Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

ADMINISTRACIÓN Y DISEÑO DE BASES DE DATOS

Tiendas Videojuegos (GameLand)

Gabriel Jonay Vera Estévez
(alu0101398198@ull.edu.es)
Daniel González de Chaves González
(alu0101407951@ull.edu.es)
Muhammad Campos Preira
(alu0101434025@ull.edu.es)

Índice

1. Introducción

El presente informe detalla el diseño y la implementación de una base de datos para la gestión integral de una cadena de tiendas de productos relacionados con el mundo gaming. Este proyecto tiene como objetivo principal crear un sistema eficiente y robusto que abarque desde la gestión de contratos y proveedores hasta la interacción entre empleados, clientes y el control exhaustivo del inventario.

La cadena de tiendas opera en un entorno dinámico, donde la gestión eficiente de la información se convierte en un factor clave para el éxito del negocio. La base de datos propuesta aborda una amplia gama de requisitos, desde la gestión de productos hasta el manejo de ventas y devoluciones, garantizando la integridad y consistencia de los datos.

A lo largo del informe, se explicará en detalle el modelo entidad-relación, el modelo relacional resultante, la implementación en PostgreSQL, así como la creación de una interfaz de programación de aplicaciones (API) que proporciona diversos endpoints para interactuar con la base de datos.

Con este informe, se busca proporcionar una visión integral de la solución propuesta, destacando su coherencia con los requisitos del proyecto y su capacidad para adaptarse a un entorno empresarial en constante evolución.

1.1 Contexto de la base de datos

Se propone el desarrollo de una base de datos para una cadena de tiendas de productos relacionados con el gaming que operan con diversos proveedores. La tienda gestiona contratos con proveedores para la adquisición de productos como videojuegos, merchandising y periféricos. Los empleados de las tiendas se encargan de la venta de productos a los clientes, quienes pueden devolver productos solo si han sido comprados previamente. Los clientes pueden disponer de una ficha de cliente con la cual se puede identificar una compra, en caso de no tenerla se tendría que crear un identificador único de cliente para esa compra.

2. Objetivos

1. Crear una Base de Datos para la Gestión de una Tienda Gaming

- Diseñar un modelo entidad-relación que refleje de manera precisa las entidades y relaciones involucradas en la operación de una tienda de estas características-
- Implementar un modelo relacional que traduzca el diseño conceptual a una estructura de base de datos con tablas y relaciones.

2. Establecer un Sistema de Gestión de Contratos y Proveedores:

- Registrar contratos entre la tienda y los proveedores, para poder llevar un control de los acuerdos establecidos.
- Almacenar información detallada sobre los proveedores, incluyendo datos de contacto

3. Gestionar el Inventario y las Relaciones con los Productos:

- Rastrear los productos suministrados por los proveedores y disponibles en las tiendas.
- Establecer relaciones entre proveedores y productos, permitiendo una gestión eficiente del inventario.

4. Administrar la Interacción entre Empleados, Clientes y Ventas:

- Registrar la relación laboral entre empleados y tiendas, especificando roles y responsabilidades.
- Capturar las transacciones de ventas entre empleados, clientes y productos, incluyendo devoluciones y compras.

5. Diferenciar entre Tipos de Productos:

- Clasificar los productos en categorías específicas, como videojuegos, merchandising y periféricos.
- Gestionar de manera eficiente los diferentes tipos de productos en la base de datos.

6. Implementar un Sistema de Devoluciones y Compras:

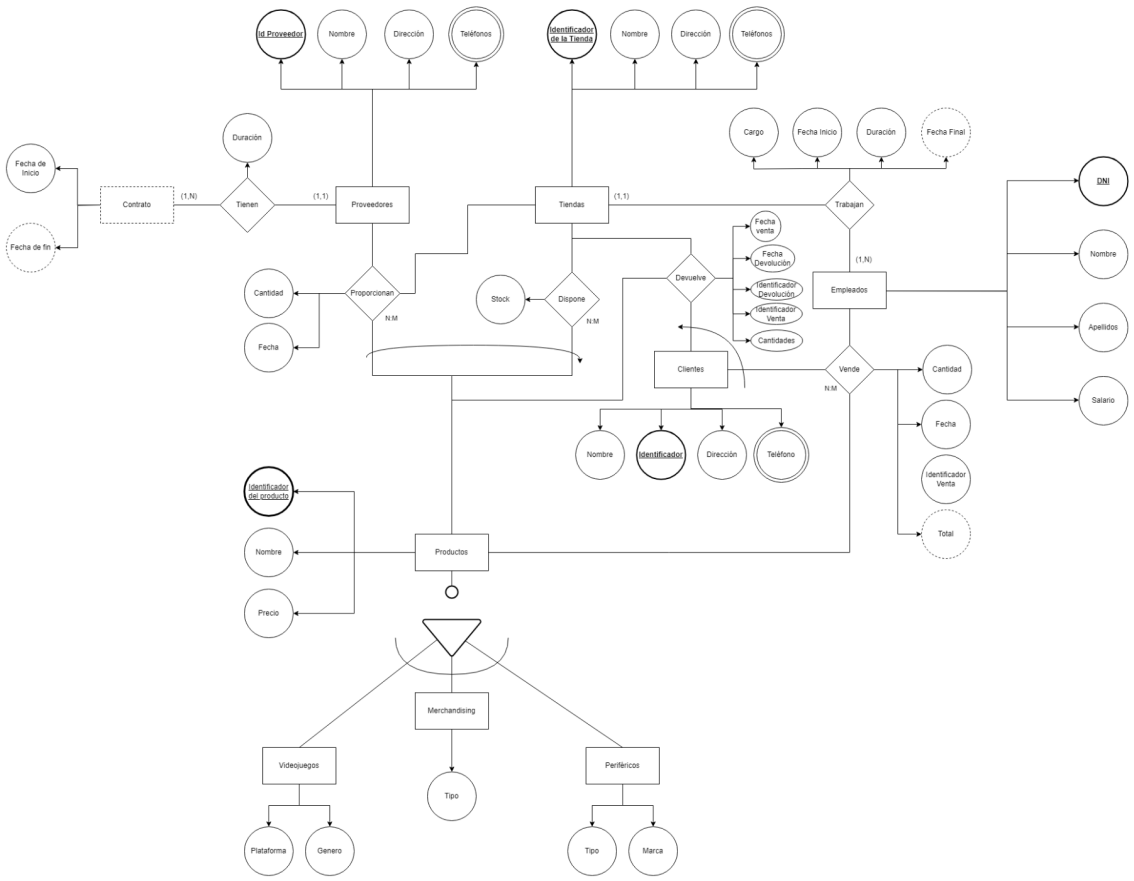
- Comprobar que un cliente haya comprado un producto antes de poder devolverlo.
- Registrar devoluciones y gestionar adecuadamente el estado del inventario después de una transacción.

7. Proporcionar un Conjunto de Endpoints para la Interfaz de Programación de Aplicaciones (API):

- Desarrollar una API que permita la interacción con la base de datos a través de varios endpoints.
- Incluir funcionalidades como consulta de productos, registro de ventas, información de clientes y gestión de inventario.

3. Diseño modelo entidad-relación

Acorde a la contextualización y los requisitos establecidos, la propuesta de modelo entidad-relación es la siguiente:



Podemos observar las siguientes entidades:

- Contrato:
  La entidad "Contrato" representa los acuerdos formales establecidos entre la tienda y los proveedores. Contiene información sobre la duración del contrato, las fechas de inicio y finalización, así como los términos específicos del acuerdo comercial.

- Proveedores:

La entidad "Proveedores" identifica a las empresas o individuos que suministran productos a la tienda. Contiene detalles como el nombre del proveedor, información de contacto y la relación con los contratos establecidos.

- Tiendas:
  La entidad "Tiendas" representa las diferentes ubicaciones físicas de la cadena de tiendas de productos electrónicos. Incluye información como la identificación de la tienda o ubicación.

- Empleados:
  La entidad "Empleados" identifica a los individuos empleados por la tienda. Contiene detalles como el nombre del empleado, su identificación única y el rol que desempeña en la tienda.

- Clientes:
  La entidad "Clientes" representa a los individuos que realizan compras en la tienda. Incluye información como el nombre del cliente y detalles de contacto.

- Productos:
  La entidad "Productos" abarca todos los productos disponibles en la tienda de productos electrónicos. Incluye detalles como el nombre del producto y su precio.

- Videojuegos:
  La entidad "Videojuegos" es una subentidad de "Productos" y contiene detalles específicos sobre los videojuegos, como la plataforma y el género.

- Merchandising:
  La entidad "Merchandising" es una subentidad de "Productos" y clasifica los productos relacionados con el merchandising, especificando el tipo (ropa, accesorios, etc.).

- Periféricos:
  La entidad "Periféricos" es una subentidad de "Productos" y clasifica los productos relacionados con periféricos electrónicos, especificando el tipo (teclado, ratón, etc.) y su marca (asus, logitech, etc.)

Las relaciones establecidas son las siguientes:

- Contrato-Proveedor:
  Esta relación establece la conexión entre la entidad "Contrato" y "Proveedores". Un contrato está asociado a un proveedor, especificando las fecha de vigencia del contrato.

- Proveedor-Producto-Tienda:

La relación "Proveedor-Producto-Tienda" vincula la entidad "Productos" con "Proveedores" y “Tienda”. Indica qué proveedor suministra qué productos a qué tienda. Esta relación es crucial para rastrear el origen de los productos en el inventario y facilitar la gestión de inventario y pedidos a los proveedores.

- Tienda-Producto
  Esta relación vincula las entidades "Tiendas" y "Productos". Esta es esencial para gestionar el inventario de productos en cada tienda específica. Contiene información sobre la cantidad de productos disponibles en una tienda particular, permitiendo un control preciso del stock.

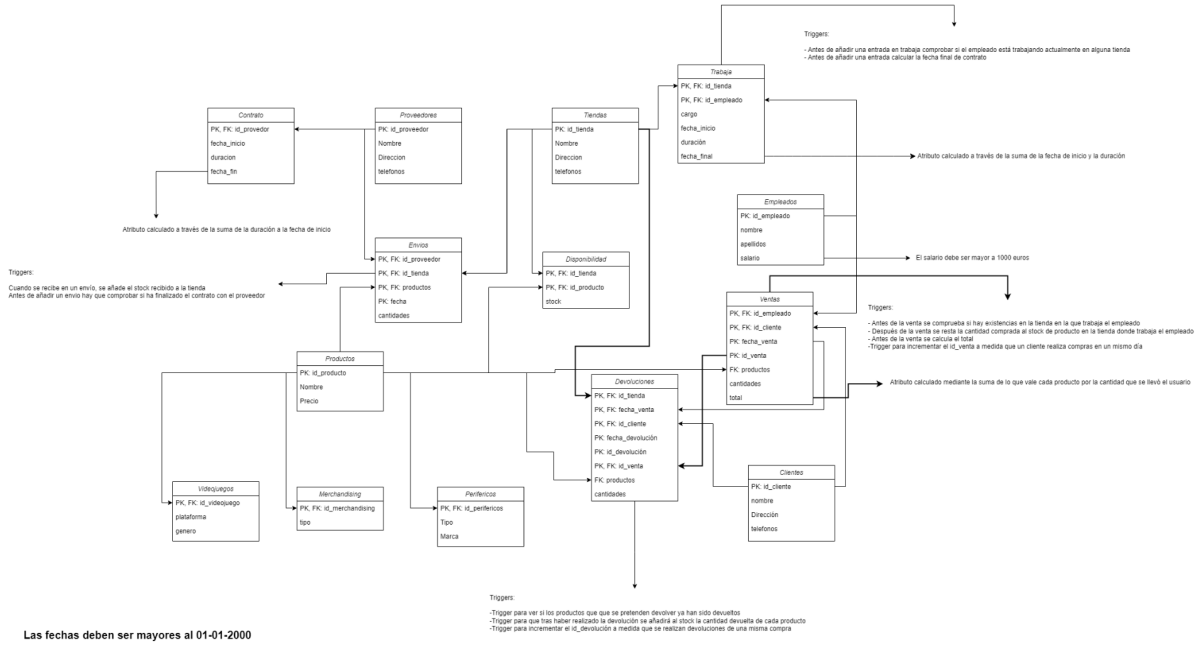
- Empleado-Tienda:
  La relación "Empleado-Tienda" establece la conexión entre las entidades "Empleados" y "Tiendas". Indica en qué tienda trabaja cada empleado y su rol dentro de la empresa, permitiendo estructurar la información de manera clara y eficiente.

- Empleado-Producto-Cliente:
  La relación "Venta" conecta las entidades "Empleados", "Clientes" y "Productos". Representa las transacciones de venta realizadas por los empleados a los clientes e incluye detalles sobre los productos vendidos. Además, se utiliza el registro de estas transacciones para asegurar que solo los productos comprados previamente puedan ser devueltos.

- Cliente-Producto-Tienda:
  Relación que conecta las entidades "Clientes", "Productos" y "Tiendas". Representa el proceso de devolución de productos por parte de los clientes a una tienda específica. Contiene detalles sobre la transacción de devolución, incluyendo el cliente que realiza la devolución, el producto devuelto y la tienda donde se efectúa la devolución.

4. Grafo relacional

Tras la realización del modelo relacional, hemos obtenido el siguiente resultado:



Por cada entidad que teníamos en el modelo entidad-relación hemos añadido una tabla:

- Contrato:
  Clave Principal: id_proveedor
  Clave Foránea: id_proveedor → Proveedor(id_proveedor)

  Triggers:
  - Fecha final es un atributo calculado a través de la suma de la duración a la fecha de inicio

- Proveedores:
  Clave Principal: id_preveedor

- Tiendas:
  Clave Principal: id_tienda

- Empleados:
  Clave Principal: id_empleado

  Restricciones: el salario debe ser mayor a 1000 euros.

- Clientes:
  Clave Principal: id cliente

- Productos:
  Clave Principal: id producto

- Videojuegos:
  Clave Principal: id videojuego
  Clave Foránea: id_videojuego → Productos(id_producto)

- Merchandising:
  Clave Principal: id_merchandising
  Clave Foránea: id_merchandising → Productos(id_producto)

- Periféricos:
  Clave Principal: id_perifericos
  Clave Foránea: id_perifericos → Productos(id_producto)

Todas las relaciones menos la de Proveedor-Contrato se han transformado en tablas nuevas, esto permite una mejor comunicación y claridad en la base de datos. Estas son las nuevas tablas provenientes de las relaciones:

- Trabaja:
  Clave Principal: id_tienda, id_empleado
  Clave Foránea: id_tienda → Tiendas(id_tienda), id_empleado → Empleados(id_empleado)

  Triggers:
  - Antes de añadir una entrada en trabaja comprobar si el empleado está trabajando actualmente en alguna tienda
  - Antes de añadir una entrada calcular la fecha final de contrato

- Disponibilidad:
  Clave Principal: id_tienda, id_producto
  Clave Foránea: id_tienda → Tiendas(id_tienda), id_producto → Productos(id_producto)

- Ventas:
  Clave Principal: id_empleado, id_cliente, fecha_venta, id_venta
  Clave Foránea: id_empleado → Empleados(id_empleado), id_cliente → Clientes(id_cliente), productos → Productos(id_producto)

  Triggers:

- Antes de la venta se comprueba si hay existencias en la tienda en la que trabaja el empleado
- Después de la venta se resta la cantidad comprada al stock de producto en la tienda donde trabaja el empleado
- Antes de la venta se calcula el total
- Trigger para incrementar el id_venta a medida que un cliente realiza compras en un mismo día

- Devoluciones:
  Clave Principal: id_tienda, fecha_venta, id_cliente, fecha_devolucion, id_devoluvion, id_venta
  Clave Foránea: id_tienda → Tiendas(id_tienda), productos → Productos(id_producto),
  (fecha_venta, id_venta → Ventas(fecha_venta, id_venta), id_cliente → Clientes(id_cliente)

  Triggers:

  - Trigger para ver si los productos que que se pretenden devolver ya han sido devueltos
  - Trigger para que tras haber realizado la devolución se añadirá al stock la cantidad devuelta de cada producto
  - Trigger para incrementar el id_devolución a medida que se realizan devoluciones de una misma compra

- Envíos:
  Clave Principal: id_proveedor, id_tienda, id_producto, fecha
  Clave Foránea: id_proveedor → Proveedores(id_proveedor),
  id_tienda → Tiendas(id_tienda), id_producto → Productos(id_producto)

  Triggers:
  - Cuando se recibe en un envío, se añade el stock recibido a la tienda
  - Antes de añadir un envío hay que comprobar si ha finalizado el contrato con el proveedor

5. Base de datos en postgreSQL

En cuanto a la base de datos hemos generado las siguientes tablas con sus respectivos comprobadores de tipos y valores, como podremos observar los atributos y claves principales o foráneas se corresponden a las tablas del grafo relacional. También veremos posteriormente los diferentes triggers que hemos creado para el esquema de tablas.

Tabla Productos:

```
Unset
CREATE TABLE Productos (
   id_producto VARCHAR(50) PRIMARY KEY,
   nombre VARCHAR(50) NOT NULL,
   precio NUMERIC(10,2) NOT NULL
);
```

Todos los atributos son obligatorios (NOT NULL). La clave principal es id_producto.

Tabla Videojuegos

```
Unset
CREATE TABLE Videojuegos (
   id_videojuego VARCHAR(50) PRIMARY KEY REFERENCES Productos(id_producto) ON DELETE
CASCADE,
   plataforma VARCHAR(50) NOT NULL,
   genero VARCHAR(50) NOT NULL
);
```

Todos los atributos son obligatorios. La clave id_videojuego hace referencia a la clave principal de la tabla Productos, con la característica ON DELETE CASCADE.

Tabla Merchandising

```
Unset
CREATE TABLE Merchandising (
   id_merchandising VARCHAR(50) PRIMARY KEY REFERENCES Productos(id_producto) ON
DELETE CASCADE,
   tipo VARCHAR(50) NOT NULL
);
```

Todos los atributos son obligatorios. La clave id_merchandising hace referencia a la clave principal de la tabla Productos, con la característica ON DELETE CASCADE.

Tabla Periféricos

```
Unset
CREATE TABLE Periféricos (
   id_periféricos VARCHAR(50) PRIMARY KEY REFERENCES Productos(id_producto) ON DELETE
CASCADE,
   tipo VARCHAR(50) NOT NULL,
   marca VARCHAR(50) NOT NULL
);
```

Todos los atributos son obligatorios. La clave id_periféricos hace referencia a la clave principal de la tabla Productos, con la característica ON DELETE CASCADE.

Tabla Proveedores

```
Unset
CREATE TABLE Proveedores (
    id_proveedor VARCHAR(50) PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    direccion VARCHAR(50) NOT NULL,
    telefonos VARCHAR(50)[] NOT NULL
);
```

Todos los atributos son obligatorios. La clave principal es id_proveedor.

Tabla Contrato

```
Unset
CREATE TABLE Contrato (
    id_provedor VARCHAR(50) REFERENCES Proveedores(id_proveedor) ON DELETE CASCADE,
    fecha_inicio DATE CHECK (fecha_inicio >= DATE '2000-01-01') NOT NULL,
    fecha_fin DATE,
    duracion INT NOT NULL,
    PRIMARY KEY (id_provedor)
);
```

Todos los atributos son obligatorios, excepto fecha_fin. La clave id_provedor hace referencia a la clave principal de la tabla Proveedores, con la característica ON DELETE CASCADE. Hay una restricción (CHECK) en el atributo fecha_inicio para asegurar que sea mayor o igual al 1 de enero de 2000.

Tabla Tiendas

```
Unset
CREATE TABLE Tiendas (
    id_tienda VARCHAR(50) PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    direccion VARCHAR(50) NOT NULL,
    telefonos VARCHAR(50)[] NOT NULL
);
```

Todos los atributos son obligatorios. La clave principal es id_tienda.

Tabla Envíos

```
Unset
CREATE TABLE Envios (
    id_proveedor VARCHAR(50) REFERENCES Proveedores(id_proveedor) ON DELETE CASCADE,
    id_tienda VARCHAR(50) REFERENCES Tiendas(id_tienda) ON DELETE CASCADE,
    productos VARCHAR(50)[] NOT NULL,
    fecha DATE CHECK (fecha >= DATE '2000-01-01') NOT NULL,
    cantidades INT[] NOT NULL,
    PRIMARY KEY (id_proveedor, id_tienda, productos, fecha)
```

```
);
```

Todos los atributos son obligatorios. Las claves id_proveedor y id_tienda hacen referencia a las claves principales de las tablas Proveedores y Tiendas respectivamente, con la característica ON DELETE CASCADE. La clave principal de esta tabla es una combinación de id_proveedor, id_tienda, productos y fecha. Hay una restricción (CHECK) en el atributo fecha para asegurar que sea mayor o igual al 1 de enero de 2000.

Tabla Clientes

```
Unset
CREATE TABLE Clientes (
    id_cliente INT PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    direccion VARCHAR(50) NOT NULL,
    telefono VARCHAR(50) NOT NULL
);
```

Todos los atributos son obligatorios. La clave principal es id_cliente.

Tabla Disponibilidad

```
Unset
CREATE TABLE Disponibilidad (
    id_tienda VARCHAR(50) REFERENCES Tiendas(id_tienda) ON DELETE CASCADE,
    id_producto VARCHAR(50) REFERENCES Productos(id_producto) ON DELETE CASCADE,
    stock INT CHECK (stock >= 0) NOT NULL,
    PRIMARY KEY (id_tienda, id_producto)
);
```

Todos los atributos son obligatorios. Las claves id_tienda y id_producto hacen referencia a las claves principales de las tablas Tiendas y Productos respectivamente, con la característica ON DELETE CASCADE. La clave principal de esta tabla es una combinación de id_tienda y id_producto. Hay una restricción para que el stock tiene que ser mayor o igual a 0.

Tabla Empleados

```
Unset
CREATE TABLE Empleados (
    id_empleado INT PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    apellidos VARCHAR(50) NOT NULL,
    salario NUMERIC(10,2) CHECK (salario >= 1000) NOT NULL
);
```

Todos los atributos son obligatorios. La clave principal es id_empleado. Hay una restricción (CHECK) en el atributo salario para asegurar que sea mayor o igual a 1000.

Tabla Trabaja

```
Unset
CREATE TABLE Trabaja (
    id_tienda VARCHAR(50) REFERENCES Tiendas(id_tienda) ON DELETE CASCADE,
    id_empleado INT REFERENCES Empleados(id_empleado) ON DELETE CASCADE,
    cargo VARCHAR(50) NOT NULL,
    fecha_inicio DATE CHECK (fecha_inicio >= DATE '2000-01-01') NOT NULL,
    fecha_final DATE,
    duracion INT NOT NULL,
    PRIMARY KEY (id_tienda, id_empleado)
);
```

Todos los atributos son obligatorios, excepto fecha_final. Las claves id_tienda y id_empleado hacen referencia a las claves principales de las tablas Tiendas y Empleados respectivamente, con la característica ON DELETE CASCADE. La clave principal de esta tabla es una combinación de id_tienda y id_empleado. Hay una restricción (CHECK) en el atributo fecha_inicio para asegurar que sea mayor o igual al 1 de enero de 2000.

Tabla Ventas

```
Unset
CREATE TABLE Ventas (
    id_empleado INT REFERENCES Empleados(id_empleado) ON DELETE CASCADE,
    id_cliente INT REFERENCES Clientes(id_cliente) ON DELETE CASCADE,
    fecha_venta DATE CHECK (fecha_venta >= DATE '2000-01-01'),
    id_venta INT,
    productos VARCHAR(50)[] NOT NULL,
    cantidades INT[] NOT NULL,
    total NUMERIC(10,2),
    PRIMARY KEY (id_empleado, id_cliente, fecha_venta, id_venta)
);
```

Todos los atributos son obligatorios, excepto total. Las claves id_empleado y id_cliente hacen referencia a las claves principales de las tablas Empleados y Clientes respectivamente, con la característica ON DELETE CASCADE. La clave principal de esta tabla es una combinación de id_empleado, id_cliente, fecha_venta y id_venta. Hay una restricción (CHECK) en el atributo fecha_venta para asegurar que sea mayor o igual al 1 de enero de 2000.

Nota: El id_venta es un identificador que permite diferenciar las ventas realizadas en el mismo día por un mismo cliente, si no ha realizado ninguna venta se pondrá a 1, en caso contrario se aumentará.

Tabla Devoluciones

```
Unset
CREATE TABLE Devoluciones (
    id_tienda VARCHAR(50) REFERENCES Tiendas(id_tienda) ON DELETE CASCADE,
    fecha_venta DATE CHECK (fecha_venta >= DATE '2000-01-01') NOT NULL,
    id_cliente INT REFERENCES Clientes(id_cliente) ON DELETE CASCADE,
    fecha_devolucion DATE CHECK (fecha_devolucion >= DATE '2000-01-01') NOT NULL,
    id_venta INT NOT NULL,
    id_devolucion INT NOT NULL,
    productos VARCHAR(50)[] NOT NULL,
    cantidades INT[] NOT NULL,
    PRIMARY KEY (id_tienda, fecha_venta, id_cliente, fecha_devolucion, id_devolucion, id_venta)
);
```

Todos los atributos son obligatorios. Las claves id_tienda y id_cliente hacen referencia a las claves principales de las tablas Tiendas y Clientes respectivamente, con la característica ON DELETE CASCADE. La clave principal de esta tabla es una combinación de id_tienda, fecha_venta, id_cliente, fecha_devolucion, id_devolucion, id_venta. Hay dos restricciones (CHECK) en el atributo fecha_venta y fecha_devolucion para asegurar que sea mayor o igual al 1 de enero de 2000.

Nota: El id_devolucion indica el número de devolución para una misma compra, esto quiere decir que será independiente del resto de ventas.

establecer_fecha_e_id_devolucion

```
Unset
CREATE OR REPLACE FUNCTION establecer_fecha_e_id_devolucion() RETURNS TRIGGER AS $$
BEGIN
    NEW.fecha_devolucion := CURRENT_DATE;

    -- Recuperar el último el último id_devolucion de la tabla Devoluciones, si no hay ninguno se
pone a 1, si hay se incrementa en 1
    IF EXISTS (SELECT 1 FROM Devoluciones WHERE fecha_venta = NEW.fecha_venta AND
id_cliente = NEW.id_cliente) THEN
        NEW.id_devolucion := (SELECT id_devolucion FROM Devoluciones WHERE fecha_venta =
NEW.fecha_venta AND id_cliente = NEW.id_cliente ORDER BY id_devolucion DESC LIMIT 1) + 1;
    ELSE
        NEW.id_devolucion := 1;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER establecer_fecha_devolucion_trigger BEFORE INSERT ON Devoluciones FOR
EACH ROW EXECUTE FUNCTION establecer_fecha_e_id_devolucion();
```

Este trigger se activa antes de insertar en la tabla Devoluciones. Establece la fecha de devolución al día actual y calcula el id_devolucion basándose en los registros existentes.

establecer_id_venta_trigger

```
Unset
CREATE OR REPLACE FUNCTION establecer_id_venta() RETURNS TRIGGER AS $$
BEGIN
  -- Recuperar el último el último id_venta de la tabla Ventas, si no hay ninguno se pone a 1, si hay se incrementa en 1
  IF EXISTS (SELECT 1 FROM Ventas WHERE id_cliente = NEW.id_cliente AND fecha_venta = NEW.fecha_venta) THEN
    NEW.id_venta := (SELECT id_venta FROM Ventas WHERE id_cliente = NEW.id_cliente AND fecha_venta = NEW.fecha_venta ORDER BY id_venta DESC LIMIT 1) + 1;
  ELSE
    NEW.id_venta := 1;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER establecer_id_venta_trigger BEFORE INSERT ON Ventas FOR EACH ROW EXECUTE FUNCTION establecer_id_venta();
```

Este trigger se activa antes de insertar en la tabla Ventas. Calcula el id_venta basándose en los registros existentes.

establecer_fecha_venta_trigger

```
Unset
CREATE OR REPLACE FUNCTION establecer_fecha_venta() RETURNS TRIGGER AS $$
BEGIN
  NEW.fecha_venta := CURRENT_DATE;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER establecer_fecha_venta_trigger BEFORE INSERT ON Ventas FOR EACH ROW EXECUTE FUNCTION establecer_fecha_venta();
```

Este trigger se activa antes de insertar en la tabla Ventas. Establece la fecha de venta al día actual.

actualizar_stock_trigger

```
Unset
CREATE OR REPLACE FUNCTION actualizar_stock() RETURNS TRIGGER AS $$
BEGIN
  FOR i IN 1..array_length(NEW.productos, 1) LOOP
    IF EXISTS (SELECT 1 FROM Disponibilidad WHERE id_tienda = NEW.id_tienda AND id_producto = NEW.productos[i]) THEN
      UPDATE Disponibilidad SET stock = stock + NEW.cantidades[i] WHERE id_tienda = NEW.id_tienda AND id_producto = NEW.productos[i];
    ELSE
```

```
        INSERT INTO Disponibilidad VALUES (NEW.id_tienda, NEW.productos[i],
NEW.cantidades[i]);
      END IF;
  END LOOP;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER actualizar_stock_trigger AFTER INSERT ON Envios FOR EACH ROW EXECUTE
FUNCTION actualizar_stock();
```

Este trigger se activa después de insertar en la tabla Envios. Actualiza el stock de la tienda con los productos recibidos.

calcular_fecha_final_contrato_trigger

```
Unset
CREATE OR REPLACE FUNCTION calcular_fecha_final_contrato() RETURNS TRIGGER AS $$
BEGIN
  NEW.fecha_fin := NEW.fecha_inicio + NEW.duracion;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER calcular_fecha_final_contrato_trigger BEFORE INSERT ON Contrato FOR EACH
ROW EXECUTE FUNCTION calcular_fecha_final_contrato();
```

Este trigger se activa antes de insertar en la tabla Contrato. Calcula la fecha final del contrato sumando la duración a la fecha de inicio.

verificar_contrato_trigger

```
Unset
CREATE OR REPLACE FUNCTION verificar_contrato() RETURNS TRIGGER AS $$
BEGIN
  IF (SELECT fecha_fin FROM Contrato WHERE id_provedor = NEW.id_proveedor) <
CURRENT_DATE THEN
      RAISE EXCEPTION 'El contrato con el proveedor ha finalizado.';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER verificar_contrato_trigger BEFORE INSERT ON Envios FOR EACH ROW
EXECUTE FUNCTION verificar_contrato();
```

Este trigger se activa antes de insertar en la tabla Envios. Verifica si el contrato con el proveedor ha finalizado.

verificar_devolucion_trigger

Unset

```
-- Antes de procesar una devolucion se comprueba si existe una compra con los mismos datos, si no existe se lanza una excepción
-- Luego hay que comprobar producto a producto si los que se intentan devolver ya han sido devueltos, si alguno ya ha sido devuelto se lanza una excepción. Hay que tener en cuenta que puede haber comprado más de un producto
CREATE OR REPLACE FUNCTION verificar_devolucion() RETURNS TRIGGER AS $$
DECLARE
    productos_comprados RECORD;
    row RECORD;

BEGIN
    IF NOT EXISTS (SELECT 1 FROM Ventas WHERE id_cliente = NEW.id_cliente AND fecha_venta = NEW.fecha_venta AND id_venta = NEW.id_venta) THEN
        RAISE EXCEPTION 'No existe una compra con esos datos.';
    END IF;

    SELECT productos, cantidades INTO productos_comprados FROM Ventas WHERE id_cliente = NEW.id_cliente AND fecha_venta = NEW.fecha_venta AND id_venta = NEW.id_venta and id_venta = NEW.id_venta;

    -- Recorremos todos los productos que se quieren devolver y comprobamos si están en la compra original
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
        IF NOT (NEW.productos[i] = ANY(productos_comprados.productos)) THEN
            RAISE EXCEPTION 'El producto % no se compró.', NEW.productos[i];
        END IF;
    END LOOP;

    FOR row IN SELECT productos, cantidades FROM Devoluciones WHERE fecha_venta = NEW.fecha_venta AND id_cliente = NEW.id_cliente AND id_venta = NEW.id_venta LOOP
        FOR i IN 1..array_length(row.productos, 1) LOOP
            FOR j IN 1..array_length(productos_comprados.productos, 1) LOOP
                IF row.productos[i] = productos_comprados.productos[j] THEN
                    productos_comprados.cantidades[j] := productos_comprados.cantidades[j] - row.cantidades[i];
                END IF;
            END LOOP;
        END LOOP;
    END LOOP;

    -- AHORA tocar restar los new.cantidades a productos_comprados.cantidades
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
        FOR j IN 1..array_length(productos_comprados.productos, 1) LOOP
            IF NEW.productos[i] = productos_comprados.productos[j] THEN
                productos_comprados.cantidades[j] := productos_comprados.cantidades[j] - NEW.cantidades[i];
                IF productos_comprados.cantidades[j] < 0 THEN
                    RAISE EXCEPTION 'No se puede devolver más productos % de los que se compraron.', NEW.productos[i];
                END IF;
```

```
        END IF;
      END LOOP;
    END LOOP;


    RETURN NEW;
END;
$$ LANGUAGE plpgsql;



CREATE TRIGGER verificar_devolucion_trigger BEFORE INSERT ON Devoluciones FOR EACH ROW
EXECUTE FUNCTION verificar_devolucion();
```

Este trigger se activa antes de insertar en la tabla Devoluciones. Verifica si existe una compra con los mismos datos y si los productos que se intentan devolver ya han sido devueltos.

devolucion_stock_trigger

```
Unset
CREATE OR REPLACE FUNCTION devolucion_stock() RETURNS TRIGGER AS $$
BEGIN
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
        IF EXISTS (SELECT 1 FROM Disponibilidad WHERE id_tienda = NEW.id_tienda AND
id_producto = NEW.productos[i]) THEN
            UPDATE Disponibilidad SET stock = stock + NEW.cantidades[i] WHERE id_tienda =
NEW.id_tienda AND id_producto = NEW.productos[i];
        ELSE
            INSERT INTO Disponibilidad VALUES (NEW.id_tienda, NEW.productos[i],
NEW.cantidades[i]);
        END IF;
    END LOOP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER devolucion_stock_trigger AFTER INSERT ON Devoluciones FOR EACH ROW
EXECUTE FUNCTION devolucion_stock();
```

Este trigger se activa después de insertar en la tabla Devoluciones. Añade al stock la cantidad devuelta de cada producto.

verificar_stock_trigger

```
Unset
CREATE OR REPLACE FUNCTION verificar_stock() RETURNS TRIGGER AS $$
BEGIN
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
```

```
        -- Si no existe el producto en la tienda se lanza una excepción
        IF NOT EXISTS (SELECT 1 FROM Disponibilidad WHERE id_tienda = (SELECT id_tienda FROM
Trabaja WHERE id_empleado = NEW.id_empleado) AND id_producto = NEW.productos[i]) THEN
            RAISE EXCEPTION 'No existe el producto % en la tienda.', NEW.productos[i];
        END IF;

        IF (SELECT stock FROM Disponibilidad WHERE id_tienda = (SELECT id_tienda FROM Trabaja
WHERE id_empleado = NEW.id_empleado) AND id_producto = NEW.productos[i]) <
NEW.cantidades[i] THEN
        RAISE EXCEPTION 'No hay suficientes existencias en la tienda del producto %',
NEW.productos[i];
        END IF;
    END LOOP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER verificar_stock_trigger BEFORE INSERT ON Ventas FOR EACH ROW EXECUTE
FUNCTION verificar_stock();
```

Este trigger se activa antes de insertar en la tabla Ventas. Verifica si hay existencias en la tienda en la que trabaja el empleado.

venta_stock_trigger

```
Unset
CREATE OR REPLACE FUNCTION venta_stock() RETURNS TRIGGER AS $$
BEGIN
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
        UPDATE Disponibilidad SET stock = stock - NEW.cantidades[i] WHERE id_tienda = (SELECT
id_tienda FROM Trabaja WHERE id_empleado = NEW.id_empleado) AND id_producto =
NEW.productos[i];
    END LOOP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER venta_stock_trigger AFTER INSERT ON Ventas FOR EACH ROW EXECUTE
FUNCTION venta_stock();
```

Este trigger se activa después de insertar en la tabla Ventas. Resta la cantidad comprada al stock de producto en la tienda donde trabaja el empleado.

calcular_total_trigger

```
Unset
CREATE OR REPLACE FUNCTION calcular_total() RETURNS TRIGGER AS $$
BEGIN
```

```
    NEW.total := 0;
    FOR i IN 1..array_length(NEW.productos, 1) LOOP
        NEW.total := NEW.total + (SELECT precio FROM Productos WHERE id_producto =
NEW.productos[i]) * NEW.cantidades[i];
    END LOOP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER calcular_total_trigger BEFORE INSERT ON Ventas FOR EACH ROW EXECUTE
FUNCTION calcular_total();
```

Este trigger se activa antes de insertar en la tabla Ventas. Calcula el total de la venta.

verificar_empleado_trigger

```
Unset
CREATE OR REPLACE FUNCTION verificar_empleado() RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM Trabaja WHERE id_empleado = NEW.id_empleado AND
new.fecha_inicio < fecha_final) THEN
        RAISE EXCEPTION 'El empleado ya está trabajando en otra tienda.';
    END IF;

    NEW.fecha_final := NEW.fecha_inicio + NEW.duracion;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER verificar_empleado_trigger BEFORE INSERT ON Trabaja FOR EACH ROW
EXECUTE FUNCTION verificar_empleado();
```

Este trigger se activa antes de insertar en la tabla Trabaja. Verifica si el empleado está trabajando actualmente en alguna tienda y calcula la fecha final del contrato.

6. Carga de datos

En este apartado veremos las diferentes inserciones que hemos realizado para el uso y prueba de nuestra base de datos.

```
Unset
INSERT INTO Productos (id_producto, nombre, precio) VALUES ('VLOL1', 'League of Legends', 5);
INSERT INTO Productos (id_producto, nombre, precio) VALUES ('MCAM1', 'Camiseta 1', 30.5);
INSERT INTO Productos (id_producto, nombre, precio) VALUES ('PMOU1', 'Ratón 1', 76.99);
```

Se insertan tres productos con identificadores únicos: ‘VLOL1’, ‘MCAM1’ y ‘PMOU1’.

```
Unset
INSERT INTO Videojuegos (id_videojuego, plataforma, genero) VALUES ('VLOL1', 'PC', 'MOBA');
INSERT INTO Merchandising (id_merchandising, tipo) VALUES ('MCAM1', 'Camiseta');
INSERT INTO Perifericos (id_perifericos, tipo, marca) VALUES ('PMOU1', 'Ratón', 'Logitech');
```

Se clasifican los productos anteriores en tres categorías, utilizando los mismos identificadores.

```
Unset
INSERT INTO Proveedores (id_proveedor, nombre, direccion, telefonos) VALUES ('PROV1',
'Proveedor 1', 'Calle 1', ARRAY['111111111']);
INSERT INTO Proveedores (id_proveedor, nombre, direccion, telefonos) VALUES ('PROV2',
'Proveedor 2', 'Calle 2', ARRAY['222222222']);
INSERT INTO Proveedores (id_proveedor, nombre, direccion, telefonos) VALUES ('PROV3',
'Proveedor 3', 'Calle 3', ARRAY['333333333']);
```

Se insertan tres proveedores con identificadores únicos: 'PROV1', 'PROV2' y 'PROV3'.

```
Unset
INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV1', '2020-01-01', 365*5);
INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV2', '2020-01-01', 365*5);
INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV3', '2020-01-01', 365);
```

Se establecen contratos con los proveedores anteriores, utilizando los mismos identificadores.

```
Unset
INSERT INTO Tiendas (id_tienda, nombre, direccion, telefonos) VALUES ('TIENDA1', 'Tienda 1', 'Calle
1', ARRAY['111111111']);
INSERT INTO Tiendas (id_tienda, nombre, direccion, telefonos) VALUES ('TIENDA2', 'Tienda 2', 'Calle
2', ARRAY['222222222']);
```

Se insertan dos tiendas con identificadores únicos: 'TIENDA1' y 'TIENDA2'.

```
Unset
INSERT INTO Envios (id_proveedor, id_tienda, productos, fecha, cantidades) VALUES ('PROV1',
'TIENDA1', ARRAY['VLOL1', 'MCAM1'], '2020-01-01', ARRAY[10, 10]);
INSERT INTO Envios (id_proveedor, id_tienda, productos, fecha, cantidades) VALUES ('PROV2',
'TIENDA2', ARRAY['PMOU1'], '2020-01-01', ARRAY[20]);
```

Se registran los envíos de productos de los proveedores a las tiendas, utilizando los identificadores de los proveedores y las tiendas.

```
Unset
INSERT INTO Clientes (id_cliente, nombre, direccion, telefono) VALUES (45000000, 'Cliente 1', 'Calle 1', '111111111');
INSERT INTO Clientes (id_cliente, nombre, direccion, telefono) VALUES (45000001, 'Cliente 2', 'Calle 2', '222222222');
```

Se insertan dos clientes con identificadores únicos: 45000000 y 45000001.

```
Unset
INSERT INTO Empleados (id_empleado, nombre, apellidos, salario) VALUES (99999999, 'Empleado 1', 'Apellido 1', 1000);
INSERT INTO Empleados (id_empleado, nombre, apellidos, salario) VALUES (99999998, 'Empleado 2', 'Apellido 2', 1000);
```

Se insertan dos empleados con identificadores únicos: 99999999 y 99999998.

```
Unset
INSERT INTO Trabaja (id_tienda, id_empleado, cargo, fecha_inicio, duracion) VALUES ('TIENDA1', 99999999, 'Cajero', '2020-01-01', 365);
INSERT INTO Trabaja (id_tienda, id_empleado, cargo, fecha_inicio, duracion) VALUES ('TIENDA2', 99999998, 'Cajero', '2020-01-01', 365);
```

Se registra en qué tienda trabaja cada empleado, utilizando los identificadores de las tiendas y los empleados.

```
Unset
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999, 45000000, ARRAY['VLOL1', 'MCAM1'], ARRAY[1, 1]);
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999998, 45000001, ARRAY['PMOU1'], ARRAY[2]);
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999998, 45000001, ARRAY['PMOU1'], ARRAY[2]);
```

Se registran las ventas realizadas por los empleados a los clientes, utilizando los identificadores de los empleados y los clientes.

```
Unset
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades) VALUES ('TIENDA1', '2023-12-16', 45000000, 1, ARRAY['VLOL1', 'MCAM1'], ARRAY[1, 1]);
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades) VALUES ('TIENDA2', '2023-12-16', 45000001, 2, ARRAY['PMOU1'], ARRAY[1]);
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades) VALUES ('TIENDA2', '2023-12-16', 45000001, 2, ARRAY['PMOU1'], ARRAY[1]);
```

Se registran tres devoluciones de productos por parte de los clientes. Las dos primeras devoluciones tienen identificadores únicos de clientes: 45000000 y 45000001. La última devolución debería poderse insertar ya que ya existe en la tabla.

7. Consultas de ejemplo

Para probar el correcto funcionamiento vamos a realizar una serie de inserciones para probar el funcionamiento de los triggers y el comportamiento de la base de datos al borrar cierta información.

7.1 Establecer fecha e id de devolución

Cuando hacemos una inserción en la tabla devoluciones no pasamos ni el id ni la fecha de devolución, pero tras insertar se completa automáticamente.

```
Unset
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA1', '2023-12-16', 45000000, 1, ARRAY['VLOL1', 'MCAM1'], ARRAY[1, 1]);

INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA2', '2023-12-16', 45000001, 2, ARRAY['PMOU1'], ARRAY[1]);

INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA2', '2023-12-16', 45000001, 2, ARRAY['PMOU1'], ARRAY[1]);
```

Tabla Devoluciones resultante tras las inserciones:

id_tienda	fecha_venta	id_cliente	fecha_devolucion	id_venta	id_devolucion	productos	cantidades
TIENDA1	2023-12-20	45000000	2023-12-20	1	1	{VLOL1,MCAM1}	{1,1}
TIENDA2	2023-12-20	45000001	2023-12-20	2	1	{PMOU1}	{1}
TIENDA2	2023-12-20	45000001	2023-12-20	2	2	{PMOU1}	{1}

(3 rows)

7.2 Establecer fecha, id de venta y total de la compra

Cuando hacemos una inserción en la tabla ventas no pasamos el id, fecha ni total de la venta, pero tras insertar se completa automáticamente.

```
Unset
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999,
45000000, ARRAY['VLOL1', 'MCAM1'], ARRAY[1, 1]);
```

```
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999998, 45000001, ARRAY['PMOU1'], ARRAY[2]);

INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999998, 45000001, ARRAY['PMOU1'], ARRAY[2]);
```

Tabla Ventas resultante tras las inserciones:

```
 id_empleado | id_cliente | fecha_venta | id_venta |   productos   | cantidades | total
-------------+------------+-------------+----------+---------------+------------+--------
    99999999 |   45000000 | 2023-12-20  |        1 | {VLOL1,MCAM1} | {1,1}      |  35.50
    99999998 |   45000001 | 2023-12-20  |        1 | {PMOU1}       | {2}        | 153.98
    99999998 |   45000001 | 2023-12-20  |        2 | {PMOU1}       | {2}        | 153.98
(3 rows)
```

7.3 Stock

Tras la carga inicial de datos el stock es el siguiente:

```
 id_tienda | id_producto | stock
-----------+-------------+-------
 TIENDA1   | VLOL1       |    10
 TIENDA1   | MCAM1       |    10
 TIENDA2   | PMOU1       |    18
(3 rows)
```

- Tras envío:

```
Unset
INSERT INTO Envios (id_proveedor, id_tienda, productos, fecha, cantidades) VALUES ('PROV1', 'TIENDA1', ARRAY['VLOL1', 'MCAM1'], '2021-01-01', ARRAY[10, 10]);
```

Tabla tras el envío:

```
 id_tienda | id_producto | stock
-----------+-------------+-------
 TIENDA2   | PMOU1       |    18
 TIENDA1   | VLOL1       |    20
 TIENDA1   | MCAM1       |    20
(3 rows)
```

- Antes venta (verificar si hay producto suficiente):

```
Unset
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999, 45000000, ARRAY['VLOL1', 'MCAM1'], ARRAY[5, 25]);
```

Error producido al tratar de vender más cantidad del producto de la que tenemos en stock:

```
ERROR:  No hay suficientes existencias en la tienda del producto MCAM1
CONTEXT:  PL/pgSQL function verificar_stock() line 10 at RAISE
```

NOTA: Si se intenta vender un producto que no existe en la tienda también da error

```
Unset
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999, 45000000, ARRAY['VElDelAnillo'], ARRAY[25]);
```

```
tienda=# INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999, 45000000, ARRAY['VElDelAnillo'], ARRAY[25]);
ERROR:  No existe el producto VElDelAnillo en la tienda.
```

- Después de una venta:

```
Unset
INSERT INTO Ventas (id_empleado, id_cliente, productos, cantidades) VALUES (99999999, 45000000, ARRAY['VLOL1', 'MCAM1'], ARRAY[20, 20]);
```

Tabla tras la venta:

```
 id_tienda | id_producto | stock
-----------+-------------+-------
 TIENDA2   | PMOU1       |    18
 TIENDA1   | VLOL1       |     0
 TIENDA1   | MCAM1       |     0
(3 rows)
```

- Después de una devolución:

```
Unset
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA2', '2023-12-20', 45000000, 2, ARRAY['VLOL1', 'MCAM1'], ARRAY[3, 3]);
```

Tabla tras la devolución:

```
 id_tienda | id_producto | stock
-----------+-------------+-------
 TIENDA2   | PMOU1       |    18
 TIENDA1   | VLOL1       |     0
 TIENDA1   | MCAM1       |     0
 TIENDA2   | VLOL1       |     3
 TIENDA2   | MCAM1       |     3
(5 rows)
```

```
Unset
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA2', '2023-12-20', 45000000, 2, ARRAY['VLOL1', 'MCAM1'], ARRAY[18, 18]);
```

Error obtenido al tratar de devolver una cantidad mayor a la comprada para cada producto:

```
ERROR:  No se puede devolver más productos VLOL1 de los que se compraron.
CONTEXT:  PL/pgSQL function verificar_devolucion() line 36 at RAISE
```

```
Unset
INSERT INTO Devoluciones (id_tienda, fecha_venta, id_cliente, id_venta, productos, cantidades)
VALUES ('TIENDA2', '2023-12-20', 45000000, 1, ARRAY['VLOL2', 'MCAM1'], ARRAY[3, 3]);
```

Error obtenido al tratar de devolver un producto que no fue comprado:

```
ERROR:  El producto VLOL2 no se compró.
CONTEXT:  PL/pgSQL function verificar_devolucion() line 16 at RAISE
```

7.4 Calcular fecha del final de contrato

Al realizar las inserciones únicamente se pasa la fecha de inicio y la duración del contrato, de manera automática se calcula la fecha final.

```
Unset
INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV1', '2020-01-01', 365*5);

INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV2', '2020-01-01', 365*5);

INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV3', '2020-01-01', 365);
```

Tabla Contrato resultante tras las inserciones:

```
id_provedor | fecha_inicio | fecha_fin  | duracion
------------+--------------+------------+----------
PROV1       | 2020-01-01   | 2024-12-30 |     1825
PROV2       | 2020-01-01   | 2024-12-30 |     1825
PROV3       | 2020-01-01   | 2020-12-31 |      365
(3 rows)
```

7.5 Verificar contrato

Para comprobar que funciona correctamente el trigger he probado a realizar un envío:

```
Unset
INSERT INTO Envios (id_proveedor, id_tienda, productos, fecha, cantidades) VALUES ('PROV3',
'TIENDA2', ARRAY['PMOU1'], '2020-01-01', ARRAY[20]);
```

Como podemos observar no ha sido posible ya que no estamos dentro de la fecha. Se puede ver en el apartado anterior que el contrato del PROV3 finalizó el 31-12-2020.

```
psql:inserts.sql:24: ERROR:  El contrato con el proveedor ha finalizado.
CONTEXT:  PL/pgSQL function verificar_contrato() line 4 at RAISE
```

7.6 Verificar empleado

```
Unset
INSERT INTO Trabaja (id_tienda, id_empleado, cargo, fecha_inicio, duracion) VALUES ('TIENDA2',
99999998, 'Cajero', '2020-05-01', 365);
```

Error obtenido al tratar de hacer que un trabajador esté en dos tiendas al mismo tiempo:

```
ERROR:  El empleado ya está trabajando en otra tienda.
CONTEXT:  PL/pgSQL function verificar_empleado() line 4 at RAISE
```

7.7 Borrado de información

- Tiendas

Al eliminar una tienda se observa como toda la información asociada desaparece:

```
Unset
DELETE FROM TIENDA WHERE id_tienda = 'TIENDA2';
```

```
tienda=# SELECT * FROM TIENDAS;
 id_tienda |  nombre  | direccion |  telefonos
-----------+----------+-----------+-------------
 TIENDA1   | Tienda 1 | Calle 1   | {111111111}
(1 row)

tienda=# SELECT * FROM DISPONIBILIDAD;
 id_tienda | id_producto | stock
-----------+-------------+-------
 TIENDA1   | VLOL1       |     0
 TIENDA1   | MCAM1       |     0
(2 rows)

tienda=# SELECT * FROM ENVIOS;
 id_proveedor | id_tienda |   productos   |   fecha    | cantidades
--------------+-----------+---------------+------------+------------
 PROV1        | TIENDA1   | {VLOL1,MCAM1} | 2020-01-01 | {10,10}
 PROV1        | TIENDA1   | {VLOL1,MCAM1} | 2021-01-01 | {10,10}
(2 rows)

tienda=# SELECT * FROM TRABAJA;
 id_tienda | id_empleado | cargo  | fecha_inicio | fecha_final | duracion
-----------+-------------+--------+--------------+-------------+----------
 TIENDA1   |    99999999 | Cajero | 2020-01-01   | 2020-12-31  |      365
(1 row)

tienda=# SELECT * FROM DEVOLUCIONES;
 id_tienda | fecha_venta | id_cliente | fecha_devolucion | id_venta | id_devolucion |   productos   | cantidades
-----------+-------------+------------+------------------+----------+---------------+---------------+------------
 TIENDA1   | 2023-12-20  |   45000000 | 2023-12-20       |        1 |             1 | {VLOL1,MCAM1} | {1,1}
(1 row)
```

- Proveedores

Al eliminar un proveedor se observa como toda la información asociada desaparece:

```
Unset
DELETE FROM PROVEEDORES WHERE id_proveedor = 'PROV1';
```

```
tienda=# SELECT * FROM PROVEEDORES;
 id_proveedor |   nombre    | direccion |  telefonos
--------------+-------------+-----------+-------------
 PROV2        | Proveedor 2 | Calle 2   | {222222222}
 PROV3        | Proveedor 3 | Calle 3   | {333333333}
(2 rows)

tienda=# SELECT * FROM CONTRATO;
 id_provedor | fecha_inicio | fecha_fin  | duracion
-------------+--------------+------------+----------
 PROV2       | 2020-01-01   | 2024-12-30 |     1825
 PROV3       | 2020-01-01   | 2020-12-31 |      365
(2 rows)

tienda=# SELECT * FROM ENVIOS;
 id_proveedor | id_tienda | productos | fecha | cantidades
--------------+-----------+-----------+-------+------------
(0 rows)
```

7.8 Checks

- Salario

Unset

```
-- salario NUMERIC(10,2) CHECK (salario >= 1000) NOT NULL

INSERT INTO Empleados (id_empleado, nombre, apellidos, salario) VALUES (99999997, 'Empleado 3', 'Apellido 1', 999);
```

```
ERROR:  new row for relation "empleados" violates check constraint "empleados_salario_check"
DETAIL:  Failing row contains (99999997, Empleado 3, Apellido 1, 999.00).
```

- Fecha

Unset

```
--  fecha_inicio DATE CHECK (fecha_inicio >= DATE '2000-01-01') NOT NULL

INSERT INTO Proveedores (id_proveedor, nombre, direccion, telefonos) VALUES ('PROV4', 'Proveedor 4', 'Calle 4', ARRAY['444444444']);

INSERT INTO Contrato (id_provedor, fecha_inicio, duracion) VALUES ('PROV4', '1999-12-31', 365*5);
```

```
ERROR:  new row for relation "contrato" violates check constraint "contrato_fecha_inicio_check"
DETAIL:  Failing row contains (PROV4, 1999-12-31, 2004-12-29, 1825).
```

8. API REST con flask

Para el diseño de nuestra API REST haciendo uso del framework de Flask, hemos creado diferentes rutas para el acceso, creación, modificación o eliminación de los datos de las tablas de nuestro esquema de base de datos en postgresql. Para ello hemos diseñado rutas para las tablas de Productos, Videojuegos, Merchandising, Periféricos, Proveedores, Tiendas, Envíos, Clientes, Disponibilidad, Empleados, Trabajan, Venta y Devolución. En cada una de estas rutas hemos diseñado diferentes métodos de petición en las rutas como puede ser GET POST PUT o DELETE, para un uso completo de la API REST.

Productos

```
Unset
@app.route('/productos', methods=['GET','POST'])
@app.route('/productos/<id_producto>', methods=['GET','PUT','DELETE'])
```

Permite obtener todos los productos (GET), crear un nuevo producto (POST), obtener un producto específico (GET), actualizar un producto específico (PUT) y eliminar un producto específico (DELETE).

La información del producto devuelta incluye la información de su tipo (videojuego, merchandising, periféricos).

Videojuegos

```
Unset
@app.route('/videojuegos', methods=['GET'])
```

Permite obtener todos los videojuegos (GET)

Merchandising

```
Unset
@app.route('/merchandising', methods=['GET'])
```

Permite obtener todos los productos de merchandising (GET).

Periféricos

```
Unset
@app.route('/perifericos', methods=['GET'])
```

Permite obtener todos los periféricos (GET).

Proveedores

```
Unset
@app.route('/proveedores', methods=['GET'])
@app.route('/proveedores/<id_proveedor>', methods=['GET', 'DELETE'])
```

Permite obtener todos los proveedores (GET) y obtener o eliminar un proveedor en específico (GET, DELETE).

Tiendas

```
Unset
@app.route('/tiendas', methods=['GET'])
@app.route('/tiendas/<id_tienda>', methods=['GET', 'DELETE'])
```

Permite obtener todas las tiendas (GET), obtener una tienda específica (GET) y eliminar una tienda específica (DELETE).

Envíos

```
Unset
@app.route('/envios', methods=['GET'])
@app.route('/envios/proveedor/<id_proveedor>', methods=['GET'])
@app.route('/envios/tienda/<id_tienda>', methods=['GET'])
```

Permite obtener todos los envíos (GET), obtener los envíos de un proveedor específico (GET) y obtener los envíos a una tienda específica (GET).

Clientes

```
Unset
@app.route('/clientes', methods=['GET'])
@app.route('/clientes/<id_cliente>', methods=['GET', 'PUT', 'DELETE'])
```

Permite obtener todos los clientes (GET), obtener un cliente específico (GET), actualizar un cliente específico (PUT) y eliminar un cliente específico (DELETE).

Disponibilidad

```
Unset
@app.route('/disponibilidad', methods=['GET'])
@app.route('/disponibilidad/tienda/<id_tienda>/', methods=['GET'])
@app.route('/disponibilidad/tienda/<id_tienda>/producto/<id_producto>/', methods=['GET'])
```

Permite obtener la disponibilidad de todos los productos (GET), obtener la disponibilidad de todos los productos en una tienda específica (GET), obtener la disponibilidad de un producto específico en una tienda específica (GET).

Empleados

```
Unset
@app.route('/empleados', methods=['GET'])
@app.route('/empleados/<id_empleado>', methods=['GET', 'DELETE'])
```

Permite obtener todos los empleados (GET), obtener un empleado específico (GET) y eliminar un empleado específico (DELETE).

Trabaja

```
Unset
@app.route('/trabaja', methods=['GET'])
@app.route('/trabaja/tienda/<id_tienda>', methods=['GET'])
@app.route('/trabaja/empleado/<id_empleado>', methods=['GET', 'DELETE'])
```

Permite obtener todos los empleados que trabajan en todas las tiendas (GET), obtener los empleados que trabajan en una tienda específica (GET), obtener o eliminar la tienda donde trabaja un empleado específico (GET, DELETE).

Ventas

```
Unset
@app.route('/ventas/', methods=['GET'])
@app.route('/ventas/<id_venta>', methods=['GET','POST'])
@app.route('/ventas/tienda/<id_tienda>', methods=['GET'])
@app.route('/ventas/cliente/<id_cliente>', methods=['GET'])
@app.route('/ventas/empleado/<id_empleado>', methods=['GET'])
```

Permite obtener todas las ventas (GET), obtener o crear una venta específica (GET, POST), obtener las ventas realizadas en una tienda específica (GET), obtener las ventas realizadas a un cliente específico (GET) y obtener las ventas realizadas por un empleado específico (GET).

Devoluciones

```
Unset
@app.route('/devoluciones', methods=['GET','POST'])
@app.route('/devoluciones/<id_tienda>', methods=['GET'])
```

Permite obtener todas las devoluciones y crear una nueva (GET y POST) y obtenerlas de forma específica por tienda

9. Conclusión

A lo largo de la elaboración de este proyecto final de asignatura hemos aprendido a como realizar a partir de una idea de modelo de negocio, un modelo entidad-relación

y un grafo relacional. También hemos visto cómo transformar este grafo relacional a una implementación real de base de datos en postgresql, teniendo que realizar las comprobaciones de valores o de tipos necesarias para cada atributo así como el comportamiento de ellas, y la generación de triggers necesarios a la hora de la inserción de valores en nuestra base de datos. Finalmente hemos aprendido a cómo realizar con Flask un API rest por la cual poder interactuar con nuestra base de datos a través de peticiones que nos permitan ejecutar las instrucciones básicas CRUD.