



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO  
Facultad de Ingeniería

---

**BASES DE DATOS  
TEMA**

---

**PROYECTO FINAL**

**López Sugahara Ernesto Danjiro  
Ramirez Martinez Luis Angel  
Rodríguez Kobeh Santiago  
Soto Rivera Marco Antonio  
Villaseñor Venegas Carlos Miguel**

*Professor:*  
**Ing. Fernando Arreola Franco**  
*Grupo 01 | semestre 2024-1*

26 de noviembre de 2023



# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>2</b>
<b>3. Propuesta de solución</b>	<b>3</b>
<b>4. Plan de trabajo</b>	<b>3</b>
<b>5. Diseño</b>	<b>6</b>
<b>6. Implementación</b>	<b>15</b>
<b>7. Presentación</b>	<b>33</b>
<b>8. Conclusiones</b>	<b>43</b>

# PROYECTO DE APLICACIÓN. BASES DE DATOS

LÓPEZ DANJIRO  
RAMÍREZ ÁNGEL  
RODRÍGUEZ SANTIAGO  
SOTO MARCO  
VILLASEÑOR CARLOS

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

## 1. Introducción

### *Análisis del problema*

Se plantea la creación de una base de datos para la organización y almacenamiento de la información de un restaurante. Para esto es necesario realizar un modelo que permita cumplir con los requerimientos establecidos por el cliente, es decir, los requerimientos acordados en la gestión de negocio. Estos requerimientos permiten modelar la base de datos, obteniendo de este modo las funcionalidades deseadas y la relación entre la información correcta. Igualmente, se planteó la creación de una página WEB que permitiese obtener las funcionalidades especificadas, y, además, permita visualizar la información de los empleados (incluyendo su imagen)

## 2. Objetivos

- Construir el modelo adecuado de datos para la posterior implementación física de la base de datos.
- Construir la base de datos física, incluyendo la funcionalidad especificada por el cliente.
- Construir la página WEB que permita la visualización de los elementos solicitados.
- Tener una integridad de información y un modelo relacional que permita tener un modelo rígido y estructurado de la información.
- Analizar vulnerabilidades y posibles mejoras en la implementación.



- Aplicar conceptos de modelos relacionales, así como construcción de diseño con diagramas entidad – relación.
- Determinar si es necesario o no normalizar o desnormalizar el modelo obtenido.

### 3. Propuesta de solución

Se plantea realizar una página WEB que permita obtener la información a través del uso del lenguaje de programación Python y la librería Flask, lo que permite la interoperabilidad entre Python, HTML y CSS. Con esto, se podrá realizar la programación de manera bastante sencilla sin la necesidad de utilizar JavaScript. Para los datos se seguirán los siguientes pasos:

1. Análisis de requerimientos: Esto con el fin de comprender en su totalidad lo que el cliente está solicitando y poder construir un modelo de datos que, al interactuar con la página WEB, cumpla todas las peticiones realizadas por el cliente.
2. Realizar un modelo conceptual para representar de forma gráfica la interpretación de los requerimientos, buscando que se cumplan todos los objetivos desde este punto.
3. Se realizará el modelo lógico, analizando la integridad de los datos y la posibilidad de normalización de las tablas que resulten del modelo planteado.
4. Finalmente se realizará el modelo físico que se entrelazará con la página WEB. El modelo físico se solicita realizarse en PostgreSQL y utilizando el lenguaje pgSQL para la construcción de bloques de programación que permitan obtener el comportamiento deseado.

### 4. Plan de trabajo

El plan de trabajo se puede visualizar con la tabla siguiente:

Entregable	Descripción y paquetes de trabajo	Responsables



Análisis de requerimientos	<p>Se realizó un análisis de los elementos que componen el plan de negocio realizado con el cliente. En este punto, se buscó identificar las entidades que componen a la base, sus relaciones y sus posibles dominios de información. Igualmente, se empezó a pensar en cómo resolver la relación entre la base de datos y la implementación de una página WEB.</p>	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
Modelo conceptual de la base de datos	<p>Planteamiento y visualización del modelo entidad – relación.</p>	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
	<p>Construcción del modelo gráfico en una herramienta de diseño gráfico.</p>	<ul style="list-style-type: none"> <li>■ Villaseñor Venegas Carlos Miguel</li> </ul>
Modelo lógico de la base de datos	<p>Análisis y diseño del modelo relacional con base en lo obtenido en el modelo ER.</p>	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
	<p>Construcción del modelo relacional mediante software de diseño.</p>	<ul style="list-style-type: none"> <li>■ Ramírez Martínez Luis Ángel</li> </ul>



Modelo físico de la base de datos	A través del análisis realizado para el modelo lógico, se procedió a realizar la implementación física de la base y las respectivas funciones dentro de PostgreSQL.	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
Implementación de la página WEB	Código en Python para enlazar la base y probar las funciones implementadas.	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
	HTML para las respectivas ventanas de las funcionalidades solicitadas.	<ul style="list-style-type: none"> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
	CSS para el diseño de cada una de las vistas que se le presentan al usuario.	<ul style="list-style-type: none"> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
	Pruebas de funcionamiento.	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Soto Rivera Marco Antonio</li> </ul>



Documentación	Se realizó la documentación de todos los elementos que componen la solución del problema planteado.	<ul style="list-style-type: none"> <li>■ López Sugahara Ernesto Danjiro</li> <li>■ Ramírez Martínez Luis Ángel</li> <li>■ Villaseñor Venegas Carlos Miguel</li> <li>■ Rodríguez Kobeh Santiago</li> <li>■ Soto Rivera Marco Antonio</li> </ul>
---------------	---	--

**Nota:** La tabla está escrita de forma secuencial, es decir, la primera entrada de la tabla es la primera actividad que se realizó y así sucesivamente hasta llegar a la documentación (aunque la documentación se encuentra al final de los paquetes de trabajo, realmente se fue desarrollando de forma paralela a los paquetes de trabajo superiores). Igualmente, cada uno de los entregables y su paquete de trabajo se desarrolla de manera más completa en el siguiente apartado del documento.

## 5. Diseño

### Análisis de los requerimientos

Para la parte del diseño fue fundamental realizar un análisis de los requerimientos solicitados para poder implementar un modelo gráfico sólido que facilitara el desarrollo de las actividades subsecuentes de diseño. Para esto, se tiene el planteamiento dado por el cliente: “Se debe almacenar el RFC, el número de empleado, nombre, fecha de nacimiento, teléfonos, edad, domicilio, sueldo; de los cocineros, su especialidad, de los meseros su horario y de los administrativos su rol, así como una foto de los empleados y considerar que un empleado puede tener varios puestos. Es necesario tener registro de los dependientes de los empleados, su CURP, nombre y parentesco. Se debe tener disponible la información de los platillos y bebidas que el restaurante ofrece, una descripción, nombre, su receta, precio y un indicador de disponibilidad, así como el nombre y descripción de la categoría a la que pertenecen (considerar que un platillo o bebida solo pertenece a una categoría). Debe tener registro del folio de la orden, fecha (con hora), la cantidad total a pagar por la orden y registro del mesero que levantó la orden, así como la cantidad de cada platillo/bebida y precio total a pagar por platillo/bebida contenidos en cada orden. Considerar que es posible que los clientes soliciten factura de su consumo, por lo que debe almacenarse su RFC, nombre, domicilio, razón social, email y fecha de nacimiento.”

### Modelo Entidad-Relación

Con base en los requerimientos planteados, se plantearon las entidades siguientes:

- Empleado:

Esta se va a plantear como una supertipo, donde los subtipos son las respectivas categorías de empleados. El empleado cuenta con los atributos siguiente:



- Número de empleado: Esta será la llave principal que va a identificar a los empleados.
  - RFC: Esta será una llave candidata y deberá de ser único.
  - Nombre: Es un atributo compuesto por el nombre de pila y los apellidos.
  - Sueldo: Se deberá de validar que el sueldo esté por encima del salario mínimo.
  - Domicilio: Es un atributo compuesto por los siguientes elementos: calle, número, colonia, estado y CP.
  - Fecha de nacimiento.
  - Edad.
  - Teléfono: Será un atributo multivalorado, es decir, varios empleados pueden tener varios números registrados. Los empleados pueden pertenecer a diferentes categorías caracterizados por:
    - Cocinero: Para los cocineros habrá que guardar su especialidad.
    - Meseros: Para ellos se deberá de guardar el horario de trabajo.
    - Administrativo: Para estos se deberá de asignar el rol que desempeñan en la empresa.
- Dependiente:
- Los dependientes son personas que dependen económicamente del empleado. En este caso se considerará que es una entidad débil, ya que, si el empleado ya no pertenece a la empresa, no es necesario tener la información de los dependientes. Estos tendrán los siguientes atributos:
- CURP: Se observó que, si hay un atributo que lo identifica de forma única, sin embargo, esto solo quita la necesidad de una llave compuesta con el atributo discriminante.
  - Nombre: Al igual que en los empleados, este será un atributo compuesto por el nombre de pila y los respectivos apellidos.
  - Parentesco.
- Producto: Para los productos se tendrán los siguientes atributos:
- El id del alimento será el atributo que identifique de forma única al producto.
  - Nombre.
  - Descripción: Aquí se deberá de especificar al producto, sus características y elementos de interés para los posibles clientes.



- Receta: Puede ser un texto plano con los pasos a seguir, o en formato de lista.
  - Disponibilidad: Únicamente servirá para señalizar si un producto se encuentra actualmente disponible o no.
  - Precio.
- Categoría:

Al restaurante le interesa tener dos categorías principales: alimentos y bebidas. Se decidió tomar este punto como una entidad aparte para evitar la repetición de información al momento de crear la base de datos. Igualmente, en caso de querer agregar otra categoría adicional, será más fácil si se trata de una entidad separada. Se compone de los atributos siguientes:

- Un identificador de la categoría.
  - El nombre de la categoría.
  - Una descripción de la categoría, donde se especifique el tipo de productos que se van a vender y clasificar.
- Orden: Para la orden, se tiene el siguiente listado de atributos:
- Folio: Este se deberá de componer de un secuenciador que vaya aumentando conforme se van registrando órdenes. Deberá de tener el formato: ORD-.
  - Fecha: Será la fecha y la hora a partir de la cual se registró la orden.
  - Total: Será el total a cobrar por todos los productos registrados dentro de la orden.
- Cliente: Al restaurante le interesa la posibilidad de generar facturas cuando el cliente lo desee, en dichos casos es necesario tener la información asociado a este. A diferencia de los dependientes, al restaurante sí le interesa almacenar la información asociada a dicho cliente, ya que puede ser que posteriormente vuelva a atender el establecimiento y necesite generar nuevamente una factura. El cliente tendrá los siguientes atributos:
- RFC: Será el identificador único de los clientes.
  - Nombre: Será un atributo compuesto por el nombre y los apellidos.
  - Fecha de nacimiento.
  - Email.
  - Razón social.

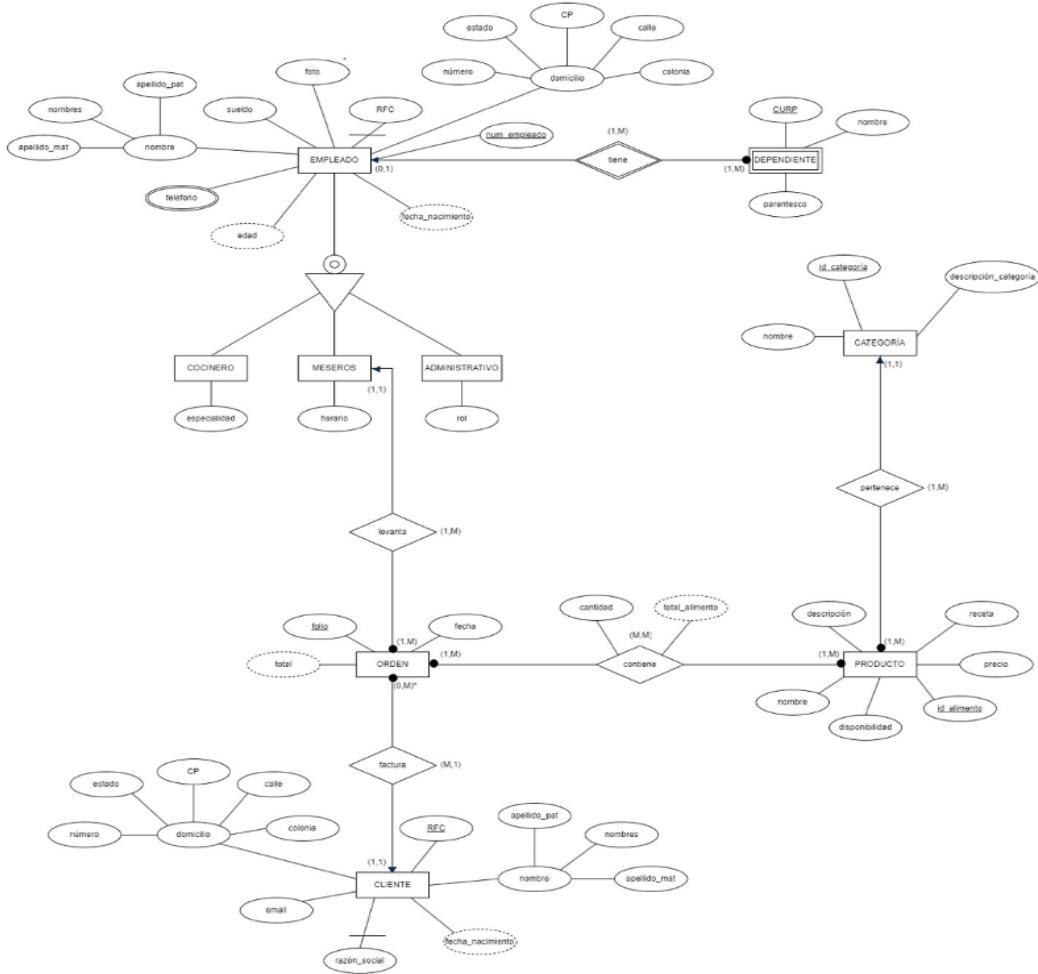


- Domicilio: Es un atributo compuesto por los siguientes elementos: calle, número, colonia, estado y CP.

Estas fueron las entidades identificadas en el análisis de requerimientos, las cuales componen la estructura base de la base de datos. Posteriormente se realizó en análisis de las relaciones entre dichos atributos, observando las siguientes:

- Hay una relación entre empleado y dependiente. En este caso se considera que un empleado puede tener varios dependientes, pero no que un dependiente puede tener varios trabajadores asociados; es decir, si una persona es hijo de dos trabajadores, únicamente podrá ser dependiente de uno de ellos. Con esto se tiene una relación de un empleado, muchos dependientes.
- Los empleados que son meseros estarán relacionados con las órdenes. Serán estos los que levanten o registren una orden. En este caso, es evidente que un empleado puede levantar varias órdenes, sin embargo, no es posible que una orden sea registrada por varios empleados. Con lo anterior, se tiene una relación muchos a uno.
- La orden se relaciona con los productos, ya que la orden se compone esencialmente de los productos registrados. En este caso, una orden puede tener varios productos, así como varios productos pueden estar asociados a diferentes órdenes. Como se menciona en los requerimientos, se desea saber la cantidad total de cada producto en la orden, así como el total a pagar por ese producto en particular. Con esto se observa que habrá una relación entre producto y órdenes de muchos a muchos. Esto se asemejará a un ticket de la orden generada.
- Las órdenes se relacionan con el cliente a través de la facturación. En este caso se tiene una participación opcional, ya que no todos los clientes quieren factura, por lo que no todas las órdenes tendrán un cliente asociado. Con esto se puede visualizar que, una o ninguna orden tiene asociado un cliente, y un cliente puede estar asociado con muchas órdenes (pensando en un cliente recurrente). En este caso la cardinalidad del cliente no es opcional, ya que, su registro en la base significa que sí o sí realizó alguna factura, por lo que finalmente se tiene la relación: un cliente tiene muchas facturas.

Con esto, ya se tiene todo el análisis para realizar el diseño del modelo entidad – relación, por lo que el modelo resultó de la manera siguiente:



### Modelo Relacional

Una vez construido el modelo entidad – relación, se comenzó a analizar la transformación al modelo lógico o modelo relacional. En este se realizó el mapeo de cada una de las entidades, así como las respectivas relaciones. El procedimiento se muestra a continuación: *Entidad: empleado y sus subtipos*

En este caso se consideró el mapeo a través de banderas, es decir, se tendrá una única relación denominada empleado que contendrá banderas para determinar si el empleado entra dentro de alguna o algunas de las categorías. Esta aproximación se escogió debido a que no son tantos los subtipos que conforman a la entidad, así como los atributos de cada uno son pocos. Este mapeo es de gran utilidad para la participación total y opcional, y, lo más importante, es de gran utilidad cuando se tiene traslape; que, como se menciona en los requerimientos del cliente, los empleados pueden pertenecer a varias categorías. Con lo anterior, se puede visualizar la relación de la manera siguiente:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
EMPLEADO	Número de empleado	Integer	PK
	Nombre	Varchar(60)	NN
	Apellido paterno	Varchar(60)	NN



Apellido materno	Varchar(60)	NN
RFC	Char(7)	UNIQUE
Fecha de nacimiento	Date	NN
Edad	Smallint	NN
Foto	Bytea	NN
Calle	Varchar(60)	NN
Número de casa	Smallint	NN
Colonia	Varchar(30)	NN
Código postal	Char(5)	NN
Estado	Varchar(30)	NN
Es mesero	Boolean	NULL
Horario	Varchar(15)	NULL
Es administrativo	Boolean	NULL
Rol	Varchar(50)	NULL
Es cocinero	Boolean	NULL
Especialidad	Varchar(50)	NULL

**Nota:** La foto se solicitó que se almacenara en la base de datos como información en bytes, por lo mismo, se seleccionó el tipo de dato bytea.

Del modelo entidad – relación se observa que el empleado cuenta con un atributo multivalorado, por lo mismo, se deberá de realizar una relación adicional la cual representará el número de teléfono. Esta se verá de la manera siguiente:

*Entidad: Teléfono*

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
TELÉFONO	Teléfono	Bigint	PK
	Número de empleado	Integer	PK,FK

Con lo anterior, ya es posible tener toda la información relacionada al empleado registrada de manera adecuada.

*Entidad: Dependiente*

Para los dependientes se propuso la siguiente relación:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
DEPENDIENTE	CURP	Char(18)	PK
	Nombre	Varchar(60)	NN
	Apellido paterno	Varchar(60)	NN
	Apellido materno	Varchar(60)	NN



	Parentesco	Varchar(40)	NN
	Número de empleado del que depende	Integer	FK

**Nota:** Como se mencionó, en este caso la entidad débil no tiene una dependencia de identificación, por lo que la llave primaria de la entidad fuerte de la que depende, únicamente se propaga como llave foránea.

#### *Entidad: Orden*

Las órdenes, además de sus atributos, se observó del análisis del modelo entidad – relación que se relaciona con los meseros y los clientes en una relación uno a muchos, siendo que la llave de las dos entidades se propaga a esta entidad como llaves foráneas. Con lo anterior en cuenta, es posible visualizar la siguiente propuesta de diseño:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
ORDEN	Folio	Char(7)	PK
	Fecha	Timestamp	NN
	Total	Double precision	NN,C
	ID del mesero	Integer	FK
	RFC el cliente	Char(13)	FK

En este caso cabe recalcar el folio, el cual se solicitó que tuviera un formato que consta de 7 caracteres específicamente.

#### *Entidad: Cliente*

Los clientes no sufrirán modificación alguna debido a las relaciones existentes en el modelo, ya que, como se observa en la entidad orden, la llave del cliente se propaga a esta entidad a través de su única relación. Con esto, se planteó la siguiente propuesta de diseño:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
CLIENTE	RFC	Char(13)	PK
	Nombre	Varchar(60)	NN
	Apellido paterno	Varchar(60)	NN
	Apellido materno	Varchar(60)	NN
	Fecha de nacimiento	Date	NN
	Email	Varchar(120)	NN
	Razón social	Text	U,NN
	Calle	Varchar(60)	NN
	Número	Smallint	NN
	Colonia	Varchar(30)	NN
	Código postal	Char(5)	NN



	Estado	Varchar(30)	NN
--	--------	-------------	----

### *Entidad: Producto*

En este caso, el producto tiene una propagación de llave foráneo debido a la categoría, por lo mismo, se tiene la siguiente propuesta de diseño:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
PRODUCTO	Identificador del producto	Integer	PK
	Nombre del producto	Varchar(60)	NN
	Descripción	Text	NN
	Precio	Double precision	NN
	Disponibilidad	Boolean	NN
	Receta	Text	NN
	Identificador de categoría	Integer	FK
	Ventas	Integer	NN,C

**Nota:** En este caso se agregó un atributo adicional a los especificados en los requerimientos del cliente. El atributo ventas se encontró útil para el planteamiento de las etapas de implementación que se presentan más adelante en el documento. Esto se debe a que se solicitó una función para mostrar los productos más vendidos. Esto sería posible resolver mediante consultas un poco largas, por lo mismo se pensó en la optimización mediante la creación de este simple atributo calculado que permite llevar el registro de las ventas y, por ende, un campo de recuperación de información que se puede obtener mediante una consulta sencilla.

### *Entidad: Categoría*

En este caso, la solución de la relación de la categoría con el producto ya se resolvió en la entidad de producto, por lo que la propuesta para la entidad es la siguiente:

Nombre de la relación	Atributos	Dominio del atributo	Restricciones
CATEGORIA	Identificador de la categoría	Integer	PK
	Nombre	Varchar(20)	NN
	Descripción	Text	NN
	Precio	Double precision	NN

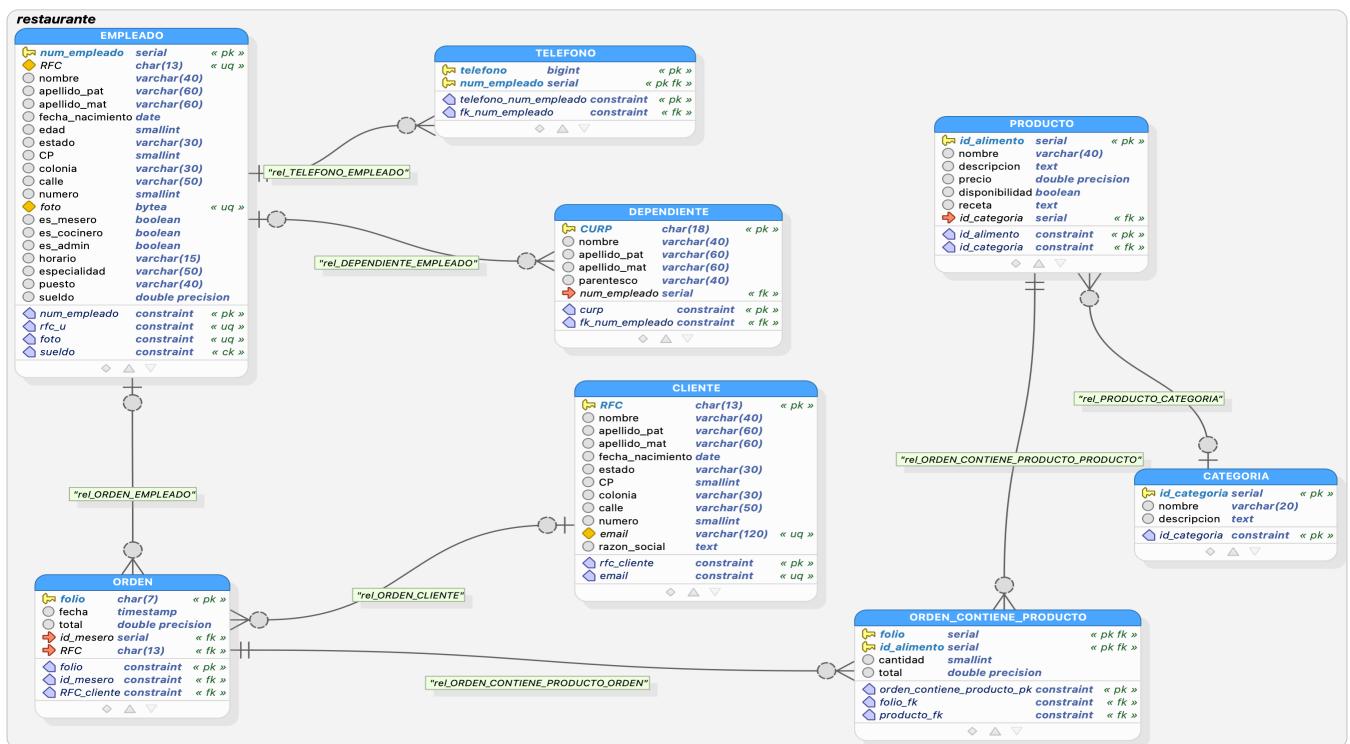
### *Relación: Orden y producto*

Del modelo entidad – relación se observó que se tiene una relación de muchos a muchos entre las órdenes y los productos, lo que asimila a un ticket. Para la resolución de esta relación, se tiene el siguiente mapeo:



Nombre de la relación	Atributos	Dominio del atributo	Restricciones
Orden contiene producto	Folio de la orden	Char(7)	FK,PK
	Identificador del alimento	Integer	FK,PK
	Cantidad	Smallint	NN
	Total del producto	Double precision	NN,C

Con lo anterior, se realizó el diseño en el software de diseño PgModeler, obteniendo lo siguiente:





## 6. Implementación

Con los análisis y diseños obtenidos en los procesos anteriores, fue posible realizar la implementación del modelo físico mediante DDL en el software PostgreSQL. Para esto, a continuación, se muestra el código con para la creación de la base de datos.

### *Creación de secuenciadores*

En este caso se tienen los siguientes secuenciadores que son de utilidad para las llaves primarias de ciertas relaciones. En este caso se tienen las siguiente:

#### Secuenciador de categoría

```
CREATE SEQUENCE IF NOT EXISTS restaurante.categoría_id_categoria_seq
    INCREMENT 1
    START 1
    MINVALUE 1
    MAXVALUE 2147483647
```

#### Secuenciador de folio

Este es de gran utilidad para el requerimiento específico solicitado, donde se solicita que los folios tengan el formato ORD-.

```
CREATE SEQUENCE IF NOT EXISTS restaurante.folio_seq
    CYCLE
    INCREMENT 1
    START 1
    MINVALUE 1
    MAXVALUE 999
```

#### Secuenciador de identificador de empleado:

Aunque no es del todo recomendable utilizar secuenciadores para llaves primarias, en este caso se pensó ya que usualmente se tiene un número de empleado con el que se identifica a los diferentes elementos de la empresa. Igualmente, es importante identificar que se tiene un atributo RFC que es único, por lo que es complicado tener algún problema con la llave primaria serial.

```
CREATE SEQUENCE IF NOT EXISTS restaurante.id_empleado_seq
    INCREMENT 1
    START 1
    MINVALUE 1
    MAXVALUE 2147483647
    CACHE 1
```



Secuenciador de identificador de producto:

En este caso se planteó lo mismo para los productos, de modo que su identificador sea fácil de identificar en el stock. Sin embargo, en este caso no es del todo óptimo del uso de dicho secuenciador, sin embargo, dada la forma en la que se realiza la conexión a la base, errores o conflictos con esta selección se limitan bastante.

```
CREATE SEQUENCE IF NOT EXISTS restaurante.producto_id_alimento_seq
INCREMENT 1
START 1
MINVALUE 1
MAXVALUE 2147483647
CACHE 1
```

*Creación de tablas*

Tabla de empleados



```
CREATE TABLE IF NOT EXISTS restaurante.empleado
(
    num_empleado integer NOT NULL DEFAULT nextval('restaurante.id_empleado_seq'::regclass),
    rfc character(13) COLLATE pg_catalog."default" NOT NULL,
    nombre character varying(60) COLLATE pg_catalog."default" NOT NULL,
    apellido_pat character varying(60) COLLATE pg_catalog."default" NOT NULL,
    apellido_mat character varying(60) COLLATE pg_catalog."default" NOT NULL,
    fecha_nacimiento date NOT NULL,
    edad smallint NOT NULL,
    estado character varying(30) COLLATE pg_catalog."default" NOT NULL,
    cp character(5) COLLATE pg_catalog."default" NOT NULL,
    colonia character varying(30) COLLATE pg_catalog."default" NOT NULL,
    calle character varying(60) COLLATE pg_catalog."default" NOT NULL,
    numero smallint NOT NULL,
    foto bytea NOT NULL,
    es_mesero boolean,
    horario character varying(15) COLLATE pg_catalog."default",
    es_admin boolean,
    rol character varying(50) COLLATE pg_catalog."default",
    es_cocinero boolean,
    especialidad character varying(50) COLLATE pg_catalog."default",
    sueldo double precision,
    CONSTRAINT empleado_pk PRIMARY KEY (num_empleado),
    CONSTRAINT empleado_rfc_key UNIQUE (rfc),
    CONSTRAINT empleado_sueldo_check CHECK (sueldo > 3000.0::double precision)
)
```



### Tabla de dependientes

```
CREATE TABLE IF NOT EXISTS restaurante.dependiente
(
    curp character(18) COLLATE pg_catalog."default" NOT NULL,
    nombre character varying(60) COLLATE pg_catalog."default" NOT NULL,
    apellido_pat character varying(60) COLLATE pg_catalog."default" NOT NULL,
    apellido_mat character varying(60) COLLATE pg_catalog."default" NOT NULL,
    parentesco character varying(40) COLLATE pg_catalog."default" NOT NULL,
    num_empleado integer NOT NULL,
    CONSTRAINT dependiente_pk PRIMARY KEY (curp, num_empleado),
    CONSTRAINT num_empleado_fk FOREIGN KEY (num_empleado)
        REFERENCES restaurante.empleado (num_empleado) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
```

### Tabla de teléfonos

```
CREATE TABLE IF NOT EXISTS restaurante.telefono
(
    telefono bigint NOT NULL,
    num_empleado integer NOT NULL,
    CONSTRAINT telefono_pk PRIMARY KEY (telefono, num_empleado),
    CONSTRAINT num_empleado_fk FOREIGN KEY (num_empleado)
        REFERENCES restaurante.empleado (num_empleado) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
```

### Tabla de productos



```
CREATE TABLE IF NOT EXISTS restaurante.producto
(
    id_alimento integer NOT NULL DEFAULT nextval('restaurante.producto_id_alimento_seq'::regclass),
    nombre character varying(60) COLLATE pg_catalog."default" NOT NULL,
    descripcion text COLLATE pg_catalog."default" NOT NULL,
    precio double precision NOT NULL,
    disponibilidad boolean NOT NULL,
    receta text COLLATE pg_catalog."default" NOT NULL,
    id_categoria integer NOT NULL,
    ventas integer DEFAULT 0,
    CONSTRAINT producto_pk PRIMARY KEY (id_alimento),
    CONSTRAINT producto_nombre_key UNIQUE (nombre),
    CONSTRAINT id_categoria_fk FOREIGN KEY (id_categoria)
        REFERENCES restaurante.categoría (id_categoria) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
```

## Tabla de categorías

```
CREATE TABLE IF NOT EXISTS restaurante.categoría
(
    id_categoria integer NOT NULL DEFAULT nextval('restaurante.categoría_id_categoria_seq'::regclass),
    nombre character varying(20) COLLATE pg_catalog."default" NOT NULL,
    descripcion text COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT categoria_pk PRIMARY KEY (id_categoria)
)
```

## Tabla de ordenes

```
CREATE TABLE IF NOT EXISTS restaurante.orden
(
    folio character(7) COLLATE pg_catalog."default" NOT NULL DEFAULT ('ORD-'::text || lpad((nextval('restaurante.folio_seq'::regclass))::text, 3, '0'::text)),
    fecha timestamp with time zone NOT NULL DEFAULT now(),
    total double precision NOT NULL DEFAULT 0.0,
    id_mesero integer NOT NULL,
    rfc_cliente character(13) COLLATE pg_catalog."default" DEFAULT NULL::bpchar,
    CONSTRAINT orden_pk PRIMARY KEY (folio),
    CONSTRAINT fk_id_mesero FOREIGN KEY (id_mesero)
        REFERENCES restaurante.empleado (num_empleado) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT rfc_cliente_fk FOREIGN KEY (rfc_cliente)
        REFERENCES restaurante.cliente (rfc) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
```

## Tabla de ordenes y productos



```

5 CREATE TABLE IF NOT EXISTS restaurante.orden_contiene_producto
6 (
7     folio character(7) COLLATE pg_catalog."default" NOT NULL,
8     id_alimento integer NOT NULL,
9     cantidad smallint NOT NULL,
10    total_producto double precision NOT NULL,
11    CONSTRAINT orden_producto_pk PRIMARY KEY (folio, id_alimento),
12    CONSTRAINT folio_fk FOREIGN KEY (folio)
13        REFERENCES restaurante.orden (folio) MATCH SIMPLE
14        ON UPDATE RESTRICT
15        ON DELETE CASCADE,
16    CONSTRAINT id_alimento_fk FOREIGN KEY (id_alimento)
17        REFERENCES restaurante.producto (id_alimento) MATCH SIMPLE
18        ON UPDATE RESTRICT
19        ON DELETE CASCADE
20 )
--
```

## Tabla de clientes

```

5 CREATE TABLE IF NOT EXISTS restaurante.cliente
6 (
7     rfc character(13) COLLATE pg_catalog."default" NOT NULL,
8     nombre character varying(60) COLLATE pg_catalog."default" NOT NULL,
9     apellido_pat character varying(60) COLLATE pg_catalog."default" NOT NULL,
10    apellido_mat character varying(60) COLLATE pg_catalog."default" NOT NULL,
11    fecha_nacimiento date NOT NULL,
12    estado character varying(30) COLLATE pg_catalog."default" NOT NULL,
13    cp character(5) COLLATE pg_catalog."default" NOT NULL,
14    colonia character varying(30) COLLATE pg_catalog."default" NOT NULL,
15    calle character varying(60) COLLATE pg_catalog."default" NOT NULL,
16    numero smallint NOT NULL,
17    email character varying(120) COLLATE pg_catalog."default" NOT NULL,
18    razon_social text COLLATE pg_catalog."default" NOT NULL,
19    CONSTRAINT cliente_pk PRIMARY KEY (rfc)
20 )
--
```

Con lo anterior se tiene la estructura de la base de datos construida con base en todos los elementos analizados en las fases de diseño. Posteriormente se registró cierta información en la base de datos.

En este caso, una vez construida la base de datos se procedió a crear procedimientos para la inserción de información (esto no se solicitaba en los requerimientos, pero se realizaron para su uso fuera de la base de datos). Con esto, se crearon los siguientes procedimientos para insertar información a la base de datos:

## Agregar empleado

El llamado a la función es el siguiente:



```

CREATE OR REPLACE PROCEDURE restaurante.agregar_empleado(
    IN rfc character,
    IN nombre character varying,
    IN apellido_pat character varying,
    IN apellido_mat character varying,
    IN fecha_nacimiento date,
    IN edad smallint,
    IN estado character varying,
    IN cp character,
    IN colonia character varying,
    IN calle character varying,
    IN numero smallint,
    IN mesero boolean,
    IN horario character,
    IN administrativo boolean,
    IN rol character varying,
    IN cocinero boolean,
    IN especialidad character varying,
    IN sueldo double precision,
    IN foto bytea,
    IN telefonos bigint[] DEFAULT NULL::bigint[])
LANGUAGE 'plpgsql'

```

En este caso la función recibe todos los elementos para agregar la información a la tabla de empleados. Se observa que en este procedimiento se incluye también el elemento de agregado de información a la tabla de teléfonos. El comportamiento es el siguiente:

```

AS $BODY$
DECLARE
    telefono bigint;
BEGIN
    INSERT INTO restaurante.empleado (rfc,nombre,apellido_pat,apellido_mat,fecha_nacimiento,edad,estado,cp,
    colonia,calle,numero,foto,es_mesero,horario,es_administrativo,rol,es_cocinero,especialidad,sueldo)
    VALUES(RFC,nombre,apellido_pat,apellido_mat,fecha_nacimiento,edad,estado,cp,colonia,
    calle,numero,foto,mesero,horario,administrativo,rol,cocinero,especialidad,sueldo);
    -- Se debe de ir agregando los respectivos números telefónicos
    IF telefonos IS NOT NULL THEN
        FOREACH telefono IN ARRAY telefonos
        LOOP
            INSERT INTO restaurante.telefono(num_empleado,telefono) VALUES (currval('restaurante.id_empleado_seq)::regclass),telefono);
        END LOOP;
    END IF;
    EXCEPTION
        WHEN others THEN
            PERFORM setval('restaurante.id_empleado_seq)::regclass, currval('restaurante.id_empleado_seq)::regclass) - 1);
            RAISE NOTICE 'Error al agregar al empleado';
    END;
$BODY$;

```

La información del empleado se agrega de forma directa. En el caso de los teléfonos, es necesario iterar a lo largo del arreglo de teléfonos que se brindó a la función, agregando un registro en la tabla por cada uno de ellos.

En caso de error, el procedimiento reiniciará el secuenciador para evitar errores posteriores. Es importante notar que se está utilizando un procedimiento, por lo que se trata de una transacción completa, con lo que, cualquier problema generará que la información no se agregue a la base de datos. Otra cuestión es importante es que no se realiza un trigger de comprobación de los datos, ya que estos se solicitan a través de formularios HTML que validan la información de forma más sencilla, quitando de este modo trabajo y complejidad en la base.

### Agregar a un dependiente



En este caso, aunque el procedimiento no cambia tanto de un llamado común, es más sencillo realizar la consulta desde el código que se conecta a la base, además de que, como se mencionó, se tiene la ventaja de que ya se trabaja como una transacción, por lo que se está seguro de que habrá una integridad en la información al usar el procedimiento:

```
CREATE OR REPLACE PROCEDURE restaurante.agregar_dependiente(
    IN curp character,
    IN nombre character varying,
    IN apellido_pat character varying,
    IN apellido_mat character varying,
    IN parentesco character varying,
    IN num_empleado integer)
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    INSERT INTO restaurante.dependiente (curp,nombre,apellido_pat,apellido_mat,parentesco,num_empleado)
    VALUES(curp,nombre,apellido_pat,apellido_mat,parentesco,num_empleado);
END;
$BODY$;
```

### Agregar producto

Al igual que el caso del empleado, la implementación de este procedimiento es de gran utilidad para mantener la integridad mediante la transacción e, igualmente, conseguir que en caso de error se modifique el secuenciador de manera automática evitando errores de referencia posteriores.

```
CREATE OR REPLACE PROCEDURE restaurante.agregar_producto(
    IN nombre character varying,
    IN descripcion text,
    IN precio double precision,
    IN disponibilidad boolean,
    IN receta text,
    IN id_categoria integer)
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    INSERT INTO restaurante.producto (nombre,descripcion,precio,disponibilidad,receta,id_categoria)
    VALUES(nombre,descripcion,precio,disponibilidad,receta,id_categoria);
EXCEPTION
    WHEN others THEN
        PERFORM setval('restaurante.producto_id_alimento_seq)::regclass, curval('restaurante.producto_id_alimento_seq)::regclass') - 1;
        RAISE NOTICE 'Error al agregar el producto';
END;
$BODY$;
```

### Agregar categoría

El comportamiento es el mismo que el procedimiento para agregar un producto. En este caso, nuevamente se reinicia el secuenciador en caso de error:



```
CREATE OR REPLACE PROCEDURE restaurante.agregar_categoria(
    IN nombre character varying,
    IN descripción text)
LANGUAGE 'plpgsql'
AS $BODY$ 
BEGIN
    INSERT INTO restaurante.categoría (nombre, descripción)
    VALUES(nombre,descripción);
EXCEPTION
    WHEN others THEN
        PERFORM setval('restaurante.categoría_id_categoria_seq'::regclass, currval('restaurante.categoría_id_categoria_seq'::regclass) - 1);
        RAISE NOTICE 'Error en agregar la categoría';
END;
$BODY$;
```

### Agregar orden

En este caso se tenía que considerar el requerimiento de que únicamente aquellos empleados que fuesen meseros podrían tomar órdenes. Para esto había dos posibilidades, el uso de triggers o el uso de estructuras de selección. La solución fue utilizar condiciones como se observa a continuación:

```
CREATE OR REPLACE PROCEDURE restaurante.agregar_orden(
    IN id_mesero integer)
LANGUAGE 'plpgsql'
AS $BODY$ 
DECLARE
    num_trabajador integer;
BEGIN
    -- Se tiene que validar que el número de trabajador corresponda a un mesero
    SELECT re.num_empleado INTO num_trabajador FROM restaurante.empleado re WHERE re.num_empleado = id_mesero AND re.es_mesero = 'true';
    IF num_trabajador IS NOT NULL THEN
        INSERT INTO restaurante.orden (id_mesero)
        VALUES(id_mesero);
    ELSE
        RAISE EXCEPTION 'Únicamente los meseros pueden tomar órdenes.';
    END IF;
END;
$BODY$;
```

En la función se observa que, mediante una consulta se valida que haya un empleado que sea mesero y tenga el número de trabajador mandado. En caso de cumplir, únicamente se inserta el campo del número de mesero a la tabla; esto se debe a que, los demás valores se comportan de la manera siguiente:

- El folio se obtiene de forma automática con el formato ORD-. Esto se observa del código de creación de tabla:

```
folio character(?) COLLATE pg_catalog."default" NOT NULL DEFAULT ("ORD-'::text || lpad((nextval('restaurante.folio_seq'::regclass))::text, 3, '0'::text)),
```

En este caso no es necesario reiniciar el secuenciador, ya que la inserción y el llamado al secuenciador únicamente se realizará cuando se cumpla la condición, por lo que la inserción será siempre con un valor esperado válido.

- La fecha, al igual que el folio, tiene un valor por default que asigna la fecha y hora del instante en el cual



se realiza la operación.

- El RFC del cliente es un valor nulo al principio, ya que no se sabe si el cliente desea facturar o no.
- Finalmente, el total se va calculando conforme se van agregando productos a la orden. Con lo que inicialmente se le asigna un valor de 0.

### Agregar cliente

En este caso se pensó en el factor de que, el agregar al cliente involucra que se está realizando un proceso de facturación. Generalmente se solicitará la información al cliente con lo cual se deberá de generar la respectiva factura. Con esto en mente, el procedimiento es el siguiente:

```

CREATE OR REPLACE PROCEDURE restaurante.agregar_cliente(
    IN folio_orden character,
    IN rfc_dado character,
    IN nombre_cliente character varying,
    IN apellido_pat_cliente character varying,
    IN apellido_mat_cliente character varying,
    IN fecha_nacimiento_cliente date,
    IN estado_cliente character varying,
    IN cp_cliente character,
    IN colonia_cliente character varying,
    IN calle_cliente character varying,
    IN numero_cliente smallint,
    IN email_cliente character varying,
    IN razon_social_cliente text)
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
    rfc_cliente char(13);
    folio char(7);
BEGIN
    -- Se desea realizar facturación, por lo que el cliente brinda su información fiscal
    -- En este caso, es necesario analizar si el cliente ya ha realizado facturas.
    SELECT ro.folio INTO folio FROM restaurante.orden ro WHERE ro.folio = folio_orden;
    IF folio IS NOT NULL THEN
        -- Se agregó correctamente el folio
        -- Se valida que el RFC existe o no.
        SELECT rc.rfc INTO rfc_cliente FROM restaurante.cliente rc WHERE rc.rfc = rfc_dado;
        IF rfc_cliente IS NOT NULL THEN
            -- El cliente ya ha facturado anteriormente
            -- Únicamente se agrega el RFC a la orden
            UPDATE restaurante.orden SET rfc_cliente = rfc_dado WHERE restaurante.orden.folio = folio_orden;
        ELSE
            -- Se debe de crear el registro para el cliente y agregar el RFC a la orden
            INSERT INTO restaurante.cliente (rfc,nombre,apellido_pat,apellido_mat,fecha_nacimiento,estado,cp,colonia,calle,numero,email,razon_social)
            VALUES(rfc_dado,nombre_cliente,apellido_pat_cliente,apellido_mat_cliente,fecha_nacimiento_cliente,
            estado_cliente,cp_cliente,colonia_cliente,calle_cliente,numero_cliente,email_cliente,razon_social_cliente);
            UPDATE restaurante.orden SET rfc_cliente = rfc_dado WHERE restaurante.orden.folio = folio_orden;
        END IF;
    ELSE
        RAISE EXCEPTION 'No se encontró la orden con folio %. Verifique que está agregando la información correcta.', folio_orden;
    END IF;
END;
$BODY$;

```

Se observa que se obtienen los valores para los respectivos campos del cliente. Dentro del procedimiento se analiza lo siguiente:

- Primero se determina si existe la orden con el folio ingresado. Si no existe, se mandará un mensaje de error.
- Una vez validado el folio, se determina si el RFC existe o no. Con lo que hay dos posibilidades:
  - Si el RFC ya existe, significa que el cliente ya ha realizado facturas, por lo mismo, ya no es nece-



sario ingresar toda la información del cliente. Únicamente se actualiza el RFC asociada a la orden para poder realizar la factura (se observa que esto genera una solicitud innecesaria de información, sin embargo, posteriormente mediante la implementación de la página se soluciona lo anterior).

- En caso de que el cliente esté facturando por primera vez, se guardará toda su información y se asignará su RFC a la respectiva orden, con lo que será posible mostrar la factura.

Una vez explicados los procedimientos adicionales que se implementaron, se contemplará la implementación de los requerimientos solicitados por el cliente.

### Agregar productos a la orden

Para resolver la agregación de productos a las órdenes, se implementó la función siguiente:

```
CREATE OR REPLACE FUNCTION restaurante.agregar_producto_orden(
    folio_orden character,
    id_producto_agregado integer,
    cantidad_producto integer)
RETURNS void
LANGUAGE "plpgsql"
COST 100
VOLATILE PARALLEL UNSAFE
AS $BODY$
DECLARE
    info_orden char(7);
    info_producto restaurante.producto%ROWTYPE;
    alimento integer;
BEGIN
    -- Primero se analizará si el producto está disponible y si el folio es correcto
    SELECT ro.folio INTO info_orden FROM restaurante.orden ro WHERE ro.folio = folio_orden;
    IF info_orden IS NOT NULL THEN
        -- En este caso significa que si hay una orden con el folio indicado
        -- Se procede a analizar si existe el producto señalizado y está disponible
        SELECT * INTO info_producto FROM restaurante.producto rp WHERE rp.id_alimento = id_producto_agregado;
        IF info_producto.id_alimento IS NOT NULL THEN
            -- En este caso ya se verificó que el producto existe. Se analiza si está disponible
            IF info_producto.disponibilidad = 'true' THEN
                -- En dicho caso, se procede a agregar el producto a la tabla
                -- Hay que analizar si el producto existe ya dentro de la orden
                SELECT ocp.id_alimento INTO alimento FROM restaurante.orden_contiene_producto ocp WHERE ocp.id_alimento = id_producto_agregado AND ocp.folio = folio_orden;
                IF alimento IS NOT NULL THEN
                    -- Se actualizan los valores existentes
                    UPDATE restaurante.orden_contiene_producto SET cantidad = cantidad + cantidad_producto, total_producto = (cantidad + cantidad_producto) * info_producto.precio
                    WHERE id_alimento = alimento AND folio = folio_orden;
                ELSE
                    -- Se agrega a la orden
                    INSERT INTO restaurante.orden_contiene_producto(folio,id_alimento,cantidad,total_producto) VALUES(folio_orden,id_producto_agregado,cantidad_producto,info_producto.precio * cantidad_producto);
                END IF;
            ELSE
                RAISE EXCEPTION 'El producto no está disponible';
            END IF;
        ELSE
            RAISE EXCEPTION 'No se encontró el producto en el catálogo';
        END IF;
    ELSE
        RAISE EXCEPTION 'No se encontró la orden con el folio %.',folio_orden;
    END IF;
END
$BODY$;
```

La función sigue la siguiente lógica:



- Se recibe el folio de la orden, la cantidad del producto y el identificador del producto. Con lo anterior, se deberán de realizar algunas validaciones.
- Lo primero es verificar que exista una orden con dicho folio. En caso de existir, se procede a verificar los datos del producto.
- Se valida que el producto exista y esté disponible. En caso de que todo lo anterior cumpla, se tendrán dos posibles casos:
  - El producto ya se encuentra registrado en la orden, por lo que únicamente se deberán de actualizar los datos de la orden con la nueva cantidad.
  - El producto no ha sido solicitado antes en la orden, por lo mismo, se tiene que registrar la información nueva para la orden.
- La modificación de estos elementos dispara un trigger para modificar la información relacionada a las ventas de los productos y el total de la orden. El trigger es el siguiente:

```

CREATE OR REPLACE FUNCTION restaurante.producto_agregado()
RETURNS trigger
LANGUAGE 'plpgsql'
COST 100
VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
    precio_producto double precision;
    cantidad_consumida integer;
BEGIN
    -- Se necesita saber el precio y cuántos elementos se agregaron
    precio_producto := NEW.total_producto / NEW.cantidad;
    -- para la cantidad agregada
    cantidad_consumida := NEW.cantidad - OLD.cantidad;
    IF (TG_OP = 'UPDATE') THEN -- Se actualizó la información existente.
        -- En este caso, es necesario obtener la cantidad ingresada, así como el precio del producto
        UPDATE restaurante.orden SET total = total + precio_producto * cantidad_consumida WHERE folio = OLD.folio;
        UPDATE restaurante.producto SET ventas = ventas + cantidad_consumida WHERE id_alimento = OLD.id_alimento;
    ELSIF (TG_OP = 'INSERT') THEN
        UPDATE restaurante.orden SET total = total + NEW.total_producto WHERE folio = NEW.folio;
        UPDATE restaurante.producto SET ventas = ventas + NEW.cantidad WHERE id_alimento = NEW.id_alimento;
    END IF;
    RETURN NULL;
END;
$BODY$;
  
```

En el trigger se puede observar lo siguiente:

- Mediante el precio del producto y la cantidad consumida (los cuales se calculan mediante los datos de los registros modificados) se realiza lo siguiente:
  - En caso de actualización, se actualizan los valores de la orden para obtener el nuevo total, así como el total de ventas del producto específico ingresado (esto es de utilidad para obtener la información del platillo más vendido).



- En caso de inserción, se deberá de sumar el total del producto y actualizar las ventas del producto.

La diferencia entre una y otra radica en que, si el producto ya se encontraba registrado, la cantidad y el total involucran la información de los registros que ya se habían realizado, por lo que únicamente se debe de agregar la información que se obtiene con los datos que se acaban de ingresar. En el caso de inserción, como no se tiene registro de la información, el total del producto es un valor único que no se ha sumado con anterioridad al total de la orden, por lo mismo, es posible realizar la suma de forma directa (lo mismo con la cantidad de ventas del producto).

El trigger se asignó a la tabla de orden y productos. Esto se observa a continuación:

```
CREATE OR REPLACE TRIGGER agregar_producto
AFTER INSERT OR UPDATE
ON restaurante.orden_contiene_producto
FOR EACH ROW
EXECUTE FUNCTION restaurante.producto_agregado();
```

### Creación de Índice

Para este punto se tomó la decisión de crear un índice sobre la columna de nombre en la tabla de empleados, esto debido a que se desea mostrar la información de los empleados mediante búsquedas sobre dicha columna. Como el interés principal del cliente es poder visualizar a los empleados en la página WEB, es recurrente el acceso a la columna de nombre del empleado, la cual no es una llave principal, por lo mismo, se decidió realizar un índice sobre la columna de tipo HASH. El código es el siguiente:

```
CREATE INDEX IF NOT EXISTS nombre_empleado
ON restaurante.empleado USING hash
(nombre COLLATE pg_catalog."default")
```

Como el tipo de dato para el nombre es de character varying, se optó por un índice de tipo HASH. Este tipo de índice es bastante útil cuando únicamente se busca realizar comparaciones del tipo igualdad, por lo mismo, como se está buscando únicamente la coincidencia con la cadena almacenada en la columna del nombre, se determinó que el índice de tipo HASH sería el más útil.

### Información de las órdenes tomadas por un empleado

Para el análisis de este punto, se planteó la siguiente función:



```

CREATE OR REPLACE FUNCTION restaurante.info_ordenes(
    id_empleado integer)
RETURNS TABLE(cantidad smallint, total numeric)
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$
DECLARE
    ordenes SMALLINT = 0;
    total NUMERIC = 0;
    empleado restaurante.empleado%ROWTYPE;
    orden restaurante.orden%ROWTYPE;
BEGIN
    /*
        Dado un número de empleado, mostrar la cantidad de órdenes que ha registrado en el día así como el total que se ha pagado por dichas órdenes.
        Si no se trata de un mesero, mostrar un mensaje de ERROR
    */
    -- Primero se analiza si se encuentra el id entre los empleados
    SELECT * INTO empleado FROM restaurante.empleado emp WHERE emp.num_empleado = id_empleado;
    IF empleado.num_empleado = id_empleado THEN
        -- Se encontró al empleado. Se tiene que analizar si es mesero
        IF empleado.es_mesero THEN
            -- El empleado es un mesero
            -- Se tiene que ir recorriendo los valores
            FOR orden IN SELECT * FROM restaurante.orden WHERE restaurante.orden.id_mesero = id_empleado LOOP
                total := total + orden.total;
                ordenes := ordenes + 1;
            END LOOP;
            -- Validamos si hay datos
            IF ordenes = 0 THEN
                RAISE EXCEPTION 'El mesero no ha tomado ninguna orden';
            ELSE
                RETURN QUERY SELECT ordenes, total;
            END IF;
        ELSE
            RAISE EXCEPTION 'El número de trabajador ingresado no pertenece al de un mesero';
        END IF;
    ELSE
        RAISE EXCEPTION 'No existe un trabajador con el número de empleado señalizado';
    END IF;
END
$BODY$;

```

La función recibe el número del empleado para poder mostrar la información de las órdenes que ha realizado (siempre y cuando sea un mesero). En esta se utilizaron las siguientes variables:

- Órdenes: Servirá de contador para determinar el número de órdenes que ha tomado el mesero.
- Total: Irá realizando la suma de los totales de cada una de las órdenes actualmente registradas.
- Empleado: Servirá como información temporal del empleado. Permite validar que el número de empleado ingresado es válido.
- Orden: Permite obtener filas de la tabla de órdenes para iterar sobre dichos registros y obtener la información del total.

Con lo anterior en mente, el funcionamiento de la función se basa en validar que el número de empleado exista y, en caso de existir, validar que se trate de un mesero. Una vez que esto es contemplado, se procede a iterar sobre la tabla de órdenes para obtener todas aquellas asociadas al empleado seleccionado. Con esto, con cada registro se irá sumando al contador de órdenes tomadas, y con los registros, se irá sumando el total de cada orden al total general. Finalmente se realiza una última validación para señalar que el empleado no ha realizado ninguna orden.

### Platillo más vendido



Para la creación de la vista del platillo más vendido, se tiene el caso de uso de la variable adicional que se agregó a la tabla de productos, es decir, la columna de ventas. Esta se va actualizando conforme se agregan productos a las órdenes (como se mostró en el caso de agregar productos a la orden). Con esto, el procedimiento realizado se volvió bastante sencilla como se observa a continuación:

```
-- PROCEDURE: restaurante.producto_mas_vendido()

-- DROP PROCEDURE IF EXISTS restaurante.producto_mas_vendido();

CREATE OR REPLACE PROCEDURE restaurante.producto_mas_vendido(
    )
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    CREATE OR REPLACE VIEW restaurante.platillo_mas_vendido AS SELECT * FROM restaurante.producto rp
        WHERE rp.ventas IN(SELECT max(rp.ventas) FROM restaurante.producto rp);
END;
$BODY$;
ALTER PROCEDURE restaurante.producto_mas_vendido()
    OWNER TO postgres;
```

Únicamente se realiza una vista bajo el siguiente seguimiento de consultas:

- Primero se determinan las tuplas que tengan la mayor cantidad de ventas de los productos registrados en la tabla de productos.
- La consulta directa que crea la vista obtiene todos aquellos productos de la tabla cuyas ventas se encuentran dentro del resultado obtenido en la primera subconsulta.

#### Mostrar los platillos que no estén disponibles

En este caso no es necesario realizar una vista, por lo que únicamente se realizará una función que retorne una tabla con los productos que no están disponibles. Lo anterior se observa en la siguiente captura:



```

CREATE OR REPLACE FUNCTION restaurante.productos_no_disponibles(
)
RETURNS TABLE(nombre_producto character varying)
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$
BEGIN
    RETURN QUERY SELECT nombre FROM restaurante.producto WHERE disponibilidad = 'f';
END;
$BODY$;

```

### Generación de la factura

La implementación en este caso se centra en la generación de una vista, así como de la obtención de cursores que permite obtener la información de la factura en la página WEB, con lo que se mostrarán los datos de facturación en la página y, en la base de datos, se tendrá una vista con los datos del cliente y otra con el ticket. El código es el siguiente:

```

CREATE OR REPLACE FUNCTION restaurante.generar_factura(
    folio_orden character,
    ref1 refcursor,
    ref2 refcursor)
RETURNS SETOF refcursor
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$
DECLARE
    orden_solicitada restaurante.orden%ROWTYPE;
    folio_valido char(7);
    rfc_cliente char(13);
BEGIN
    Primero se debe de validar que el folio exista y tenga un RFC asociado
    SELECT * INTO orden_solicitada FROM restaurante.orden ro WHERE ro.folio = folio_orden AND ro.rfc_cliente IS NOT NULL;
    IF orden_solicitada IS NOT NULL THEN -- Se encontró la información para poder realizar la factura
        RAISE NOTICE '%', orden_solicitada.rfc_cliente;
        -- Se realiza la factura
        -- En este caso se va a optar por generar dos vistas:
        -- 1. Una se enfoca en los datos del cliente para imprimirse con formato
        -- 2. La segunda se enfocará en recopilar un ticket
        -- 3. Obtener los nombres de los productos
        OPEN ref1 FOR SELECT * FROM restaurante.cliente rc WHERE rc.rfc = orden_solicitada.rfc_cliente;
        RETURN NEXT ref1;

        OPEN ref2 FOR (SELECT rp.nombre, ocp.cantidad, ocp.total_producto FROM restaurante.orden_contiene_producto ocp
        JOIN restaurante.producto rp ON ocp.id_alimento = rp.id_alimento WHERE ocp.folio = folio_orden);
        RETURN NEXT ref2;
        EXECUTE 'CREATE OR REPLACE VIEW restaurante.datos_cliente AS ' || 'SELECT * FROM restaurante.cliente WHERE rfc = '''|| orden_solicitada.rfc_cliente ||'''';
        EXECUTE 'CREATE OR REPLACE VIEW restaurante.ticket AS ''';
        'SELECT rp.nombre, ocp.cantidad, ocp.total_producto FROM restaurante.orden_contiene_producto ocp JOIN restaurante.producto rp
        ON ocp.id_alimento = rp.id_alimento WHERE ocp.folio = '''|| folio_orden || '''';

    ELSE
        RAISE NOTICE 'No es posible realizar la factura de la orden ingresada';
    END IF;
$BODY$;

```

Para la generación se manda el folio de la orden junto con dos cursores que se retornarán como un conjunto de cursores (esto con el fin de rescatar la información de las dos tablas en Python). La función debe de validar que existe un folio de orden que ya tenga la información del RFC del cliente para poder obtener la in-



formación de facturación. En caso contrario, señalizará que no es posible. Los dos cursos que se mandan son:

- Los datos del cliente con el RFC asociada.
- - Para la segunda se realiza una unión entre los productos de la orden y la información que se encuentra en la tabla de productos. Para esto se debe de realizar la unión a través de los identificadores de productos.

Bajo estas dos sentencias, se vuelven a ejecutar con código dinámica, ya que, dentro de una función no es posible realizar vistas con variables o parámetros de la función. Por lo mismo, se buscó solucionar el problema mediante código dinámico (similar al que se utiliza para realizar operaciones desde Python). Con esto fue posible realizar las vistas dentro de la función y, del mismo modo, obtener los cursos de las tablas necesarias para la página WEB.

### Ventas por fecha

Para esto se realizaron dos funciones, ya que una recibe una fecha y la otra recibe un rango de fecha. Para la primera función se tiene lo siguiente:

```
CREATE OR REPLACE FUNCTION restaurante.ventas_por_fecha(
    fecha date)
RETURNS TABLE(cantidad integer, total double precision)
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$ 
DECLARE
    total_ventas double precision = 0.0;
    cantidad_ventas integer = 0;
    fila_ventas restaurante.orden%ROWTYPE;
BEGIN
    -- Para esto, se tiene el registro de las órdenes tomadas. Cada orden tiene el monto total y la fecha en formato timestamp
    FOR fila_ventas IN SELECT * FROM restaurante.orden LOOP
        -- Se verifica si la orden fue realizada en la fecha dada
        IF DATE(fila_ventas.fecha) = fecha THEN
            cantidad_ventas = cantidad_ventas + 1;
            total_ventas = total_ventas + fila_ventas.total;
        ELSE
            CONTINUE;
        END IF;
    END LOOP;
    RETURN QUERY SELECT cantidad_ventas, total_ventas;
END
$BODY$;
```

La función recibe una fecha y retorna la cantidad y el total generado (el comportamiento es bastante similar al utilizado para obtener la información de las órdenes). Por cada una de las órdenes, se analiza cuáles de ellas se realizaron en la fecha indicada, actualizando los valores de la cantidad de ventas y el total de ventas. Con los valores obtenidos, se retorna una tabla con la cantidad de ventas y el total de ventas.



Para el caso en el que se brinde la información de dos fechas, el código es el siguiente:

```

CREATE OR REPLACE FUNCTION restaurante.ventas_por_fecha(
    fecha_inferior date,
    fecha_superior date)
RETURNS TABLE(cantidad integer, total double precision)
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$
DECLARE
total_ventas double precision = 0.0;
cantidad_ventas integer = 0;
fila_ventas restaurante.orden%ROWTYPE;
BEGIN
    IF fecha_inferior < fecha_superior THEN
        -- Para esto, se tiene el registro de las ordenes tomadas. Cada orden tiene el monto total y la fecha en formato timestamp
        FOR fila_ventas IN SELECT * FROM restaurante.orden LOOP
            -- Se verifica si la orden fue realizada en la fecha dada
            IF DATE(fila_ventas.fecha) >= fecha_inferior AND DATE(fila_ventas.fecha) <= fecha_superior THEN
                cantidad_ventas = cantidad_ventas + 1;
                total_ventas = total_ventas + fila_ventas.total;
            ELSE
                CONTINUE;
            END IF;
        END LOOP;
        RETURN QUERY SELECT cantidad_ventas, total_ventas;
    ELSE
        RAISE EXCEPTION 'Ingrese un intervalo válido. La fecha inferior es mayor que la superior.';
    END IF;
END
$BODY$;

```

El comportamiento es prácticamente lo mismo. En este caso, lo primero que se realiza es validar que el intervalo tenga sentido, es decir, que la fecha superior sea mayor a la fecha inferior. Y, en el caso de las comparaciones, se debe de validar que la fecha esté dentro del intervalo, en cuyo caso se realizará el aumento del contador y la suma del total.

### Obtener información de empleados

Finalmente se agregó una función adicional para poder obtener la información completa de los empleados (incluyendo los teléfonos de contacto). Esto se realizó con el fin de obtener la información del empleado en la página WEB. La función es la siguiente:



```
CREATE OR REPLACE FUNCTION restaurante.obtener_info_empleados(
    nombre_empleado character varying)
RETURNS TABLE(num_empleado integer, rfc character, nombre character varying, apellido_pat character varying,
apellido_mat character varying, fecha_nacimiento date, edad smallint, estado character varying, cp character,
colonia character varying, calle character varying, numero smallint, foto bytea, es_mesero boolean, horario character varying, es_administrativo boolean,
rol character varying, es_cocinero boolean, especialidad character varying, sueldo double precision, teléfonos text)
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000

AS $BODY$
DECLARE
    cadena character varying;
BEGIN
    cadena := '%' || rtrim(ltrim(lower(nombre_empleado))) || '%';
    RETURN QUERY
    SELECT * FROM restaurante.empleado re JOIN (SELECT rt.num_empleado, STRING_AGG(telefono::TEXT, ', ') AS teléfonos
    FROM restaurante.telefono rt GROUP BY rt.num_empleado)
    USING (num_empleado) WHERE lower(re.nombre) LIKE cadena ORDER BY num_empleado;
END
$BODY$;
```

La función debe de recibir el nombre del empleado. Para asegurar posibles opciones de selección, se realiza una modificación de la cadena recibida para asegurar que no tenga espacios en blanco y todo esté en minúscula. En el caso de los porcentajes, se utiliza para emplear la instrucción LIKE y poder obtener aquellos registros que contengan el posible nombre deseado.

Para el caso de los teléfonos, se obtienen todos los números asociados a un solo empleado mediante la primera subconsulta, esto mediante la agrupación mediante el número de empleado y utilizando la función de agregación STRING AGG, que permite obtener el arreglo de teléfonos asociados a un solo empleado. Posteriormente esta tabla obtenida de la subconsulta se une con la tabla de empleados, obteniendo de este modo toda la información asociada a dicho trabajador.

## 7. Presentación

Se decidió realizar una página WEB mediante el uso de HTML y CSS. Para el enlace y el comportamiento se utilizó Python con la biblioteca Flask y Psycopg2. Flask permite enlazar el código de Python con el HTML. Se optó por el uso de este aproximamiento debido a la facilidad de uso de Python, así como la familiaridad del equipo con dicho lenguaje de programación. Igualmente, debido a que se solicitaba procesamiento de imagen, Python resulta muy noble y amigable para realizar manejo de imagen y transformaciones a binario (cuestión de principal interés para el proyecto). Igualmente, mediante la librería de Psycopg2 es bastante sencillo realizar la conexión a la base de datos, lo que permite una interacción bastante dinámica y sencilla del programa PostgreSQL y Python.



```

from flask import Flask, request, render_template
import psycopg2
import base64
from io import BytesIO
from PIL import Image
import os
from psycopg2 import sql
from io import BytesIO

host = "localhost"
database = "restaurante_bd"
user = "postgres"
password = "da4njilro0?."

app = Flask(__name__)

```

En la imagen anterior se muestra la importancia de las bibliotecas mencionadas. Igualmente, se muestran los datos de conexión a la base de datos local.

```

try:
    #Parametros para conexión a la base
    connection = psycopg2.connect(host=host,
                                  database=database,
                                  user=user,
                                  password=password)
    #

    # Crear un cursor para ejecutar consultas
    cur = connection.cursor()

    #Instrucción a ejecutar en sintaxis postgres
    instruction = f"SELECT * FROM restaurante.info_ordenes(\'{id_empleado}\')"
    #Los datos son las variables declaradas

    #cur.execute para ejecutar la instrucción
    cur.execute(instruction)
    records = cur.fetchall()
    # Se tiene la información de las órdenes que ha tomado el empleado.
    for record in records:
        print(record)
    cur.close()
    connection.close()

```

La siguiente imagen muestra un ejemplo de cómo es posible enlazar la base de datos con el código de Python y realizar consultas mediante sentencias estilo SQL dinámico. Todo lo anterior se realiza sobre un cursor que interactúa con la base y obtiene la información de esta.

El enlace entre Python y HTML se muestra a continuación:



```
form action="/agregar_empleado" method="POST" autocomplete="off" class="form" enctype="multipart/form-data" id="formulario">


## Agregar empleado



</span>



</span>


```

```
@app.route('/agregar_empleado', methods=['POST'])
def agregar_empleado():
    if request.method == 'POST':
        #Variables para ingresar a la base
        ruta_destino = ""
        rfc = request.form['rfc']
        nombre = request.form['nombre']
        appat = request.form['appat']
        apmat = request.form['apmat']
        fechanac = request.form['fechanac']
        edad = request.form['edad']
```

Esto permite realizar la obtención de información de HTML y Python. Para mandar información desde Python hacia HTML se realiza mediante la función de renderizado de plantilla como se muestra a continuación:

```
return render_template('mostrar-factura.html', msg='Formulario enviado', cliente=tbl1[0], ticket=tbl2, orden=tbl3[0], fecha=str(tbl3[0][1]).split('.')[0])
```

Se manda la información a través de la asignación de variables y el renderizado de cierto documento HTML. Con estas condiciones es posible agregar bloques de código dentro de HTML, ya que Flask los interpreta y generar el renderizado deseado:

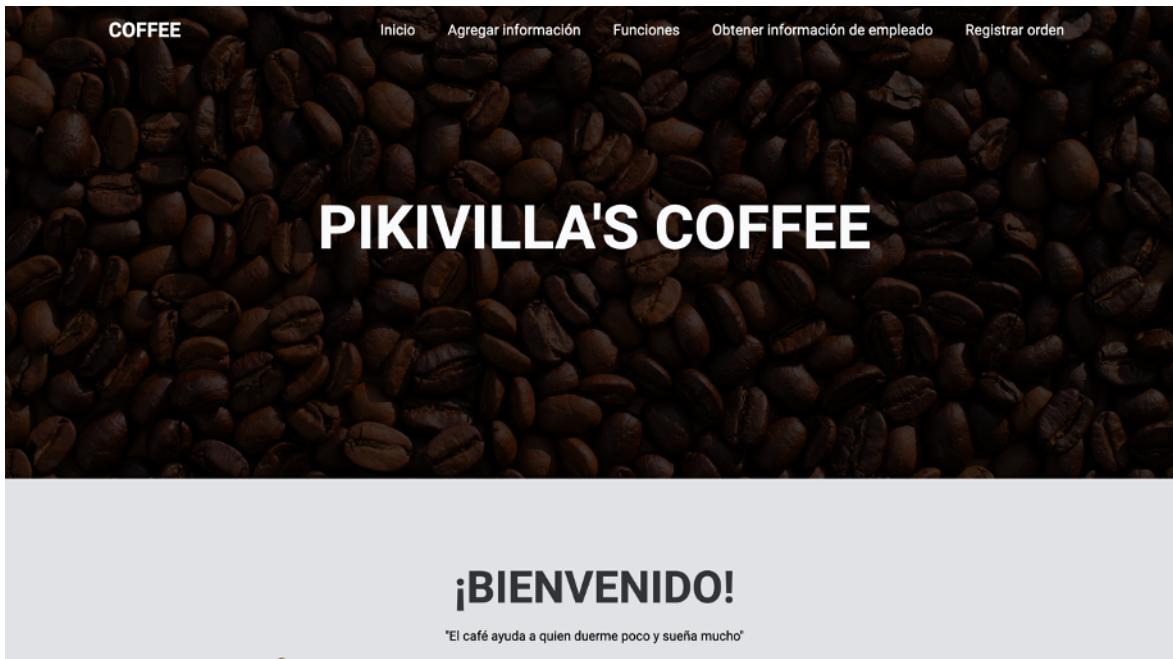


```
<hr class=".meun_line-ventas"><br>
{& for tickets in ticket %}
<p class="ticket_prod">{{tickets.0}}</p><p class="ticket_cant">{{tickets.1}}</p><p class="ticket_price">{{tickets.2}}</p><br>
{& endfor %}
<p class="ticket_data">Total</p><p class="ticket_value-t">{{orden.2}}</p><br>
</article>
<br><h3 class="title_product">Datos Adicionales</h3>
<article class="item">
<p class="ticket_data-h">Datos</p><p class="ticket_value-h">Valor</p><br>
<hr class=".meun_line-ventas"><br>
<p class="ticket_data">Folio</p><p class="ticket_value">{{orden.0}}</p><br>
<p class="ticket_data">Fecha</p><p class="ticket_value">{{fecha}}</p><br>
<p class="ticket_data">Mesero</p><p class="ticket_value">{{orden.3}}</p><br>
```

Con esto se observa la lógica general empleada para la implementación de las solicitudes del cliente. En este caso, específicamente se solicitó la posibilidad de mostrar la información de los clientes, sin embargo, se decidió implementar todas las funcionalidades dentro de la página WEB. A continuación, se muestra el resultado generado para las funciones de interés:



## Interfaz principal



En la interfaz se brinda una presentación del establecimiento. En la parte superior es posible identificar diferentes pestañas de navegación. Las de principal interés son las relacionadas a las funciones y la obtención de información de los empleados. La pestaña de agregar información y registrar orden son de utilidad para insertar información a la base de datos.

## Obtener la información de empleado



Este campo solicita el nombre del empleado para buscarlo dentro de los registros de la base de datos. En este caso, únicamente existe un empleado llamado Carlos, con lo que se obtiene la siguiente tarjeta del empleado:



COFFEE

Inicio

Mostrar un menú

**Ernesto Danjiro López Sugahara**

\* Fecha de nacimiento: 2001-11-28

\* Edad: 21

\* Dirección: Viaducto Río Becerra 137, Col. Nápoles, C.P. 03810 CDMX

Puestos de Trabajo  
El empleado es administrativo con un rol de Jefe

Si no se especifica un nombre, o hay varios empleados con el mismo nombre de pila, se mostrará un listado con la posibilidad de seleccionar al empleado con dicho nombre (o la lista de todos los empleados):

COFFEE

Inicio

Buscar empleado:

Ingresar el nombre de pila:

Agregar

Se obtuvieron los registros siguientes:

✓ Empleado 1: Ernesto Danjiro López Sugahara  
Empleado 2: Carlos Miguel Villegas Viniegra  
Empleado 3: Santiago Rodríguez Kobay  
Empleado 4: Fernanda Pérez Martínez  
Empleado 5: Frida Contreras Jiménez  
Empleado 6: Carla Martínez Pérez  
Empleado 7: Carla Romero Pérez  
Empleado 8: Ángel Ramírez Martínez  
Empleado 9: Fernando Sávater Amador



### *Funciones implementadas*

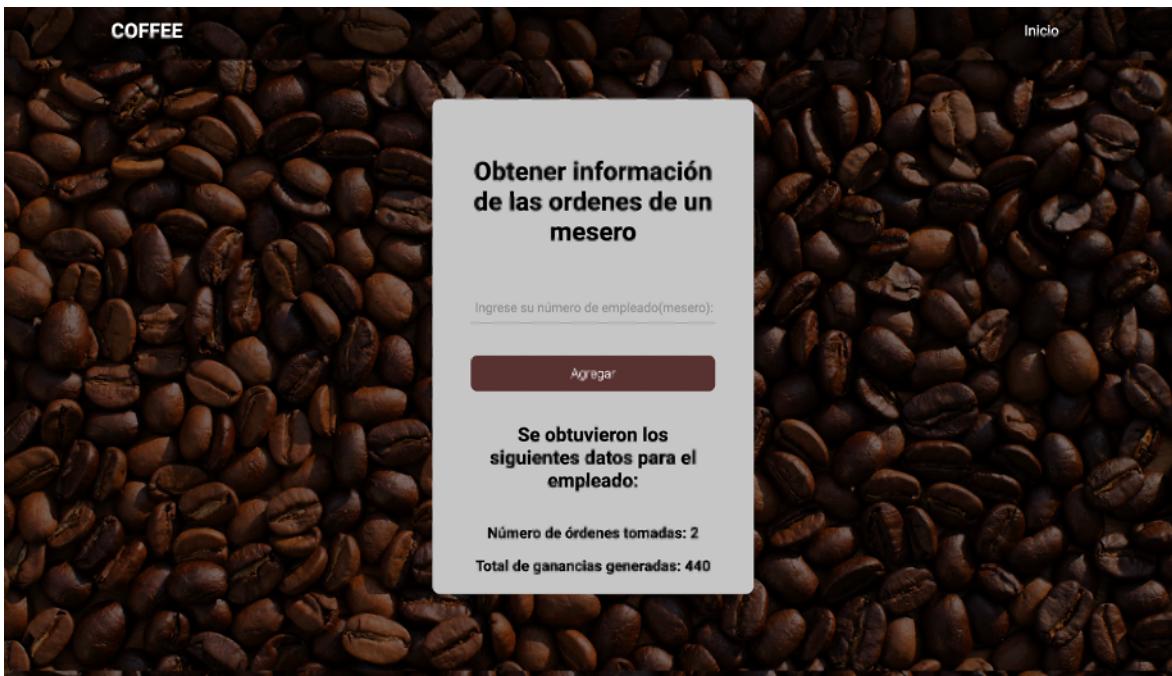
Las funciones implementadas, específicamente las que regresan cierta información de interés o estadística, se observan en la siguiente pestaña de navegación:



A continuación, se muestra el funcionamiento de cada una de ellas:

#### Información de ordenes:

Dado el número de un mesero, se mostrará la cantidad de ventas que ha realizado y el total generado por dichas ventas. Con esto, se observa lo siguiente:



En caso de ingresar el número inválido, se mostrará un mensaje señalizando que hubo un error.

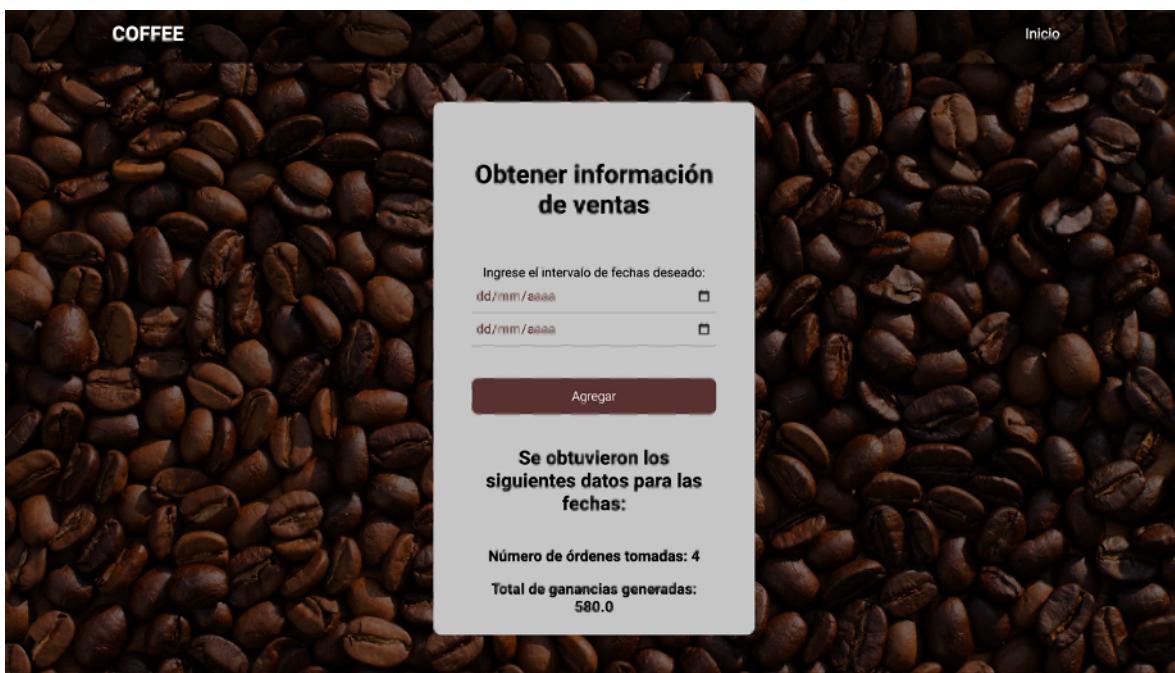
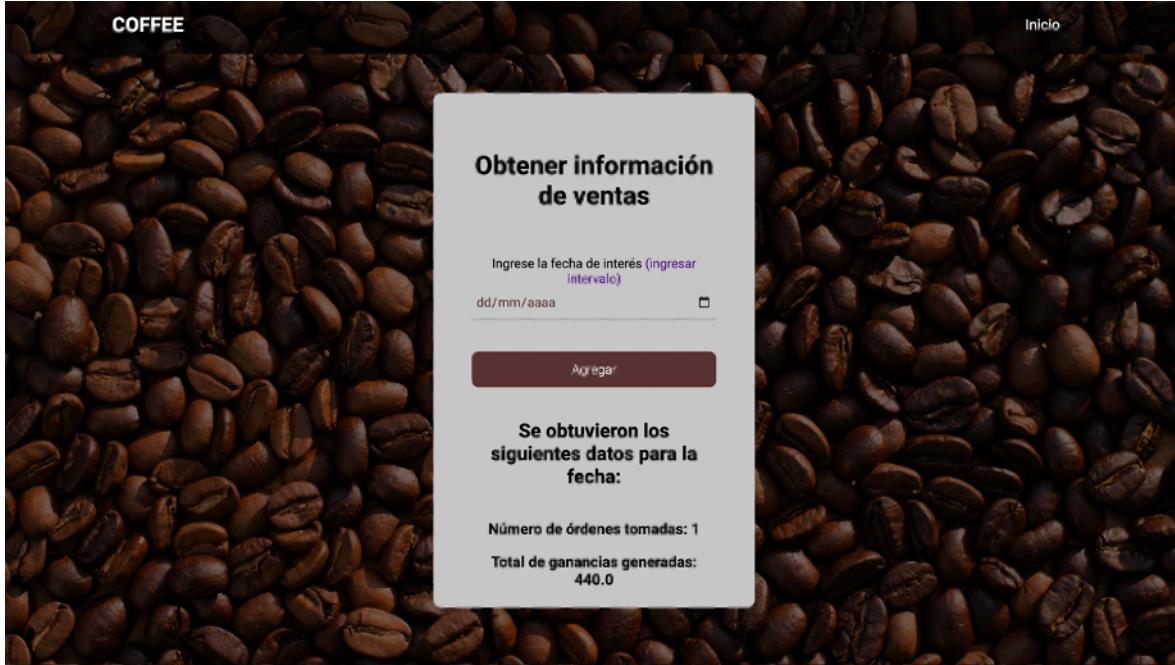
Mostrar platillos no disponibles:





Mostrar ventas por fecha:

En este caso se tienen dos posibilidades, agregar una fecha y mostrar la cantidad de ventas y el total, así como la posibilidad de ingresar un intervalo de fechas. Esto se observa a continuación:





Mostrar platillos mas vendidos:

En este caso se mostrará un listado de todos aquellos productos que tienen la cantidad de ventas más alta. Se mostrará una tabla con la siguiente forma:

Productos mas vendidos			
<i>pikivillas's coffee</i>			
ID	Producto	Ventas	Precio
id:2	Café Americano	4	30.0
id:4	Sandwich	4	70.0

Generar factura:

Dado el folio de una orden, se mostrará la factura (en caso de existir). En caso contrario se solicitará la información del cliente para realizar el registro y, con lo mismo, mostrar la factura final.

Ticket		
<i>pikivillas's coffee</i>		
Datos	Cliente	Valor
RFC		SALP960223D12
NOMBRE		Pedro
Ap. Paterno		Sánchez
Ap. Materno		Lezama
Fecha Nac.		1996-02-23
Correo		pedro@gmail.com
Razón Social		Razón social
Datos	Vivienda	Valor
Ciudad		CDMX
Código Postal		43132
Colonia		Narvarte
Calle		Calle 3
Número		12
Productos	Productos	Precio
	Cantidad	
Café Americano	4	120.0
Enchiladas	3	180.0
Sandwich	2	140.0
Total		440.0
Datos Adicionales	Datos	Valor
Folio		ORD-001
Fecha		2023-11-14 21:41:22
Mesero		2



## 8. Conclusiones

*López Sugahara Ernesto Danjiro:*

A través de la realización del proyecto fue posible comprender y aterrizar varias ideas de forma mucho más tangible mediante la creación de todo un sistema que permite manipular y trabajar con los datos de una base de datos. La verdad fue un proyecto bastante enriquecedor donde cada día surgían ideas y retos que resolver para intentar llevar un poco más allá la entrega del trabajo. En general, el diseño de la base fue la parte más simple, sin embargo, desde un inicio fue necesario definir el cómo se iba a enlazar y representar esta información en el sistema, ya que esto permitió realizar un modelo de datos mucho más flexible y que permitiera obtener las solicitudes planteadas mediante consultas y operaciones menos complicadas de las que pudieron haber sido. En cuestión de la implementación de la segunda, fue en este punto donde más complicaciones se presentaron a la hora de realizar el trabajo, esto debido a varias cuestiones: la emoción de querer agregar más elementos y el desconocimiento de cómo trabajar todos los elementos que se solicitaban. Lograr el comportamiento que obtuvimos en la página final fue algo sin duda extenuante, pero increíblemente placentero.

La selección de qué elementos se utilizarían para trabajar sentimos que fue un verdadero acierto, ya que la mayoría del equipo ya tenía familiaridad con el lenguaje de programación Python, así como la estructura básica de HTML y CSS, por lo mismo, el poder haber enlazado estos entornos a través de Flask, e igualmente enlazar Python con PostgreSQL a través de psycopg2, aunque fue complicado en un inicio, fue mucho más sencillo de lo que hubiera sido intentar utilizar alguna otra herramienta que se desconociera en su totalidad. Esto permitió agilizar el desarrollo de los elementos solicitados una vez que se entendió la lógica de enlazamiento. Igualmente, la selección de Python facilitó bastante la solución de uno de los puntos más intimidantes del proyecto: la fotografía del empleado. Con todo lo anterior, se puede concluir que se cumplió el propósito del proyecto para mostrar los conocimientos adquiridos a lo largo de la materia, así como algunos conocimientos adicionales que nos podrían beneficiar en nuestra futura carrera.

*Ramirez Martinez Luis Angel:*

Este proyecto representa un ejemplo destacado de cómo la tecnología de bases de datos y el desarrollo web pueden trabajar juntos para ofrecer soluciones efectivas en el ámbito empresarial. A través de un enfoque detallado y una implementación cuidadosa, el equipo logró no solo cumplir con los requerimientos del cliente, sino también proporcionar una plataforma robusta y flexible para el manejo de datos. Este proyecto demuestra la importancia de una comprensión profunda de los requisitos del cliente, un diseño detallado y una implementación cuidadosa en proyectos de desarrollo de sistemas de información. El resultado es un sistema que no solo mejora la eficiencia operativa del restaurante, sino que también sienta las bases para futuras expansiones y mejoras, demostrando así un claro logro de objetivos y un aprendizaje significativo en el campo del desarrollo de bases de datos y aplicaciones web.

*Rodríguez Kobe Santiago:*



Desde mi punto de vista, se lograron los objetivos y requerimientos descritos al momento de plantear el proyecto. Me pareció un proyecto completo, ya que se aterriza de forma más concisa tanto lo aprendido a lo largo del curso, como los conocimientos que hemos adquirido a lo largo de la carrera. Tal vez, tanto en laboratorio como en teoría, algunos temas pueden quedar al aire, sin entender mucho como se podrían emplear en un contexto tangible, gracias a la elaboración de este programa y proyecto, se entendió la importancia de temas y el como aplicarlos.

El proyecto se planteó cuando aún no se conocía del PL/pgSQL, sólo teníamos los conocimientos para realizar consultas. Gracias a esto, se tuvo que investigar y aprender por nuestra cuenta tanto a como trabajar y manejar procedimientos y funciones como a conectar la base al lenguaje de programación Python y HTML y CSS, por medio del framework Flask.

Una gran parte del curso, me atrevería a decir que más de la mitad del mismo, se enfocó en entender los requerimientos y como plantearlos de la mejor manera. Este enfoque me pareció muy acertado ya que al momento de comenzar con el análisis de requerimientos, no fue tan complicado entenderlo y plantear una primera solución a estos.

Creo que tenemos áreas de oportunidad, como podría ser la organización, comunicación y el como administrar la carga de trabajos y el delegar tareas.

*Soto Rivera Marco Antonio:*

Este proyecto nos ayudó a implementar los conocimientos adquiridos en clase. Si bien hemos realizado prácticas en el laboratorio, esta actividad nos ayudó a tener una mejor comprensión de cómo se pueden aplicar las bases de datos, además de cómo aplicar cada una de las etapas de desarrollo. Partimos desde el análisis del problema hasta llegar a la implementación de nuestros modelos. Experimentamos con el uso de procedimientos y funciones que nos resultaron de mucha ayuda para lograr los objetivos del proyecto. Una de las cosas que más tiempo requirió fue la imagen, ya que tuvimos que ver, a partir de los lenguajes que manejábamos, cómo se podría obtener el código fuente de la imagen para luego ingresarla a la base de datos y, posteriormente, obtenerla y mostrarla.

La elección de utilizar Python, en conjunto con las herramientas específicas como Flask y la librería "psycopg2" para la vinculación de la base de datos, y en caso de poder transformar la imagen a bits y obtener la imagen nuevamente "pillow" demostró ser una decisión acertada. La versatilidad de Python y la facilidad de integración de estas librerías facilitaron significativamente tanto el enlace de nuestra página web como la implementación del proyecto en su totalidad. Además, el hecho de que el equipo ya contara con conocimientos previos sobre estas tecnologías permitió una colaboración eficiente y la posibilidad de que cada miembro aportara de manera activa a la ejecución exitosa del proyecto. Esta experiencia no solo fortaleció nuestras habilidades técnicas, sino también nuestra capacidad para trabajar en equipo y resolver desafíos de manera conjunta.

*Villaseñor Venegas Carlos Miguel:*



Gracias a este proyecto pudimos aprender principalmente sobre 2 conceptos, el primero es la creación de la base de datos, esta parte nos ayudo a terminar de asimilar los temas vistos en clase y en laboratorio, ya que, hasta este momento, solo habíamos visto ejemplos aislados de como trabajar cada una de las etapas de diseño e implementación, a diferencia de este proyecto, donde tuvimos que trabajar con el mismo caso durante todas las fases. Además nos sirvió para practicar el uso de comandos SQL para trabajar con lenguajes como DDL y DML, y a terminar de entender el uso de triggers, funciones, procedimientos y vistas, que solo vimos de manera teórica y con ejemplos en clase.

El otro concepto del proyecto fue el aprender a unir la base de datos ya creada para hacer uso de ella desde una página de internet. Esta parte resultó mas educativa porque nos permitió darnos una idea de conceptos completamente nuevos, como: las diferentes alternativas para hacer una página web, los elementos que hay que considerar, las opciones para ligar la base de datos, el uso de HTML y CSS, entre otras cosas.