock numeric, estado_stock
         text)
4     LANGUAGE 'plpgsql'
5     COST 100
6     VOLATILE PARALLEL UNSAFE
7     ROWS 1000
8
9   AS $BODY$
10  BEGIN
11      RETURN QUERY
12      SELECT
13          a.v_cod_barras,
14          a.v_nombre_art,
15          a.n_precio_compra,
16          a.n_precio_venta,
17          a.n_stock,
18          CASE
19              WHEN a.n_stock = 0 THEN 'Articulo no disponible'
20              ELSE 'Articulo poco disponible'
21          END AS estado_stock
22      FROM public.articulo a
23      WHERE a.n_stock < 3;
24  END;
25  $BODY$;
```

Propósito: En cumplimiento del requerimiento para listar los articulos no disponibles o con stock menor que tres y categorizándolos según su nivel de disponibilidad.

- **Parámetros de entrada:** Ninguno.
- **Tipo de retorno:** Tabla con las siguientes columnas:
  - `v_cod_barras`: Código de barras del artículo.
  - `v_nombre_art`: Nombre del artículo.

- `n_precio_compra`: Precio de compra.
- `n_precio_venta`: Precio de venta.
- `n_stock`: Cantidad en inventario.
- `estado_stock`: Texto calculado que indica si el artículo está *no disponible* (cuando el stock es 0) o *poco disponible* (cuando el stock es mayor a 0 pero menor a 3).

- **Lógica:** Se realiza una consulta sobre la tabla `articulo`, filtrando aquellos artículos cuyo stock sea menor a 3. La columna `estado_stock` se genera dinámicamente mediante una cláusula `CASE` que evalúa el valor de `n_stock`.

FUNCION PARA CONTROL DE VENTA

```
    CREATE OR REPLACE FUNCTION public.
       sp_insertar_venta_completa(
  p_cliente_rfc character varying,
  p_vendedor_clave numeric,
  p_cajero_clave numeric,
  p_articulos json)
    RETURNS numeric
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
DECLARE
    articulo JSON;
    cod_barras VARCHAR;
    cantidad NUMERIC;
    precio_unitario NUMERIC;
    total_venta NUMERIC := 0;
    total_cantidad NUMERIC := 0;
    ultimo_folio VARCHAR;
    ultimo_numero INT;
    nuevo_numero INT;
    nuevo_folio VARCHAR;
    folio_facturacion VARCHAR;
    validacion_result RECORD;
    articulos_agrupados JSON;
BEGIN
    -- Agrupar artculos duplicados del JSON de entrada
    SELECT json_agg(
```

```
28          json_build_object(
29              'cod_barras', sub.cb,
30              'cantidad', sub.sum_cant
31          )
32      )
33      INTO articulos_agrupados
34      FROM (
35          SELECT
36              (art->>'cod_barras') AS cb,
37              SUM((art->>'cantidad')::NUMERIC) AS sum_cant
38          FROM json_array_elements(p_articulos) AS art
39          GROUP BY (art->>'cod_barras')
40      ) AS sub;
41
42      -- Validar que vendedor y cajero pertenecen a la misma
           sucursal
43      SELECT * INTO validacion_result
44      FROM sp_validar_vendedor_cajero_misma_sucursal(
           p_vendedor_clave, p_cajero_clave);
45
46      IF validacion_result.validacion = 0 THEN
47          RAISE EXCEPTION '%', validacion_result.mensaje;
48      END IF;
49
50      -- Obtener el nuevo folio para la venta
51      SELECT v_folio INTO ultimo_folio
52      FROM venta
53      WHERE v_folio LIKE 'MBL-%'
54      ORDER BY v_folio DESC
55      LIMIT 1;
56
57      IF ultimo_folio IS NULL THEN
58          nuevo_numero := 1;
59      ELSE
60          ultimo_numero := CAST(SUBSTRING(ultimo_folio FROM 5)
               AS INTEGER);
61          nuevo_numero := ultimo_numero + 1;
62      END IF;
63
64      nuevo_folio := 'MBL-' || LPAD(nuevo_numero::TEXT, 3, '0');
65      folio_facturacion := 'FAC-' || UPPER(SUBSTRING(md5(random
```

```
        ()::text) FROM 1 FOR 6));
66
67      -- Validar existencia del cliente y empleados
68      IF NOT EXISTS (SELECT 1 FROM cliente WHERE v_rfc_client =
           p_cliente_rfc) THEN
69          RAISE EXCEPTION 'El cliente con RFC % no existe',
               p_cliente_rfc;
70      END IF;
71
72      IF NOT EXISTS (SELECT 1 FROM empleado WHERE n_clave_emp =
           p_vendedor_clave) THEN
73          RAISE EXCEPTION 'El empleado (vendedor) con clave % no
               existe', p_vendedor_clave;
74      END IF;
75
76      IF NOT EXISTS (SELECT 1 FROM empleado WHERE n_clave_emp =
           p_cajero_clave) THEN
77          RAISE EXCEPTION 'El empleado (cajero) con clave % no
               existe', p_cajero_clave;
78      END IF;
79
80      -- Calcular totales de la venta
81      FOR articulo IN SELECT * FROM json_array_elements(
           articulos_agrupados)
82      LOOP
83          cod_barras := articulo->>'cod_barras';
84          cantidad := (articulo->>'cantidad')::NUMERIC;
85
86          SELECT n_precio_venta
87          INTO precio_unitario
88          FROM articulo
89          WHERE v_cod_barras = cod_barras;
90
91          IF NOT FOUND THEN
92              RAISE EXCEPTION 'El articulo con codigo de barras
                   % no existe', cod_barras;
93          END IF;
94
95          total_venta := total_venta + (precio_unitario *
               cantidad);
96          total_cantidad := total_cantidad + cantidad;
```

```
97      END LOOP;
98
99      -- Insertar en tabla venta
100     INSERT INTO venta (
101         d_fecha, v_folio, n_monto_total, n_cant_totalart,
102         v_cliente_rfc_client, v_folio_facturacion
103     )
104     VALUES (
105         CURRENT_DATE, nuevo_folio, total_venta, total_cantidad
                ,
106         p_cliente_rfc, folio_facturacion
107     );
108
109     -- Insertar relacion con empleados
110     INSERT INTO venta_empleado (v_venta_folio,
           n_empleado_clave_emp)
111     VALUES
112         (nuevo_folio, p_vendedor_clave),
113         (nuevo_folio, p_cajero_clave);
114
115     -- Insertar los articulos agrupados a la venta
116     FOR articulo IN SELECT * FROM json_array_elements(
           articulos_agrupados)
117     LOOP
118         cod_barras := articulo->>'cod_barras';
119         cantidad := (articulo->>'cantidad')::NUMERIC;
120
121         INSERT INTO articulo_venta (
122             v_venta_folio, v_articulo_cod_barras, n_cantidad
123         )
124         VALUES (
125             nuevo_folio, cod_barras, cantidad
126         );
127     END LOOP;
128
129     RETURN total_venta;
130
131 EXCEPTION
132     WHEN OTHERS THEN
133         RAISE NOTICE 'Error al insertar la venta: %', SQLERRM;
134         RAISE;
```

```
135 END;
136 $BODY$;
```

Propósito: Esta función permite registrar una venta completa en el sistema, incluyendo la generación del folio de venta, asignación de artículos vendidos, relación con los empleados participantes (vendedor y cajero), y el cálculo del total monetario. Además, agrupa los artículos duplicados presentes en el JSON de entrada para evitar registros redundantes. Esta función forma parte del cumplimiento del *Requerimiento 1*.

- **Parámetros de entrada:**
  - `p_cliente_rfc`: RFC del cliente que realiza la compra.
  - `p_vendedor_clave`: Clave del empleado que realiza la venta.
  - `p_cajero_clave`: Clave del empleado que cobra la venta.
  - `p_articulos`: JSON que contiene los artículos vendidos (código de barras y cantidad).
- **Tipo de retorno:** Valor `numeric` que representa el monto total de la venta.
- **Proceso general:**
  1. Agrupa los artículos duplicados por código de barras para sumar sus cantidades.
  2. Verifica que el cajero y el vendedor pertenezcan a la misma sucursal, utilizando una función auxiliar.
  3. Genera un nuevo folio de venta con formato `MBL-XXX` y un código de facturación aleatorio.
  4. Valida la existencia del cliente, vendedor y cajero.
  5. Calcula el monto total de la venta y la cantidad total de artículos vendidos, consultando el precio de cada artículo.
  6. Inserta la venta en la tabla `venta`.
  7. Registra la relación de la venta con el vendedor y el cajero en la tabla `venta_empleado`.
  8. Inserta cada artículo vendido en la tabla `articulo_venta`.

Esta función centraliza el proceso de venta, asegurando integridad y validación en cada paso clave del registro.

FUNCION DETALLE VENTA

```
1     CREATE OR REPLACE FUNCTION public.sp_obtener_detalle_venta
          (
2   p_folio_venta character varying)
3     RETURNS TABLE(folio_venta character varying,
          empleado_nombre text, cliente_nombre text,
          total_articulos numeric, monto_total numeric,
          folio_facturacion character varying, clave_sucursal
          character varying, articulos text)
4     LANGUAGE 'plpgsql'
5     COST 100
6     VOLATILE PARALLEL UNSAFE
7     ROWS 1000
8
9   AS $BODY$
10  BEGIN
11      RETURN QUERY
12      SELECT
13          v.v_folio,
14          CONCAT(emp.v_nombre, ' ', emp.v_ap_paterno, ' ', emp.
                v_ap_mat),
15          CONCAT(cli.v_nombre, ' ', cli.v_ap_paterno, ' ', cli.
                v_ap_materno),
16          v.n_cant_totalart,
17          v.n_monto_total,
18          v.v_folio_facturacion,
19          emp.v_sucursal_clave_suc,
20          string_agg(
21              CONCAT(
22                  '(', art.v_cod_barras, ') ',
23                  art.v_nombre_art, ' x', av.n_cantidad,
24                  ' $', art.n_precio_venta
25              ),
26              ' | '
27          )
28      FROM venta v
29      JOIN venta_empleado ve ON ve.v_venta_folio = v.v_folio
30      JOIN empleado emp ON emp.n_clave_emp = ve.
            n_empleado_clave_emp
31      JOIN cliente cli ON cli.v_rfc_client = v.
```

```
        v_cliente_rfc_client
32      JOIN articulo_venta av ON av.v_venta_folio = v.v_folio
33      JOIN articulo art ON art.v_cod_barras = av.
            v_articulo_cod_barras
34      WHERE v.v_folio = p_folio_venta AND emp.v_tipo = 'Vendedor
            '
35      GROUP BY v.v_folio, emp.v_nombre, emp.v_ap_paterno, emp.
            v_ap_mat,
36              cli.v_nombre, cli.v_ap_paterno, cli.v_ap_materno,
37              v.n_cant_totalart, v.n_monto_total, v.
                    v_folio_facturacion,
38              emp.v_sucursal_clave_suc;
39  END;
40  $BODY$;
```

Propósito: Esta función recupera el detalle completo de una venta específica con base en su folio, incluyendo información del vendedor, cliente, artículos vendidos, totales y sucursal relacionada asemejando la vista del ticket de venta, en cumplimiento del *Requerimiento 4*

- **Parámetro de entrada:**
  - `p_folio_venta (character varying)`: Folio único de la venta a consultar.
- **Valor de retorno:** `TABLE` con los siguientes campos:
  - `folio_venta`: Folio de la venta.
  - `empleado_nombre`: Nombre completo del vendedor que realizó la venta.
  - `cliente_nombre`: Nombre completo del cliente.
  - `total_articulos`: Total de artículos vendidos.
  - `monto_total`: Monto total de la venta.
  - `folio_facturacion`: Folio único de facturación.
  - `clave_sucursal`: Clave de la sucursal donde se realizó la venta.
  - `articulos`: Cadena de texto con el listado de artículos vendidos, en el formato `(código) nombre x cantidad $precio`, separados por `|`.

Lógica: La función realiza un conjunto de `JOINs` sobre las tablas `venta`, `venta_empleado`, `empleado`, `cliente`, `articulo_venta` y `articulo`. Se filtra únicamente al empleado con tipo `Vendedor` para mostrar su información. Los artículos vendidos se concatenan usando `string_agg` para devolver una lista textual.

Observaciones:

- El uso de `GROUP BY` asegura que la agregación de los artículos no genere duplicados en la consulta.
- La validación del tipo de empleado garantiza que se muestre únicamente el vendedor asociado a la venta.

FUNCION OBTENER JERARQUIA ORGANIZACIONAL

```
    CREATE OR REPLACE FUNCTION public.sp_obtener_jerarquia(
  in_clave_emp numeric)
    RETURNS TABLE(clave numeric, nombre character varying,
       nivel integer)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
BEGIN
    RETURN QUERY
    WITH RECURSIVE jerarquia AS (
        SELECT
            e.n_clave_emp,
            e.v_nombre,
            1 AS nivel
        FROM empleado e
        WHERE e.n_clave_emp = in_clave_emp

        UNION ALL

        SELECT
            jefe.n_clave_emp,
            jefe.v_nombre,
            j.nivel + 1
        FROM empleado jefe
        JOIN jerarquia j ON jefe.n_clave_emp = (
            SELECT e.n_empleado_clave_emp
            FROM empleado e
            WHERE e.n_clave_emp = j.n_clave_emp
        )
        WHERE jefe.n_clave_emp IS NOT NULL
    )
```

```
34      SELECT
35          j.n_clave_emp,
36          j.v_nombre,
37          j.nivel
38      FROM jerarquia j
39      ORDER BY j.nivel DESC;
40  END;
41  $BODY$;
```

Propósito: La función anterior tiene como finalidad obtener de manera recursiva la jerarquía de empleados por niveles, partiendo desde un empleado base (clave proporcionada como parámetro) y subiendo a través de sus jefes directos hasta llegar al nivel más alto de la estructura organizacional. Esta función cumple el *Requerimiento 6*

Parámetros de entrada:

- `in_clave_emp` (`numeric`): Clave del empleado del cual se desea conocer su jerarquía ascendente.

Tipo de retorno: La función retorna una tabla con los siguientes campos:

- `clave` (`numeric`): Clave del empleado en la jerarquía.
- `nombre` (`character varying`): Nombre del empleado.
- `nivel` (`integer`): Nivel jerárquico dentro de la cadena de mando, siendo 1 el empleado inicial y aumentando conforme se asciende en la jerarquía.

Lógica: Se utiliza una expresión `WITH RECURSIVE` para recorrer la jerarquía de manera ascendente:

1. Se parte del empleado base, con nivel 1.
2. A través de un `JOIN` sobre la misma tabla `empleado`, se recupera el jefe inmediato superior usando el campo `n_empleado_clave_emp`.
3. El proceso se repite recursivamente mientras existan superiores jerárquicos.
4. Finalmente, se ordena el resultado de forma descendente según el nivel, de manera que el empleado de mayor jerarquía aparece al inicio.

FUNCION DE VALIDACION EMPLEADO

```
    CREATE OR REPLACE FUNCTION public.sp_inicio_sesion(
  p_n_clave_emp numeric,
  "p_contrasena" text)
    RETURNS TABLE(validacion integer, n_clave_emp numeric,
       v_rfc_emp character varying, d_fecha_ing date, v_curp
       character varying, v_email_emp character varying,
       v_nombre character varying, v_ap_paterno character
       varying, v_ap_mat character varying,
       n_empleado_clave_emp numeric, v_sucursal_clave_suc
       character varying, v_tipo character varying,
       n_direccion_id_direccion numeric)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
BEGIN
    RETURN QUERY
    SELECT
        1 AS validacion,
        e.n_clave_emp,
        e.v_rfc_emp,
        e.d_fecha_ing,
        e.v_curp,
        e.v_email_emp,
        e.v_nombre,
        e.v_ap_paterno,
        e.v_ap_mat,
        e.n_empleado_clave_emp,
        e.v_sucursal_clave_suc,
        e.v_tipo,
        e.n_direccion_id_direccion
    FROM empleado e
    WHERE e.n_clave_emp = p_n_clave_emp
      AND e.t_contrasena = p_contrasena
      AND LOWER(e.v_tipo) IN ('vendedor', 'cajero');

    IF NOT FOUND THEN
```

```
33          RETURN QUERY
34          SELECT
35              0 AS validacion,
36              NULL::numeric,
37              NULL::character varying,
38              NULL::date,
39              NULL::character varying,
40              NULL::character varying,
41              NULL::character varying,
42              NULL::character varying,
43              NULL::character varying,
44              NULL::numeric,
45              NULL::character varying,
46              NULL::character varying,
47              NULL::numeric;
48      END IF;
49  END;
50  $BODY$;
```

Propósito: La función `sp_validar_empleado` tiene como objetivo autenticar a un empleado verificando su clave y contraseña, y asegurar que su tipo sea válido (`vendedor` o `cajero`). En caso de éxito, devuelve sus datos personales y laborales relevantes; de lo contrario, retorna un conjunto de campos nulos indicando una validación fallida.

Parámetros de entrada:

- `p_n_clave_emp` (`numeric`): Clave única del empleado.
- `p_contrasena` (`text`): Contraseña del empleado.

Tipo de retorno: Una tabla con los siguientes campos:

- `validacion` (`integer`): Indicador de validación (1 si es válida, 0 si no).
- `n_clave_emp` (`numeric`): Clave del empleado.
- `v_rfc_emp` (`character varying`): RFC del empleado.
- `d_fecha_ing` (`date`): Fecha de ingreso del empleado.
- `v_curp` (`character varying`): CURP del empleado.
- `v_email_emp` (`character varying`): Correo electrónico.
- `v_nombre`, `v_ap_paterno`, `v_ap_mat`: Datos del nombre completo.

- `n_empleado_clave_emp` (`numeric`): Clave del jefe inmediato (si aplica).
- `v_sucursal_clave_suc` (`character varying`): Clave de la sucursal a la que pertenece.
- `v_tipo` (`character varying`): Tipo de empleado.
- `n_direccion_id_direccion` (`numeric`): Identificador de la dirección asociada.

Lógica :

1. Se realiza una búsqueda en la tabla `empleado` comparando la clave y contraseña proporcionadas.
2. Se filtra que el tipo de empleado sea `vendedor` o `cajero`, ignorando mayúsculas/-minúsculas.
3. Si la validación es exitosa, se retorna una fila con todos los datos del empleado y un indicador `validacion = 1`.
4. Si no se encuentra coincidencia, se retorna una fila con `validacion = 0` y el resto de campos nulos.

FUNCION VALIDAR CAJERO

```
1     CREATE OR REPLACE FUNCTION public.
          sp_validar_vendedor_cajero_misma_sucursal(
2   p_emp1 numeric,
3   p_emp2 numeric)
4     RETURNS TABLE(validacion integer, mensaje text)
5     LANGUAGE 'plpgsql'
6     COST 100
7     VOLATILE PARALLEL UNSAFE
8     ROWS 1000
9
10 AS $BODY$
11 DECLARE
12     emp1 RECORD;
13     emp2 RECORD;
14 BEGIN
15     SELECT v_tipo, v_sucursal_clave_suc INTO emp1
16     FROM empleado
17     WHERE n_clave_emp = p_emp1;
18
19     IF NOT FOUND THEN
20         RETURN QUERY SELECT 0, format('Empleado %s no existe',
               p_emp1);
21         RETURN;
22     END IF;
23
24     SELECT v_tipo, v_sucursal_clave_suc INTO emp2
25     FROM empleado
26     WHERE n_clave_emp = p_emp2;
27
28     IF NOT FOUND THEN
29         RETURN QUERY SELECT 0, format('Empleado %s no existe',
               p_emp2);
30         RETURN;
31     END IF;
32
33     IF (
34         (LOWER(emp1.v_tipo) = 'vendedor' AND LOWER(emp2.v_tipo
              ) = 'cajero') OR
35         (LOWER(emp1.v_tipo) = 'cajero' AND LOWER(emp2.v_tipo)
```

```
            = 'vendedor')
36    ) THEN
37  IF emp1.v_sucursal_clave_suc = emp2.v_sucursal_clave_suc
       THEN
38            RETURN QUERY SELECT 1, 'Validacion exitosa:
                 vendedor y cajero en la misma sucursal';
39        ELSE
40            RETURN QUERY SELECT 0, 'Error: los empleados no
                 pertenecen a la misma sucursal';
41        END IF;
42    ELSE
43        RETURN QUERY SELECT 0, 'Error: los empleados no son
              vendedor y cajero de forma excluyente';
44    END IF;
45 END;
46 $BODY$;
```

Propósito: Esta función valida que dos empleados dados (por su clave) sean respectivamente un vendedor y un cajero, sin importar el orden, y que ambos pertenezcan a la misma sucursal. Se utiliza para garantizar que ciertos procesos o transacciones se lleven a cabo solo entre personal autorizado y en el mismo contexto organizacional. Dicha funció forma parte del cumplimiento del *Requerimiento 5*.

Parámetros de entrada:

- `p_emp1` (`numeric`): Clave del primer empleado.
- `p_emp2` (`numeric`): Clave del segundo empleado.

Tipo de retorno: Una tabla con los siguientes campos:

- `validacion` (`integer`): 1 si la validación es exitosa, 0 en caso contrario.
- `mensaje` (`text`): Mensaje descriptivo del resultado de la validación.

Lógica:

1. Se consulta la tabla `empleado` para obtener el tipo y la sucursal de cada clave proporcionada.
2. Si alguno de los empleados no existe, se retorna un mensaje de error específico.
3. Se verifica que uno de los empleados sea `vendedor` y el otro `cajero`, sin importar el orden.
4. Si cumplen con ese criterio, se compara la clave de sucursal de ambos.

5. Si pertenecen a la misma sucursal, la validación es exitosa; de lo contrario, se retorna un mensaje de error.

6. Si los tipos no corresponden a vendedor y cajero, también se rechaza la validación.

Observaciones: Esta función es útil para validar roles colaborativos en operaciones como ventas, devoluciones o autorizaciones, asegurando integridad entre roles definidos y su ubicación organizacional.

FUNCION OBTENER ARTICULOS

```
    CREATE OR REPLACE FUNCTION sp_obtener_articulos()
RETURNS TABLE (
    cod_barras VARCHAR(20),
    nombre_art VARCHAR(50),
    precio_compra numeric(8,2),
    precio_venta numeric(8,2),
    stock numeric(7,0),
    fotografia BYTEA,
    categoria_clave_cat numeric(5,0),
    nombre_cat VARCHAR(20),
    tipo_cat VARCHAR(20)
)
AS $$
BEGIN
    RETURN QUERY
    SELECT
        a.v_cod_barras,
        a.v_nombre_art,
        a.n_precio_compra,
        a.n_precio_venta,
        a.n_stock,
        a.b_fotografia,
        a.n_categoria_clave_cat,
        c.v_nombre_cat,
        c.v_tipo_cat
    FROM
        articulo a
    JOIN
        categoria c ON a.n_categoria_clave_cat = c.n_clave_cat
           ;
END;
$$ LANGUAGE plpgsql;
```

Parámetros de entrada:
Sin parametros de entrada.
Tipo de retorno:
Retorna una tabla con varias columnas que contienen información de los artículos registrados en el sistema. Los campos devueltos incluyen el código de barras del artículo, su nombre, precios de compra y venta, el stock disponible, la fotografía del artículo en formato binario, así como la clave, el nombre y el tipo de la categoría a la que pertenece.
Lógica:
Esta función tiene como propósito recuperar un listado completo de artículos registrados en el sistema, incluyendo información esencial como código de barras, nombre, precios (compra y venta), stock disponible y fotografía. Además, incorpora detalles de la categoría a la que pertenece cada artículo (nombre y tipo), lo cual se logra mediante una unión (`JOIN`) entre las tablas `articulo` y `categoria`. Esta función facilita la visualización estructurada de los productos.

4.2. Triggers

4.2.1. Función y Trigger: sp_set_razon_social

```
CREATE OR REPLACE FUNCTION public.sp_set_razon_social()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
BEGIN
    -- Solo si razon_social esta vacio o NULL
    IF NEW.v_razon_social IS NULL OR trim(NEW.v_razon_social)
       = '' THEN
        NEW.v_razon_social := trim(concat_ws(' ', NEW.v_nombre
           , NEW.v_ap_paterno, NEW.v_ap_materno));
    END IF;
    RETURN NEW;
END;
$BODY$;
```

Propósito:

- Autocompleta el campo `v_razon_social` cuando está vacío
- Concatena nombre, apellido paterno y apellido materno para formar la razón social

Trigger Asociado:

```
CREATE OR REPLACE TRIGGER tr_set_razon_social
    BEFORE INSERT OR UPDATE
    ON public.cliente
    FOR EACH ROW
    EXECUTE FUNCTION public.sp_set_razon_social();
```

Detalles de Implementación:

- Tipo: Trigger `BEFORE` (antes de la operación)
- Eventos: Se ejecuta en `INSERT` y `UPDATE`
- Tabla: `public.cliente`
- Nivel: `FOR EACH ROW` (por cada fila afectada)

- Condición: Solo actúa cuando `v_razon_social` es NULL o string vacío
- Función: `trim(concat_ws())` elimina espacios y concatena con separador

4.2.2. Función y Trigger: sp_validar_y_actualizar_stock

```
CREATE OR REPLACE FUNCTION public.
   sp_validar_y_actualizar_stock()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
    stock_actual numeric;
BEGIN
    SELECT n_stock INTO stock_actual
    FROM articulo
    WHERE v_cod_barras = NEW.v_articulo_cod_barras;

    IF stock_actual IS NULL THEN
        RAISE EXCEPTION 'Articulo % no existe', NEW.
           v_articulo_cod_barras;
    END IF;

    IF stock_actual < NEW.n_cantidad THEN
        RAISE EXCEPTION 'Stock insuficiente para el articulo %
            (Stock: %, Solicitado: %)',
            NEW.v_articulo_cod_barras, stock_actual, NEW.
               n_cantidad;
    END IF;

    -- Descontar del stock
    UPDATE articulo
    SET n_stock = n_stock - NEW.n_cantidad
    WHERE v_cod_barras = NEW.v_articulo_cod_barras;

    RETURN NEW;
END;
$BODY$;
```

Propósito:

- Valida el stock disponible antes de registrar una venta
- Actualiza el inventario automáticamente
- Previene ventas de artículos inexistentes o con stock insuficiente

Trigger Asociado:

```
CREATE OR REPLACE TRIGGER tr_validar_stock
    BEFORE INSERT
    ON public.articulo_venta
    FOR EACH ROW
    EXECUTE FUNCTION public.sp_validar_y_actualizar_stock();
```

Detalles de Implementación:

- Tipo: Trigger `BEFORE INSERT`
- Tabla: `public.articulo_venta`
- Lógica:
  1. Obtiene el stock actual del artículo
  2. Valida que el artículo exista
  3. Verifica que haya suficiente stock
  4. Si todo es válido, descuenta la cantidad vendida
  5. Si hay error, aborta la operación con mensaje descriptivo
- Excepciones:
  - Código de barras no encontrado
  - Stock insuficiente

5. Aplicación

5.1. Requisitos e Instalación

Para poder visualizar el dashboard creado para esta base de datos, es necesario realizar algunos pasos.

5.1.1. Instalación de psycopg2

Como primer paso, necesitaremos conectar con el manejador usado, en este caso, PostgreSQL. Instalamos la biblioteca psycopg2, diseñada para poder conectar desde python a la base de datos de dicho manejador. Para llevar a cabo la instalación, basta con ejecutar el siguiente comando desde nuestra terminal:

```
pip install psycopg2
```

5.1.2. Ejecución del Programa

Finalizada la instalación de psycopg2, nuevamente en la terminal de comandos desde la ubicación del archivo, ejecutar el script principal con:

```
python dashboard.py
```

5.2. Funciones Principales

5.2.1. Conexión a la Base de Datos

```
def conectar_db():
    return psycopg2.connect(
        dbname="muebleria",
        user="tu_usuario",
        password="tu_contrasenia",
        host="localhost",
        port="5432"
    )
```

Propósito: Establece conexión con la base de datos PostgreSQL. Para poder ejecutar correctamente el programa, es necesario realizar algunas modificaciones en el segmento de código mostrado. En `dbname`, debemos sustituir `muebleria` por el nombre que lleve la base de datos cargada en nuestro sistema, así como el usuario en nuestro manejador y la contraseña, sustituyendo la asignación respectivamente en `user` y `password`.

5.2.2. Consulta de Ventas

```
def obtener_ventas(anio, sucursal=None):
    query = """
    SELECT
        c.v_nombre_cat as categoria,
        EXTRACT(MONTH FROM v.d_fecha) as mes,
        SUM(v.n_monto_total) as ventas          --Total de
           ventas
    FROM venta v
    JOIN articulo_venta av ON v.v_folio = av.v_venta_folio
    JOIN articulo a ON av.v_articulo_cod_barras = a.
       v_cod_barras
    JOIN categoria c ON a.n_categoria_clave_cat = c.
       n_clave_cat
    JOIN venta_empleado ve ON v.v_folio = ve.v_venta_folio
    JOIN empleado e ON ve.n_empleado_clave_emp = e.n_clave_emp
    WHERE EXTRACT(YEAR FROM v.d_fecha) = %s
    --Condicion si se especifica alguna sucursal o se omite
    """ + ("AND e.v_sucursal_clave_suc = %s" if sucursal else
       "") + """
    GROUP BY c.v_nombre_cat, EXTRACT(MONTH FROM v.d_fecha)
    """

    params = (anio,) if not sucursal else (anio, sucursal)

    with conectar_db() as conn:
        with conn.cursor() as cursor:
            #Ejecucion de la consulta, retorna un diccionario
               con la informacion extraida
            cursor.execute(query, params)
            return cursor.fetchall()
```

Este segmento de código ejecuta una consulta en lenguaje SQL.
Parámetros:

- `anio`: Año a consultar (ej. 2023)
- `sucursal`: (Opcional) Código de sucursal (ej. SUC001)

Retorno:

- Lista de tuplas con los resultados de la consulta, donde cada tupla contiene:

- `categoria`: Nombre de la categoría (String)
- `mes`: Número del mes (1-12) (Integer)
- `ventas`: Suma total de ventas (Float)

- Ejemplo de retorno: `[('Electrónicos', 1, 15000.50), ('Ropa', 1, 8000.75), ...]`
- Lista vacía `[]` si no hay datos para los parámetros dados

5.2.3. Generación de Gráficos

```
def generar_graficos(datos, anio, sucursal=None):
    #Inicializamos los datos a graficar
    categorias = {}
    #Meses en ceros
    meses = {mes: 0 for mes in range(1, 13)}

    #Suma ventas por categorias y por mes
    for categoria, mes, ventas in datos:
        categorias[categoria] = categorias.get(categoria, 0) +
            ventas
        meses[mes] += ventas


    #Crear figura, 14, 6 es el tamanio, define el tamanio de
       toda la ventana donde se graficara
    plt.figure(figsize=(14, 6))

    #Division de la ventana, el segundo '2' es la posicion de
       la grafica, en este caso saldra a la derecha
    plt.subplot(1, 2, 2)
    #Grafica de pastel de las ventas por categoria
    if categorias:
        plt.pie(categorias.values(), labels=categorias.keys(),
            autopct='%1.1f%%')
        #autopct es el formato del porcentaje
    plt.title(f'Ventas por Categoria\nAnio: {anio}' + (f'\
       nSucursal: {sucursal}' if sucursal else ''))

    #Segundo 2, veremos la grafica de barras a la izquierda
```

```
    plt.subplot(1, 2, 1)
    nombres_meses = ['Ene', 'Feb', 'Mar', 'Abr', 'May', 'Jun',
        'Jul', 'Ago', 'Sep', 'Oct', 'Nov', 'Dic']
    valores = [meses[mes] for mes in range(1, 13)]

    #Grafico de barras de ventas por mes en un anio dado
    plt.bar(nombres_meses, valores)
    plt.title(f'Ventas Mensuales\nAnio: {anio}' + (f'\
       nSucursal: {sucursal}' if sucursal else ''))
    plt.ylabel('Total de Ventas ($)')

    #Ajustes establecidos de plt.subplot
    plt.tight_layout()

    #Generar nombre del archivo dependiendo de las entradas
       ingresadas
    nombre_sucursal = sucursal if sucursal else "ALLSUC"
    nombre_archivo = f"ventas{nombre_sucursal}anio{anio}.png"

    #Guardar la figura, resolucion de 300 puntos por pulgada,
       tight para eliminar espacios en blanco alrededor
    plt.savefig(nombre_archivo, dpi=300, bbox_inches='tight')
    print(f"Grafica guardada como: {nombre_archivo}")

    plt.show()
```

Este segmento de código es utilizado para generar el dashboard.
Parámetros de Entrada:

- `datos`: Lista de tuplas con los datos de ventas en el formato:
  - Estructura: `(categoria, mes, ventas)`
  - Ejemplo: `[('Electrónicos', 1, 15000.50), ('Ropa', 1, 8000.75), ...]`
- `anio`: Año de referencia para los gráficos (Integer)
  - Ejemplo: `2023`
- `sucursal`: (Opcional) Código de sucursal para filtrar (String)
  - Si es `None`, se incluyen todas las sucursales
  - Ejemplo: `'SUC001'`

Retorno:

- La función no retorna valores explícitos (`None`)
- Muestra de la gráfica:
  - Muestra una ventana con dos gráficos:
    - Gráfico de barras: Ventas mensuales
    - Gráfico de pastel: Distribución por categorías
  - Guarda la imagen como archivo PNG con el nombre: `ventas[SUCURSAL]anio[ANIO].png`

Ejemplo de Uso:

```
datos = [('Electronicos', 1, 15000.50), ('Ropa', 1, 8000.75)]
generar_graficos(datos, 2023, 'SUC001')
```

5.2.4. Función Principal

```
def main():
    anio = int(input("Ingrese el anio a consultar: "))
    if(anio != 2024 and anio != 2023):
        print("Ingresa un anio correcto!")
        main()

    sucursal = input("Ingrese codigo de sucursal... ").strip()
       .upper()

    datos = obtener_ventas(anio, sucursal if sucursal else
       None)

    if datos:
        generar_graficos(datos, anio, sucursal if sucursal
           else None)
    else:
        print(f"No se encontraron datos..." + (f" y sucursal {
           sucursal}" if sucursal else ""))

if __name__ == "__main__":
    main()
```

Propósito:

- Función principal que coordina el flujo del programa
- Interactúa con el usuario para obtener parámetros de consulta
- Gestiona la lógica de visualización de resultados

Flujo de Ejecución:

1. Solicita al usuario:
    - Año a consultar (Ej. 2023)
    - Código de sucursal (opcional)
2. Valida el año ingresado (recursivamente si es inválido)
3. Consulta los datos llamando a `obtener_ventas()`
4. Si hay datos:
    - Genera gráficos llamando a `generar_graficos()`
5. Si no hay datos:
    - Muestra mensaje informativo

Notas Importantes:

- El bloque `if __name__ == "__main__":` asegura que la función solo se ejecute cuando el script es invocado directamente
- Usa recursión simple para validación de año
- Normaliza el input de sucursal (mayúsculas, sin espacios)

Ejemplo de Uso:

```
$ python dashboard.py
Ingrese el anio a consultar: 2023
Ingrese codigo de sucursal (ej. SUC001) o presione enter para
   todas: SUC001
```

5.3. Ejemplo de Dashboard generado

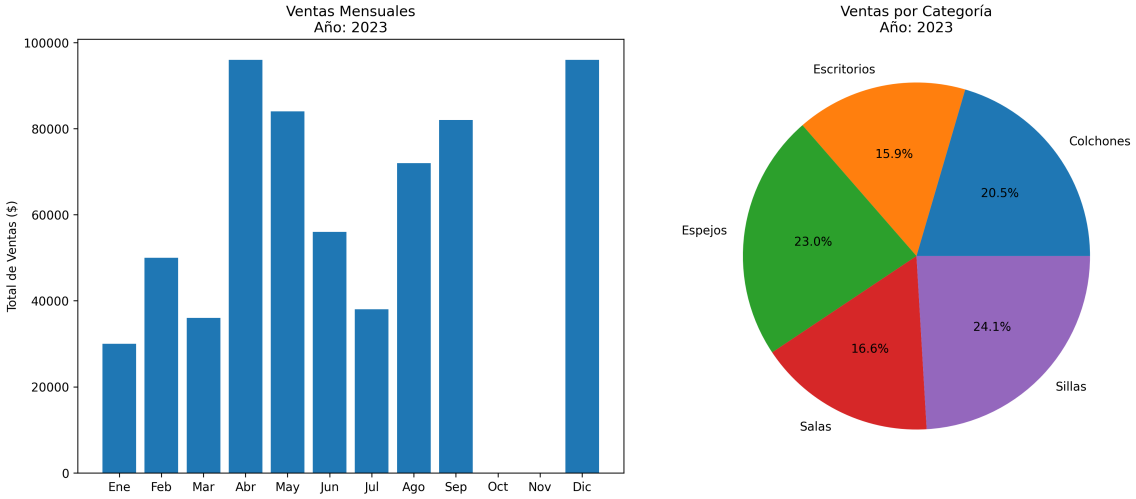

Figura 3: Gráfica de ventas por categoría (pastel) y ventas mensuales en un año dado (barras)

6. Documentación Técnica

6.1. API Backend – FastAPI (Python)

Se utilizo FastApi de Python en la versión 3.11+ con lo que se hizo un servidor local el cual puede funcionar para la conexión con nuestro front-end.

```
    from typing import List
    import psycopg2
    from fastapi import FastAPI
    import psycopg2.extras
    from fastapi.middleware.cors import CORSMiddleware
    from pydantic import BaseModel
```

Los cuales se utilizaron para nuestra implementación, para la observación se tiene List la cual da contexto del tipo de dato, ya que para la utilización de fastapi se necesita un control de los tipos de datos. En segundo lugar, la biblioteca `pyscopg2` la cual es necesaria para nuestra conexión con la base de datos. Las demás bibliotecas son utilidades extras para nuestra función.

6.2. CORS Middleware

Se ha habilitado CORS para permitir solicitudes desde la URL del frontend:

```
    app = FastAPI()

    app.add_middleware(
        CORSMiddleware,
        allow_origins=["http://localhost:5173"],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )
```

También de la inicialización de app como nuestro API.

```
    class LoginRequest(BaseModel):
        Id_Emp: int
        password: str

    class Articulo(BaseModel):
        cod_barras: str
        cantidad: int
```

```
    class Compra(BaseModel):
        rfc_cliente: str
        id_empleado: int
        id_cajero: int
        articulos: List[Articulo]
    class JerarquiaRequest(BaseModel):
        id_empleado: int
    class Jefe(BaseModel):
        clave: int
        nombre: str
        nivel: int
```

Todas estas clases son necesarias para este contexto de tipo de datos, que pueden recibir o que espera retornar el API. Para la conexion de base de datos se utilizo psycopg2 la cual puede conectar con los siguientes parámetros, eso cambia con respecto al usuario y su base de datos local, al menos que sea una versión de producción.

```
    def get_connection():
        try:
            connection = psycopg2.connect(
                host='localhost',
                user='admin_dev',
                password='320334489',
                database='proyectofinal',
                port="5432",

            )
            return connection
        except Exception as ex:
            print(ex)
```

Pasemos a las funciones que contiene nuestro API que puede consumir el front mediante un método.

6.3. POST/login

`Descripción:` Es de tipo post ya que se le manda un valor, que en este caso es Id y contraseña, también por que va a retornar un valor. Autentica a un empleado usando su ID y contraseña, consultando la función sp_validar_empleado, la cual es una función de base de datos que se emplea para validar si las credenciales de este usuario existen y si es correcta la contraseña que guardo.

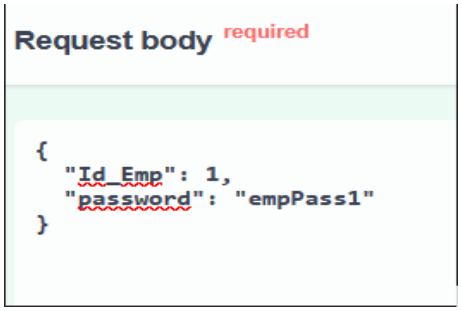

Figura 4: Request Body

Esto es lo que se lo que la función del API recibe de front-end:

Figura 5: API

Esto es lo que va a retornar a la parte de front-end si las credenciales son correctas, si no va a retornar un error. En la siguiente petición tenemos la llamada a los artículos para ponerlos en el front-end.

6.4. GET/art

`Descripción:` Obtiene todos los artículos disponibles utilizando la función sp_obtener_articulos(), la cual va a traer la información de los artículos junto con la de categoría.

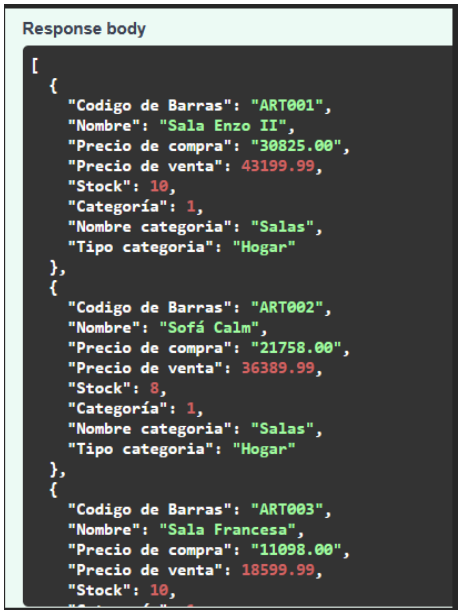

Figura 6: API - 2

6.5. POST/vent

`Descripción:` Inserta una venta completa, llamando a la función sp_insertar_venta_completa con datos del cliente y los artículos, la cual va a guardar la compra. Lo que espera recibir el API:

```
{
  "rfc_cliente": "XXXX010203AB1",
  "id_empleado": 1,
  "id_cajero": 1,
  "articulos": [
    { "cod_barras": "1234567890123", "cantidad": 2 }
  ]
}
```

Figura 7: Venta

Lo que retorna el API:

Figura 8: Retorno Venta

6.6. POST/jerarquia

`Descripción:` Obtiene todos los jefes jerárquicos de un empleado según la función sp_obtener_jerarquia(id_empleado). Lo que espera recibir:

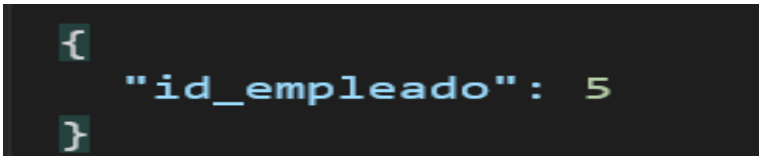

Figura 9: API Jerarquia

Lo que retorna el API:

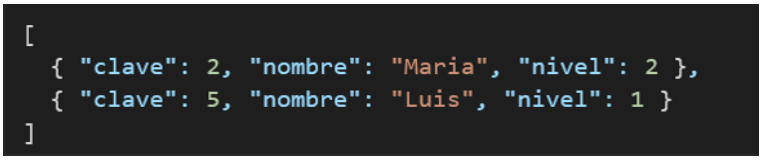

Figura 10: Retorno Jeraquia

6.7. Front-end

Para este proyecto se utilizo react con typescript estilado con styled-components. Como principal se inicio el proyecto con npm créate vite@latest con el nombre de proyectofinal. De inicios el proyecto inicia con distintas carpetas, las cuales no se le movieron y se inicio directamente en el apartado de App.tsx

```
import { BrowserRouter as Router, Routes, Route } from "
   react-router-dom";
import Home from "./page/Home";
import Login from "./page/Login/Login";
import { ProviderUser } from "./Context/ProviderUser";
import Articulos from "./page/Articulos";
import Datos from "./page/Datos";
import Venta from "./page/Venta";

function App() {
  return (
      <ProviderUser>
        <Router>
          <Routes>
            <Route path="/" element={<Login />} />
            <Route path="/home" element={<Home />} />
            <Route path="/articulos" element={<Articulos
               />}/>
            <Route path="/datos" element={<Datos/>}/>
            <Route path="/articulos" element={<Articulos
               />}/>
            <Route path="/venta" element={<Venta/>}/>
          </Routes>
        </Router>
      </ProviderUser>
  );
}

export default App;
```

Lo primero, se utilizo el apartado de router para referenciar distintas vistas las cuales puede ver el usuario y transportarse en ellas, después importaciones de las distintas paginas que utilizamos para estas direcciones. De forma importante utilizamos un contexto que se utilizar para almacenar información del usuario y todas estas páginas que encierra el ProviderUser. Este se utiliza por medios de hooks el cual son métodos para la utilización

de esta información guardada en el contexto. Parte ProviderUser.tsx

```
    interface Usuario {
      id: number;
      RFC: string;
      Fecha_Ing: string;
      Curp: string;
      email: string;
      nombre: string;
      Paterno: string;
      Materno: string;
      id_gerente: number;
      clave_sucursal: string;
      tipo: 'Cajero' | 'vendedor' | 'limpieza' | 'seguridad' |
          'administrativo';
      Colonia: string;
      Calle: string;
      Numero: number;
      Codigo_Postal: number;
      Estado: string;
    }
    interface Acceso { validacion: boolean; }
```

Así va a guardar la información del usuario en el provider, esto da el contexto de nuestra información. La parte de hooks: useUser.tsx

```
    const { data, setData } = useProviderUser();
```

Se le pasa este contexto que va a ser nuestra data y la modificación de esta. Como primer método tenemos el añadir al usuario al contexto

```
    const addUser = (item: AuthResponse) => {
        setData(item); // ejemplo: addUser([{ acceso: true },
            usuario])
      };
```

Retorna todo el usuario, con la validación y su información.

```
    const showUser = (): AuthResponse => {
        return data;
    };
```

Se elimina el contexto, puede servir para un cierre de sesión.

```
const eliminateUser = () => {
```

```
    setData([{ validacion: false }, null]); // resetea el
        contexto
  };
```

Se utiliza para extraer el tipo de usuario para el navbarmenu.

```
const getTipoUsuario = (): Usuario["tipo"] | null => {
    return data[1]?.tipo ?? null;
  };
```

Este método se utiliza para obtener el nombre y pegarlo en el inicio.

```
const getNombreCompleto = (): string | null => {
  if (!data[1]) return null;
  return `${data[1].nombre} ${data[1].Paterno} ${data[1].
     Materno}`;
  };
```

Se obtiene el id para ver la jerarquía.

```
    return {
    addUser,
    showUser,
    eliminateUser,
    modificarUser,
    getTipoUsuario,
    getNombreCompleto,
    getId
  };
```

Al final se retornan todas estos métodos

```
return {
    addUser,
    showUser,
    eliminateUser,
    modificarUser,
    getTipoUsuario,
    getNombreCompleto,
    getId
  };
```

6.7.1. Pages

Login.tsx

Se utilizo un form para el cuestionario de login, todo esto estilado con styled-componts, se mostrara un único ejemplo para no hacer tan larga la documentación.

```
<NewForm onSubmit={handleLogin}>
        <Title>Iniciar sesion</Title>
        <Input
          type="number"
          max="99999"
          placeholder="ID EMPLEADO"
          value={Id_Emp}
          onChange={(e) => setId_Emp(e.target.value)}
          required
        />
        <Input
          type="password"
          placeholder="Contrasena"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
          required
        />
        <Button type="submit">Enviar</Button>
      </NewForm>
```

Ejemplo de utilizar el styled componts

```
const NewForm = styled.form`
  display: flex;
  flex-direction: column;
  gap: 1rem;
  background-color:rgb(219, 219, 219);
  padding: 2rem;
  border-radius: 1rem;
  box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);
  width: 100%;
  max-width: 400px;
`;
```

El botón llama a esta función una vez que sea clickeado. Esta función es una petición a la API con axios, esta biblioteca se encarga de hacer esta petición en forma de link y pasas los datos al igual de recibirlos.

```
const handleLogin = async (e: { preventDefault: () => void; }) => {
    e.preventDefault();
    try {
      const response =await axios.post('http://localhost:8000/login', {
        Id_Emp,
        password
      })
      const [authData, userData] = response.data;
      if(authData.validacion){
        addUser([authData, userData]);
        navigate("/home");
      }else{
        alert("X Credenciales incorrectas. Verifica tu ID y contrasena.");
      }
    } catch (error) {
      console.error("Error al enviar los datos", error);
      alert("X Conexion invalida. Veridica tu conexion.");
    };
  };
```

Una vez validada se cambia a /come y se guarda el usuario en el contexto. Home.tsx

Se utiliza como vemos un método del hook, el cual es getNombreCompleto.

```
const {getNombreCompleto} = useUser();
  const nombre = getNombreCompleto();

  return (
    <MainCointeiner>
      <NabvarLateral/>
      <ContentContainer>
        <h1>Bienvenido</h1>
```

```
        <p>{nombre}</p>
      </ContentContainer>
    </MainCointeiner>
  );
```

Como siguiente se implementa otro componente que se exporta de otra carpeta, pero es un componente no una vista. NavbarLateral.tsx Por temas de abstracción este se divide en tres componentes distintos los cuales se pueden modificar.

```
return (
    <>
        <NewNavbarLateral>
            <HeaderLateral/>
            <MenuLateral/>
            <FooderLateral/>
        </NewNavbarLateral>
    </>
  );
```

HeaderLateral.tsx
Aquí solo se tiene la imagen

```
return (
    <Navbardiv>
      <ImgHeader src="https://cdn.pixabay.com/photo
         /2024/05/31/18/54/meme-8801100_960_720.png" />
    </Navbardiv>
  );
```

MenuLateral.tsx
Aquí se utiliza para mostrar las opciones las cuales van a varias respecto al tipo de usuario

```
const navigate = useNavigate();

    const {getTipoUsuario}=useUser();
    const tipo = getTipoUsuario();

    const opciones = tipo ? menuConfig[tipo] : [];
```

```
7
8       return(
9           <MenuItemDiv>
10              <Conteiner>
11                  {opciones.map(({ label, path}, idx) => (
12                      <LinkItem key={idx} onClick={() =>
                            navigate(path)}>
13              {label}
14            </LinkItem>
15                  ))}
16              </Conteiner>
17          </MenuItemDiv>
18
19      );
```

Para el tipo de usuario se tiene

```
1  const menuConfig = {
2    vendedor: [
3      { label: "Inicio", path: "/home" },
4      { label: "Datos", path: "/datos" },
5      { label: "Articulos", path: "/articulos"},
6    ],
7    Cajero: [
8      { label: "Inicio", path: "/home" },
9      { label: "Datos", path: "/datos" },
10     { label: "Venta", path: "/venta"}
11   ],
12   administrativo: [
13     { label: "Reportes", path: "/Reportes" },
14   ],
15   limpieza: [],
16   seguridad: [],
17 };
```

FooderLateral.tsx

Este solo se ocupa para poner un botón que indique el cierre de sesión

```
1  const navigate = useNavigate();
2    const { eliminateUser } = useUser();
3
4    const onClick = () => {
```

```
    eliminateUser();
    navigate("/");
  };

  return (
    <FooderCointeiner>
      <Button onClick={onClick}>Cerrar Sesion</Button>
    </FooderCointeiner>
  );
```

Otra vez para las páginas:

Datos.tsx

Aquí vamos a extraer los datos y los vamos a colocar en cards y mostrarlo, se le añadió también la función de obtener la jerarquía de jefes. Esta jerarquía de jefes se obtiene mediante otra petición a la API.

```
 const [, usuario] = useUser().showUser();
  const {getId} = useUser();
  const [jefes, setJefes] = useState([]);
  const handleConectarAPI = async () => {
    try {
      const res = await axios.post("http://localhost:8000/
         jerarquia", {
        id_empleado: getId(),
      });
      setJefes(res.data);
    } catch (error) {
      console.error("X Error:", error);
      alert("X Fallo la peticion");
    }
  };

  if (!usuario) return null;

  return (
    <MainContainer>
      <NabvarLateral />
      <ContentContainer>
        <h1>Datos del Usuario</h1>
        <GridDatos>
```

```
            {Object.entries(usuario)
              .filter(([, valor]) => valor !== null)
              .map(([clave, valor]) => (
                <DatoBox key={clave}>
                  <h3>{formatLabel(clave)}</h3>
                  <p>{String(valor)}</p>
                </DatoBox>
              ))}
          </GridDatos>
          <div style={{ textAlign: "center" }}>
            <BotonConectar onClick={handleConectarAPI}>
              Jefes
            </BotonConectar>
          </div>
          {jefes.length > 0 && (
            <>
              <h2 style={{ textAlign: "center", margin: "3rem 0
                 1.5rem" }}>Jefes Jerarquicos</h2>
              <GridDatos>
                {jefes.map((jefe: any, index) => (
                  <DatoBox key={index}>
                    <h3>Clave</h3>
                    <p>{jefe.clave}</p>
                    <h3>Nombre</h3>
                    <p>{jefe.nombre}</p>
                    <h3>Nivel</h3>
                    <p>{jefe.nivel}</p>
                  </DatoBox>
                ))}
              </GridDatos>
            </>
          )}
        </ContentContainer>
      </MainContainer>
    );
```

Articulos.tsx

Aquí se hace otra petición al API para obtener los datos de articulo he imprimirlos igual en las cards que imprime los datos con todo lo necesario.

```
const [Art, setArt] = useState([]);

  useEffect(() => {
    const fetchArticulos = async () => {
      try {
        const response = await axios.post('http://localhost
           :8000/art');
        setArt(response.data);
      } catch (error) {
        console.error("X Error al cargar articulos:", error);
      }
    };

    fetchArticulos();
  }, []);

  return (
    <MainContainer>
      <NabvarLateral />
      <ContentContainer>
        <h1>Catalogo de Articulos</h1>
        <GridDatos>
          {Art.map((e, idx) => (
            <DatoBox key={idx}>
              <h3>{e["Nombre"] || "Sin nombre"}</h3>
              {e["Codigo de Barras"] && <p><strong>Codigo:</
                 strong> {e["Codigo de Barras"]}</p>}
              {e["Precio de compra"] && <p><strong>Compra:</
                 strong> ${e["Precio de compra"]}</p>}
              {e["Precio de venta"] && <p><strong>Venta:</
                 strong> ${e["Precio de venta"]}</p>}
              {e["Stock"] != null && <p><strong>Stock:</strong
                 > {e["Stock"]}</p>}
              {e["Categoria"] && <p><strong>Categoria:</strong
                 > {e["Categoraa"]}</p>}
              {e["Nombre categoria"] && <p><strong>Nombre
                 Categoria:</strong> {e["Nombre categoria"]}</
                 p>}
              {e["Tipo categoria"] && <p><strong>Tipo
                 categoria:</strong> {e["Tipo categoria"]}</p
```

```
                >}
32            </DatoBox>
33          ))}
34        </GridDatos>
35      </ContentContainer>
36    </MainContainer>
37  );
```

Venta.tsx

La venta necesita varias funciones de validación que checa si existe ese artículo extrañándolo igual con el API, si no existe no lo añade.

```
1  const handleAgregar = (e: React.FormEvent) => {
2      e.preventDefault();
3
4      const articulo = articulos.find(a => a["Codigo de Barras"]
           === codigoBarras);
5
6      if (!articulo) {
7        alert("X Codigo no encontrado");
8        return;
9      }
10
11     if (cantidad <= 0 || cantidad > articulo.Stock) {
12       alert(`X Cantidad invalida. Stock disponible: ${articulo
             .Stock}`);
13       return;
14     }
15
16     const yaExiste = seleccionados.find(item => item.
          cod_barras === codigoBarras);
17
18     if (yaExiste) {
19       setSeleccionados(prev =>
20         prev.map(item =>
21           item.cod_barras === codigoBarras
22             ? { ...item, cantidad: item.cantidad + cantidad }
23             : item
24         )
25       );
26     } else {
```

```
      setSeleccionados(prev => [
        ...prev,
        {
          cod_barras: articulo["Codigo de Barras"],
          nombre: articulo.Nombre,
          cantidad,
          precio: articulo["Precio de venta"]
        }
      ]);
    }

    setCodigoBarras("");
    setCantidad(1);
  };
```

Si existe el articulo solo aumenta la cantidad, esto lo guarda en un estado que es seleccionados. De esta vista este es el método esencial que se conecta con las base de datos.

```
const handleCompra = async () => {
  const compra = {
    rfc_cliente: rfcCliente,
    id_empleado: Number(idEmpleado),
    id_cajero: getId(),
    articulos: seleccionados
  };

  try {
    const response = await fetch("http://localhost:8000/vent",
        {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(compra),
    });

    const data = await response.json();

    if (response.ok) {
      console.log("?? Compra realizada:", data);
```

```
22        alert(" Compra registrada correctamente.");
23        // Limpiar estados solo si fue exitoso
24        setSeleccionados([]);
25        setIdEmpleado("");
26        setRfcCliente("");
27      } else {
28        console.error("X Error al registrar compra:", data);
29        alert("X Error al registrar la compra. Revisa la consola
              .");
30      }
31    } catch (error) {
32      console.error("X Error de red o servidor:", error);
33      alert("X Fallo de conexion con el servidor.");
34    }
35  };
```

Este método es el que realiza la compra, la conecta con el back end y finalmente la guarda en nuestra base de datos.

7. Conclusiones

Jimenez Elizalde Josué: Durante el desarrollo del proyecto, mi principal responsabilidad fue la elaboración del Modelo Relacional. Esta tarea implicó un análisis detallado de las entidades, atributos y relaciones presentes en el modelo Entidad-Relación, así como su correcta traducción a una estructura relacional coherente y funcional.

Uno de los principales retos a los que me enfrenté fue la adecuada interpretación de los tipos de datos, su capacidad y la correcta normalización de las tablas para garantizar integridad, eficiencia y escalabilidad. Sin embargo, el modelo Entidad-Relación proporcionó una base clara sobre la cual construir, ya que en él se encontraban representadas de forma explícita las relaciones y atributos necesarios para el diseño del modelo lógico.

En resumen, esta experiencia me permitió consolidar mis conocimientos sobre el diseño estructurado de bases de datos y comprender la importancia de una buena planificación previa al momento de implementar modelos relacionales en un sistema real.

Medina Guzmán Santiago: A lo largo del desarrollo de este proyecto desempeñé el rol de tester, asumiendo con la responsabilidad de validar la correcta implementación de las funcionalidades solicitadas y diseñadas en la base de datos. Este rol implicó la ejecución de una amplia variedad de pruebas para verificar la integridad de los datos, la coherencia de las relaciones entre tablas y el comportamiento esperado de funciones, triggers y restricciones definidas en el sistema.

Utilizar la biblioteca `psycopg2` en Python me permitió conectar al manejador `PostgreSQL` de la base de datos desde dicho lenguaje, pudiendo llevar a cabo una correcta extracción de datos y así, llevar a cabo una exitosa elaboración y distintos escenarios de pruebas del dashboard verificando el cumplimiento de las reglas de negocio defini-

das.

Asumir este rol fortaleció mi capacidad analítica apreciando la suma importancia de las pruebas como un componente importante que no se puede pasar por alto, garantizando la robustez y fiabilidad de cualquier sistema de bases de datos.

Nava Benítez David Emilio: A lo largo del desarrollo del proyecto, asumí un rol integral que abarcó desde la etapa inicial de análisis de requerimientos hasta la elaboración del documento final. Una de mis principales responsabilidades fue realizar el levantamiento y análisis detallado de los requerimientos del sistema, tarea que representó un reto importante al implicar una comprensión profunda de las necesidades del cliente y su correcta traducción a un lenguaje técnico claro y funcional para el equipo de desarrollo. Esta fase fue clave para garantizar una base sólida sobre la cual construir todo el sistema.

Uno de los mayores retos fue la compilación del documento final del proyecto. Esta actividad requirió una revisión minuciosa de todas las aportaciones técnicas: plan de trabajo, diseño conceptual y lógico, implementación de funciones, triggers, validaciones, y pruebas. Para lograr una documentación exitosa, fue necesario analizar cada componente, comprenderlo a profundidad y presentarlo de manera ordenada, clara y profesional. Este proceso, aunque demandante, me permitió consolidar mis habilidades de síntesis, redacción técnica y coordinación.

En términos generales, considero que el proyecto representó una valiosa experiencia de aprendizaje, ya que no solo reforcé mis conocimientos técnicos, sino también habilidades transversales como la comunicación efectiva con el equipo, la organización de información compleja y la capacidad de liderar una documentación integral que refleje con precisión el trabajo realizado por todos los integrantes.

Rodriguez Zacarias Ivan: En el desarrollo del proyecto, asumí principalmente la responsabilidad de construir la base de datos dentro del sistema gestor PostgreSQL. Esto implicó trasladar las entidades del modelo lógico al lenguaje SQL, permitiendo su implementación y posterior manipulación a través de funciones especializadas diseñadas para facilitar el control y la integridad de la información.

La transformación del modelo relacional al conjunto de tablas no representó una dificultad significativa, gracias a la base teórica proporcionada en clase y su aplicación práctica durante las sesiones de laboratorio. No obstante, fue necesario en distintas ocasiones consultar la documentación oficial de PostgreSQL, particularmente para afinar aspectos relacionados con la creación de relaciones entre tablas, como la correcta declaración de llaves foráneas y restricciones de integridad.

Esta experiencia reforzó mi comprensión de los principios del diseño de bases de datos y me permitió desarrollar soluciones funcionales y estructuradas que dieron soporte a otros componentes del sistema.

Santiago Estrada Samantha: Durante el desarrollo del proyecto, me enfoqué principalmente en la elaboración del Modelo Entidad-Relación (MER), el cual constituye uno de los pilares fundamentales para una correcta estructuración de la base de datos. Esta actividad implicó un análisis profundo de los requerimientos del sistema, con el objetivo de representar fielmente las entidades, atributos y relaciones que dieran soporte a los procesos definidos por el cliente.

Uno de los principales retos a los que me enfrenté fue la correcta interpretación de los datos, su tipo, cardinalidad y la normalización de las relaciones para evitar redundancias y asegurar la integridad. Además, fue fundamental mantener una comunicación constante con el equipo para validar la coherencia del modelo con respecto a la lógica del negocio y asegurar que el diseño propuesto fuera comprensible y funcional para los desarrolladores encargados de su implementación.

Tavera Castillo David Emmanuel: Durante el desarrollo del proyecto, asumí la responsabilidad de diseñar funciones y triggers en la base de datos. Al inicio, uno de los mayores retos fue comprender la sintaxis específica de PL/pgSQL y cómo estructurar correctamente las funciones para que cumplieran con los requerimientos del sistema. Este desafío implicó una revisión constante de la documentación oficial, así como la realización de múltiples pruebas y ajustes hasta alcanzar los resultados esperados.

A pesar de la complejidad inicial, se logró implementar funciones que optimizan tareas clave del sistema, mejorando la modularidad y reutilización del código. Además, se desarrollaron triggers que permiten automatizar procesos críticos, como validaciones y actualizaciones automáticas, lo cual redujo la posibilidad de errores manuales y mejoró la integridad de los datos.

Gracias a este trabajo, la base de datos no solo se volvió más eficiente en términos de desempeño, sino también más confiable en su operación diaria, sentando una base sólida para futuras mejoras o escalabilidad del sistema.