Final Report
Restaurant Management System

Created By:

Maddie McDowell
mmcdowell0490@sdsu.edu

Nhan Le
nle7674@sdsu.edu

Dania Bawab
dbawab9623@sdsu.edu

Laurie Liu
lliu5194@sdsu.edu

Kevin Ramirez
kramirez9560@sdsu.edu

Reported To:

Professor Morteza Safaei Pour

I. Introduction:

As competition within the restaurant industry increases and customer expectations continue to rise, there is a growing need for technology to control customer flow, staff operations, orders, and payments with precision. However, coordinating reservations, walk-ins, table assignments, orders, and payments can be challenging without a robust database management system (DBMS).

Our Restaurant Database Management System (RDBM) focuses on a full-service restaurant environment in which customers interact with the business through channels (walk-ins, reservations, or waitlist). Employees manage visits, take orders, serve items, and process payments. The database must support these interactions while enforcing business rules that maintain data integrity.

This system also standardizes operational processes across branches, enabling management to monitor performance, track employee activity, analyze sales, and deliver customer experiences.

1.1. Business Rules

Visit, Customer, and Table Management Rules

- Customers may have zero, one, or many visits.; each visit must be associated with exactly one customer and must record its creation time and party size.
- Each branch can host many visits, tables, and employees.
- Orders and payments are indirectly associated with a branch through the visit that created the order.
- Each table belongs to exactly one branch and may serve many visits over time, but only one active visit at a time (enforced via procedure), assignment of a table to a visit is optional.

- Visits are categorized by a VisitType attribute as walk-in (no matching waitlist or reservation record), waitlist, or reservation, and every visit must fall into exactly one of these types.
- Waitlist entries correspond to exactly one visit and a visit may appear on the waitlist at most once; each waitlist entry must store a quoted wait time and a current status such as waiting, seated, left.
- Reservation records also correspond to exactly one visit, and a visit may have at most one reservation; each reservation must store the reserved time.
- A visit may be walk-in only, a waitlist visit, or a reservation visit; meaning a visit cannot be both a waitlist and a reservation at the same time.

Order & OrderItem Rules

- Each visit may have zero, one, or many orders; each order must belong to exactly one visit.
- Each order must be taken by exactly one employee; employees may take zero, one, or many orders.
- Each order may contain one or many order items; each order item must belong to exactly one order.
- Each order item must have a positive quantity and stores price and tax as snapshots (UnitPriceSnapshot, TaxAmountSnapshot) so that historical orders remain accurate over time.
- Orders track SubTotal, TaxAmount, and TotalDue, matched to their order items.
- OrderStatus and timestamps (Created_at, Updated_at) track the lifecycle of the order such as open, in_kitchen, served, closed, voided.

Payment Rules

- Payments apply to a single order; orders may have zero, one, or many payments (supporting split checks or mixed methods).
- Payments methods include cash, card, platform, or gift card.
- Payment records include amount, optional tip, method, status, and created time.
- Total successful payments for an order must not exceed the TotalDue unless overpayment is intentionally allowed.
- Each payment has a status (e.g., pending, captured, refunded, failed) indicates whether the payment has been successfully completed.

Customer Rules

- Customers are identified by unique CustomerID.
- Customer information includes first and last name, phone and email.

Employee Rules

- Employees must have a unique EmployeeID.
- Each employee has exactly one primary role; each role may be held by zero, one, or many employees.
- Employees belong to one branch and may take multiple orders.

1.2. Use Cases

Hospitality Industry:
- Full-service restaurants
- Fast-casual dining
- Cafes and bistros
- Franchises restaurant

Event & Catering Service:
- Private dinning
- Banquet halls
- Hotel restaurants

II. Conceptual Data Modeling and Database Design

2.1. Conceptual Data Modeling:

The core of the data model includes customers, visits, reservations, waitlist operations, employees, orders, order items, and payment processing.

Customer:

- Attributes: CustomerID, FirstName, LastName, Phone, Email
- Relationship:
    - One-to-Many with Visit: one customer can create one or multiple visits; each visit is associated with exactly one customer.

Visit (Supertype)

- Attributes: VisitID, CustomerID, BranchID, TableID, TimeCreated, PartySize, VisitType.
- Relationships:
    - Many-to-One with Customer: each visit is made by one customer.
    - Many-to-One with Branch: each visit occurs at one branch.
    - Many-to-One (Optional) with Table: a visit may be assigned to one table; a table may serve many visits.
    - One-to-One (Optional) with Waitlist: a visit may have one waitlist record, or none.
    - One-to-One (Optional) with Reservation: a visit may have one reservation record, or none.
    - One-to-Many with Order: a visit can generate zero, one, or many orders.

Reservation (Subtype of Visit)

- Attributes: ReservationID, VisitID, Reserved_time
- Relationships:

    - One-to-One (Optional) with Visit: each reservation record corresponds exactly to one visit; a visit can have at most one reservation record.

Waitlist (Subtype of Visit)

- Attributes: WaitlistID, Quoted_wait_time, Status
- Relationships:
    - One-to-One with Visit: each waitlist record corresponds to one visit; a visit can have at most one waitlist record.

Branch

- Attributes: BranchID, Name, Location, Status.
- Relationships:
    - One-to-Many with Employee: one branch employs zero, one, or many employees; each employee belongs to one branch.
    - One-to-Many with Table: one branch contains one, zero, or many tables; each table belongs to one branch.
    - One-to-Many with Visit: one branch can host zero, one, or many visits; each visit occurs at one branch.

Table

- Attributes: TableID, BranchID, Section, Capacity, Status
- Relationships:
    - Many-to-One with Branch: each table belongs to one branch;a branch may have many tables.
    - One-to-Many (Optional) with Visits: a table may be used in zero, one, or many visits; each visit may be assigned to zero or one table.

Order

- Attributes: OrderID, VisitID, EmployeeID, OrderStatus, SubTotal, TaxAmount, TotalDue, Created_at, Updated_at
- Relationships:
    - Many-to-One with Visit: each order is placed during exactly one visit; a visit may have zero, one, or many orders.
    - Many-to-One with Employee: each order is taken by exactly one employee; an employee may take zero, one, or many orders.
    - One-to-Many with OrderItem: each order must have one or many order items; each order item belongs to exactly one order.
    - One-to-Many (Optional) with Payment: each order may have zero, one, or many payments; each payment applies to exactly one order.

OrderItem

- Attributes: OrderItemID, OrderID, Quantity, UnitPriceSnapshot, TaxAmountSnapshot
- Relationships:
    - Many-to-One with Order: each order item belongs to exactly one order; each order has one or many order items.

Employee

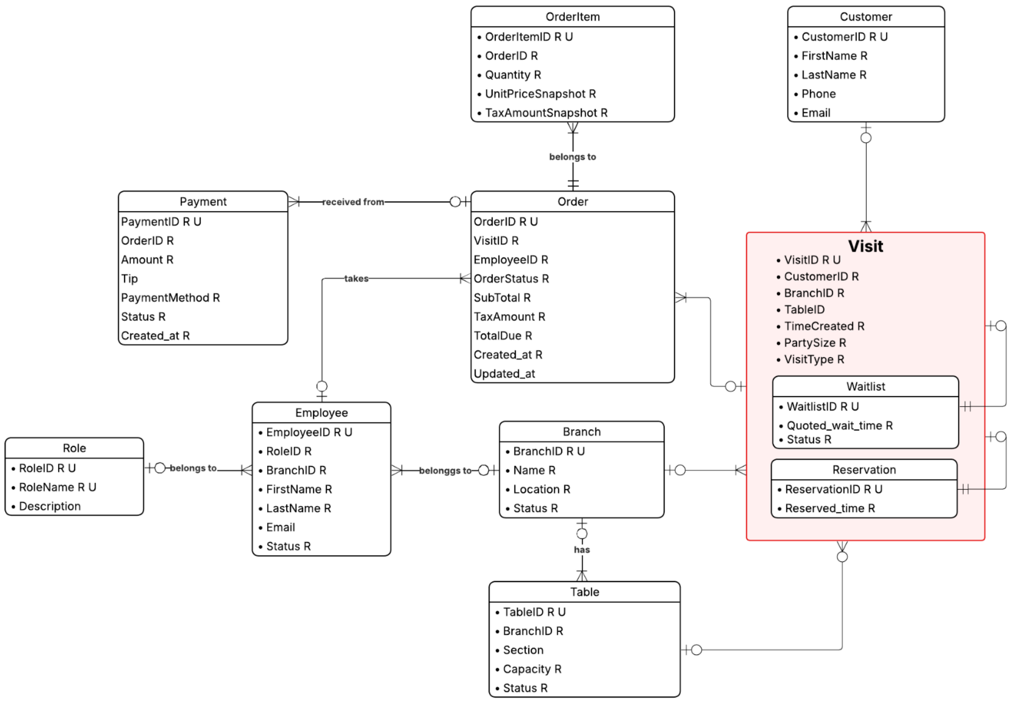
- Attributes: EmployeeID, RoleID, BranchID, FirstName, LastName, Email, Status
- Relationships:
    - Many-to-One with Branch: each employee works at exactly one branch; a branch may employ many employees.
    - Many-to-One with Role: each employee has exactly one primary role; a role may be held by zero, one, or many employees.
    - One-to-Many with Order: an employee may take zero, one, or many orders; each order is taken by exactly one employee.

Role

- Attributes: RoleID, RoleName, Description
- Relationships:
    - One-to-Many with Employee: a role (e.g., Server, Manager, Chef) may be held by zero, one, or many employees; each employee has exactly one role.

Payment

- Attributes: PaymentID, OrderID, Amount, Tip, PaymentMethod, Status, Created_at
- Relationships:
    - Many-to-One with Order: each payment is associated with exactly one order; an order may have zero, one, or many payments (supporting split checks or multiple methods).

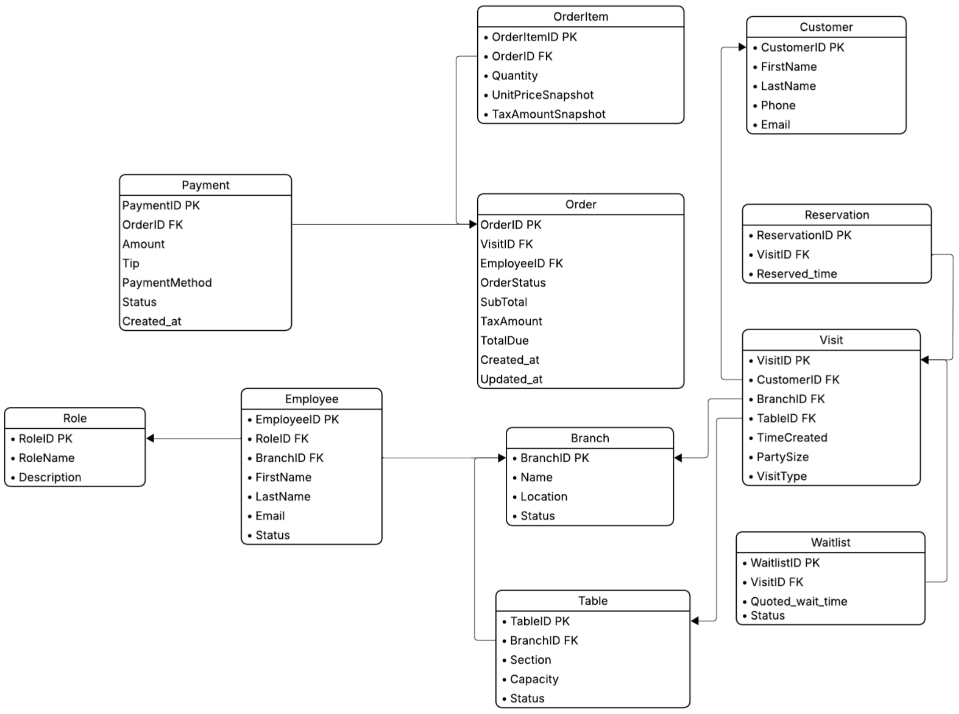


2.2. Transform ER/EER Model to Relational Model:

1. **Customer Table:**
    - Attributes: CustomerID (Primary Key), FirstName, LastName, Phone, Email
    - Primary Key: CustomerID
2. **Branch Table:**
    - Attributes: BranchID (Primary Key), Name, Location, Status
    - Primary Key: BranchID
3. **Table Table:**
    - Attributes: TableID (Primary Key), BranchID (Foreign Key), Section, Capacity, Status
    - Primary Key: Table ID
    - Foreign Key: BranchID references Branch(BranchID)
4. **Role Table:**
    - Attributes: RoleID (Primary Key), RoleName, Description
    - Primary Key: RoleID
5. **Employee Table:**
    - Attributes: EmployeeID (Primary Key), RoleID (Foreign Key), BranchID (Foreign Key), FirstName, LastName, Email, Status

- Primary Key: EmployeeID
- Foreign Keys:
  - RoleID references Role(RoleID)
  - BranchID references Branch(BranchID)

6. **Visit Table:**
   - Attributes: VisitID (Primary Key), CustomerID (Foreign Key), BranchID (Foreign Key), TableID (Foreign Key, optional), TimeCreated, PartySize, VisitType
   - Primary Key: VisitID
   - Foreign Keys:
     - CustomerID references Customer(CustomerID)
     - BranchID references Branch(BranchID)
     - TableID references Table(TableID)
7. **Waitlist Table (subtype extension of Visit):**
   - Attributes:WaitlistID (Primary Key), VisitID (Foreign Key, UNIQUE), Quoted_wait_time, Status
   - Primary Key: WaitlistID
   - Foreign Keys:
     - VisitID references Visit(VisitID)
8. **Reservation Table (subtype extension of Visit):**
   - Attributes: ReservationID (Primary Key), VisitID (Foreign Key, UNIQUE), Reserved_time
   - Primary Key: ReservationID
   - Foreign Keys:
     - VisitID references Visit(VisitID)
9. **Orders Table:**
   - Attributes: OrderID (Primary Key), VisitID (Foreign Key), EmployeeID (Foreign Key), OrderStatus, SubTotal, TaxAmount, TotalDue, Created_at, Updated_at
   - Primary Key: OrderID
   - Foreign Keys:
     - VisitID references Visit(VisitID)
     - EmployeeID references Employee(EmployeeID)
10. **OrderItem Table:**
    - Attributes: OrderItemID (Primary Key), OrderID (Foreign Key), Quantity, UnitPriceSnapshot, TaxAmountSnapshot
    - Primary Key: OrderItemID
    - Foreign Keys:
      - OrderID references Orders(OrderID)
11. **Payment Table:**
    - Attributes: PaymentID (Primary Key), OrderID (Foreign Key), Amount, Tip, PaymentMethod, Status, Created_at
    - Primary Key: PaymentID
    - Foreign Keys:
      - OrderID references Orders(OrderID)



III. Database Implementation

```
CREATE DATABASE IF NOT EXISTS restaurantdb;

USE restaurantdb;

-- 1) DROP (dependency order)
DROP TABLE IF EXISTS Payment;
DROP TABLE IF EXISTS OrderItem;
DROP TABLE IF EXISTS Orders;
DROP TABLE IF EXISTS Reservation;
DROP TABLE IF EXISTS Waitlist;
DROP TABLE IF EXISTS Visit;
DROP TABLE IF EXISTS TableInfo;
DROP TABLE IF EXISTS Employee;
DROP TABLE IF EXISTS Customer;
DROP TABLE IF EXISTS Branch;
DROP TABLE IF EXISTS Role;

-- 2) TABLE DEFINITIONS
CREATE TABLE Role (
   RoleID INT AUTO_INCREMENT PRIMARY KEY,
   RoleName VARCHAR(50) UNIQUE NOT NULL,
   Description TEXT
);

CREATE TABLE Branch (
   BranchID INT AUTO_INCREMENT PRIMARY KEY,
   Name VARCHAR(100) NOT NULL,
   Location VARCHAR(200) NOT NULL,
   Status VARCHAR(20) NOT NULL
);

CREATE TABLE Employee (
   EmployeeID INT AUTO_INCREMENT PRIMARY KEY,
   RoleID INT NOT NULL,
   BranchID INT NOT NULL,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50) NOT NULL,
   Email VARCHAR(100),
   Status VARCHAR(20) NOT NULL,
   FOREIGN KEY (RoleID) REFERENCES Role(RoleID) ON DELETE RESTRICT ON UPDATE
CASCADE,
   FOREIGN KEY (BranchID) REFERENCES Branch(BranchID) ON DELETE RESTRICT ON
UPDATE CASCADE
);

CREATE TABLE Customer (
   CustomerID INT AUTO_INCREMENT PRIMARY KEY,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50) NOT NULL,
   Phone VARCHAR(20),
   Email VARCHAR(100)
);

CREATE TABLE TableInfo (
   TableID INT AUTO_INCREMENT PRIMARY KEY,
   BranchID INT NOT NULL,
   Section VARCHAR(20),
   Capacity INT NOT NULL,
   Status VARCHAR(20) NOT NULL,
   FOREIGN KEY (BranchID) REFERENCES Branch(BranchID) ON DELETE RESTRICT ON
UPDATE CASCADE
);

CREATE TABLE Visit (
   VisitID INT AUTO_INCREMENT PRIMARY KEY,
   CustomerID INT NOT NULL,
   BranchID INT NOT NULL,
   TableID INT NULL,
   TimeCreated TIMESTAMP NOT NULL,
   PartySize INT NOT NULL,
   VisitType VARCHAR(20) NOT NULL,
   FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON DELETE RESTRICT ON
UPDATE CASCADE,
   FOREIGN KEY (BranchID) REFERENCES Branch(BranchID) ON DELETE RESTRICT ON
UPDATE CASCADE,
   FOREIGN KEY (TableID) REFERENCES TableInfo(TableID) ON DELETE SET NULL ON
UPDATE CASCADE
);

CREATE TABLE Waitlist (
   WaitlistID INT AUTO_INCREMENT PRIMARY KEY,
   VisitID INT UNIQUE NOT NULL,
   Quoted_wait_time INT NOT NULL,
   Status VARCHAR(20) NOT NULL,
   FOREIGN KEY (VisitID) REFERENCES Visit(VisitID) ON DELETE CASCADE ON UPDATE
CASCADE
);

CREATE TABLE Reservation (
   ReservationID INT AUTO_INCREMENT PRIMARY KEY,
   VisitID INT UNIQUE NOT NULL,
   Reserved_time DATETIME NOT NULL,
   FOREIGN KEY (VisitID) REFERENCES Visit(VisitID) ON DELETE CASCADE ON UPDATE
CASCADE
);

CREATE TABLE Orders (
   OrderID INT AUTO_INCREMENT PRIMARY KEY,
   VisitID INT NOT NULL,
   EmployeeID INT NOT NULL,
   OrderStatus VARCHAR(20) NOT NULL,
   SubTotal DECIMAL(10,2) NOT NULL,
   TaxAmount DECIMAL(10,2) NOT NULL,
   TotalDue DECIMAL(10,2) NOT NULL,
   Created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   Updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
   FOREIGN KEY (VisitID) REFERENCES Visit(VisitID) ON DELETE RESTRICT ON UPDATE
CASCADE,
   FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON DELETE RESTRICT
ON UPDATE CASCADE
);

CREATE TABLE OrderItem (
   OrderItemID INT AUTO_INCREMENT PRIMARY KEY,
   OrderID INT NOT NULL,
   Quantity INT NOT NULL,
   UnitPriceSnapshot DECIMAL(10,2) NOT NULL,
   TaxAmountSnapshot DECIMAL(10,2) NOT NULL,
   FOREIGN KEY (OrderID) REFERENCES Orders(OrderID) ON DELETE CASCADE ON UPDATE
CASCADE
);

CREATE TABLE Payment (
   PaymentID INT AUTO_INCREMENT PRIMARY KEY,
   OrderID INT NOT NULL,
   Amount DECIMAL(10,2) NOT NULL,
   Tip DECIMAL(10,2),
   PaymentMethod VARCHAR(20) NOT NULL,
   Status VARCHAR(20) NOT NULL,
   Created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (OrderID) REFERENCES Orders(OrderID) ON DELETE RESTRICT ON UPDATE
CASCADE
);

-- 3) INDEXES
CREATE INDEX idx_visit_customer  ON Visit(CustomerID);
CREATE INDEX idx_visit_branch    ON Visit(BranchID);
CREATE INDEX idx_visit_table     ON Visit(TableID);
CREATE INDEX idx_order_visit     ON Orders(VisitID);
CREATE INDEX idx_order_employee  ON Orders(EmployeeID);
CREATE INDEX idx_payment_order   ON Payment(OrderID);

-- 4) VIEWS
DROP VIEW IF EXISTS vw_active_waitlist;
CREATE VIEW vw_active_waitlist AS
SELECT
```

```
    w.WaitlistID,
    v.VisitID,
    c.FirstName,
    c.LastName,
    v.PartySize,
    w.Quoted_wait_time,
    w.Status
FROM Waitlist w
JOIN Visit v ON v.VisitID = w.VisitID
JOIN Customer c ON c.CustomerID = v.CustomerID
WHERE w.Status = 'Waiting';

DROP VIEW IF EXISTS vw_reservations;
CREATE VIEW vw_reservations AS
SELECT
    r.ReservationID,
    v.VisitID,
    CONCAT(c.FirstName, ' ', c.LastName) AS Customer,
    r.Reserved_time,
    v.PartySize
FROM Reservation r
JOIN Visit v ON v.VisitID = r.VisitID
JOIN Customer c ON c.CustomerID = v.CustomerID;

-- 5) TRIGGERS
-- Update Orders.Updated_at on UPDATE is already handled by "ON UPDATE
CURRENT_TIMESTAMP"
-- but we'll provide a BEFORE UPDATE trigger to ensure explicit behavior (optional)
DROP TRIGGER IF EXISTS trg_update_order_timestamp;
DELIMITER $$
CREATE TRIGGER trg_update_order_timestamp
BEFORE UPDATE ON Orders
FOR EACH ROW
BEGIN
    SET NEW.Updated_at = CURRENT_TIMESTAMP;
END$$
DELIMITER ;

-- 6) STORED PROCEDURES (MySQL style)
-- create_visit
DROP PROCEDURE IF EXISTS create_visit;
DELIMITER $$
CREATE PROCEDURE create_visit(
    IN p_customer INT,
    IN p_branch INT,
    IN p_table INT,
    IN p_party INT,
    IN p_type VARCHAR(20),
    OUT new_visit INT
)
BEGIN
    INSERT INTO Visit (CustomerID, BranchID, TableID, TimeCreated, PartySize, VisitType)
    VALUES (p_customer, p_branch, p_table, NOW(), p_party, p_type);
    SET new_visit = LAST_INSERT_ID();
END$$
DELIMITER ;

-- create_order (uses JSON_TABLE)
DROP PROCEDURE IF EXISTS create_order;
DELIMITER $$
CREATE PROCEDURE create_order(
    IN p_visit INT,
    IN p_employee INT,
    IN p_items JSON,
    OUT new_order INT
)
BEGIN
    -- create order
    INSERT INTO Orders (VisitID, EmployeeID, OrderStatus, SubTotal, TaxAmount, TotalDue)
    VALUES (p_visit, p_employee, 'Open', 0, 0, 0);

    SET new_order = LAST_INSERT_ID();

    -- bulk insert order items from JSON array: [{ "quantity":1, "price":12.00, "tax":1.20 }, ...]
    INSERT INTO OrderItem (OrderID, Quantity, UnitPriceSnapshot, TaxAmountSnapshot)
    SELECT
        new_order,
        jt.quantity,
        jt.price,
        jt.tax
    FROM JSON_TABLE(
        p_items,
        '$[*]' COLUMNS (
            quantity INT PATH '$.quantity',
            price DECIMAL(10,2) PATH '$.price',
            tax DECIMAL(10,2) PATH '$.tax'
        )
    ) AS jt;
```

```
END$$
DELIMITER ;

-- seat_waitlist_customer
DROP PROCEDURE IF EXISTS seat_waitlist_customer;
DELIMITER $$
CREATE PROCEDURE seat_waitlist_customer(
   IN p_waitlist INT,
   IN p_table INT
)
BEGIN
   UPDATE Visit
   SET TableID = p_table
   WHERE VisitID = (SELECT VisitID FROM Waitlist WHERE WaitlistID = p_waitlist);

   UPDATE Waitlist
   SET Status = 'Seated'
   WHERE WaitlistID = p_waitlist;
END$$
DELIMITER ;

-- process_payment
DROP PROCEDURE IF EXISTS process_payment;
DELIMITER $$
CREATE PROCEDURE process_payment(
   IN p_order INT,
   IN p_amount DECIMAL(10,2),
   IN p_tip DECIMAL(10,2),
   IN p_method VARCHAR(20),
   OUT new_payment INT
)
BEGIN
   INSERT INTO Payment (OrderID, Amount, Tip, PaymentMethod, Status)
   VALUES (p_order, p_amount, p_tip, p_method, 'Completed');

   SET new_payment = LAST_INSERT_ID();
END$$
DELIMITER ;


INSERT INTO Role (RoleName, Description)
 VALUES
         ('Manager','Oversees branch operations'),
         ('Server','Serves customers'),
         ('Chef','Prepares food');


INSERT INTO Branch (Name, Location, Status)
VALUES
         ('Downtown','123 Main St','Open'),
         ('Uptown','45 Hill Ave','Open');


INSERT INTO Employee (RoleID, BranchID, FirstName, LastName, Email, Status)
VALUES
         (1,1,'Alice','Morgan','alice@rest.com','Active'),
         (2,1,'Bob','Turner','bob@rest.com','Active'),
         (3,1,'Carlos','Diaz','carlos@rest.com','Active'),
         (2,1,'Diana','Reeves','diana@rest.com','Active'),
         (2,1,'Evan','Wills','evan@rest.com','Active'),
         (3,1,'Frank','Ng','frank@rest.com','Active'),
         (1,2,'Grace','Hill','grace@rest.com','Active'),
         (2,2,'Henry','Park','henry@rest.com','Active'),
         (3,2,'Ivy','Chen','ivy@rest.com','Active'),
         (2,2,'Jake','Foster','jake@rest.com','Active');


INSERT INTO Customer (FirstName, LastName, Phone, Email)
VALUES
         ('John','Smith','555-1001','john.smith@gmail.com'),
         ('Sophia','Lee','555-1002','sophia.lee@yahoo.com'),
         ('Michael','Brown','555-1003','mbrown@gmail.com'),
         ('Emma','Williams','555-1004','emma.will@hotmail.com'),
         ('David','Garcia','555-1005','davidg@gmail.com'),
         ('Olivia','Martinez','555-1006','olivia.m@yahoo.com'),
         ('Noah','Lopez','555-1007','noah.lopez@gmail.com'),
         ('Ava','Hernandez','555-1008','ava_h@hotmail.com'),
         ('Liam','Wilson','555-1009','liam.w@gmail.com'),
         ('Mia','Anderson','555-1010','mia.a@gmail.com'),
         ('Lucas','Thomas','555-1011','lucast@gmail.com'),
         ('Isabella','Taylor','555-1012','izzyt@gmail.com'),
         ('Ethan','Moore','555-1013','ethanmo@gmail.com'),
         ('Amelia','Jackson','555-1014','ameliaj@hotmail.com'),
         ('James','Martin','555-1015','jamesm@gmail.com'),
         ('Harper','Lee','555-1016','harperlee@yahoo.com'),
         ('Benjamin','Perez','555-1017','benp@gmail.com'),
         ('Evelyn','White','555-1018','evewhite@gmail.com'),
         ('Logan','Harris','555-1019','loganh@gmail.com'),
         ('Lily','Sanchez','555-1020','lilys@gmail.com');


INSERT INTO TableInfo (BranchID, Section, Capacity, Status)
 VALUES
         (1,'A',2,'Available'),
         (1,'A',4,'Available'),
         (1,'A',4,'Occupied'),
         (1,'B',6,'Available'),
         (1,'B',6,'Occupied'),
         (1,'C',8,'Available'),
         (2,'A',2,'Available'),
         (2,'A',4,'Available'),
         (2,'B',4,'Occupied'),
         (2,'B',6,'Available'),
         (2,'C',8,'Available'),
         (2,'C',6,'Available');


INSERT INTO Visit (CustomerID, BranchID, TableID, TimeCreated, PartySize, VisitType)
VALUES
         (1,1,NULL,NOW(),3,'Walk-in'),
         (2,1,1,NOW(),2,'Reservation'),
         (3,1,2,NOW(),4,'Walk-in'),
         (4,1,NULL,NOW(),5,'Walk-in'),
         (5,1,3,NOW(),2,'Reservation'),
         (6,1,NULL,NOW(),3,'Walk-in'),
         (7,1,4,NOW(),6,'Reservation'),
         (8,2,7,NOW(),2,'Walk-in'),
         (9,2,NULL,NOW(),4,'Walk-in'),
         (10,2,8,NOW(),2,'Reservation'),
         (11,2,NULL,NOW(),3,'Walk-in'),
         (12,2,9,NOW(),4,'Walk-in'),
         (13,1,5,NOW(),2,'Reservation'),
         (14,1,NULL,NOW(),3,'Walk-in'),
         (15,2,10,NOW(),6,'Reservation');


INSERT INTO Waitlist (VisitID, Quoted_wait_time, Status)
VALUES
         (1,15,'Waiting'),
         (4,25,'Waiting'),
         (6,20,'Waiting'),
         (9,30,'Waiting'),
         (11,10,'Waiting');


INSERT INTO Reservation (VisitID, Reserved_time)
VALUES
         (2,  DATE_ADD(NOW(), INTERVAL 30 MINUTE)),
         (5,  DATE_ADD(NOW(), INTERVAL 1 HOUR)),
         (7,  DATE_ADD(NOW(), INTERVAL 2 HOUR)),
         (10, DATE_ADD(NOW(), INTERVAL 3 HOUR)),
         (13, DATE_ADD(NOW(), INTERVAL 90 MINUTE)),
         (15, DATE_ADD(NOW(), INTERVAL 4 HOUR));


INSERT INTO Orders (VisitID, EmployeeID, OrderStatus, SubTotal, TaxAmount, TotalDue)
VALUES
         (2,2,'Closed',40.00,4.00,44.00),
         (3,2,'Open',28.00,2.80,30.80),
         (5,1,'Closed',60.00,6.00,66.00),
         (7,4,'Open',90.00,9.00,99.00),
         (8,8,'Closed',25.00,2.50,27.50),
         (10,7,'Open',42.00,4.20,46.20),
         (12,9,'Closed',33.00,3.30,36.30),
         (13,3,'Open',55.00,5.50,60.50),
         (14,2,'Open',22.00,2.20,24.20),
         (15,8,'Closed',80.00,8.00,88.00);


INSERT INTO OrderItem (OrderID, Quantity, UnitPriceSnapshot, TaxAmountSnapshot)
VALUES
         (1,2,10.00,1.00),(1,1,20.00,2.00),
         (2,1,15.00,1.50),(2,3,5.00,0.50),
         (3,2,25.00,2.50),
         (4,3,30.00,3.00),(4,1,20.00,2.00),
         (5,1,12.00,1.20),(5,2,8.00,0.80),
         (6,2,18.00,1.80),(6,1,6.00,0.60),
         (7,3,11.00,1.10),
         (8,2,13.00,1.30),(8,1,7.00,0.70),
         (9,2,19.00,1.90),(9,1,17.00,1.70),
         (10,4,12.00,1.20);



INSERT INTO Payment (OrderID, Amount, Tip, PaymentMethod, Status)
VALUES
```

(1,44.00,5.00,'Credit Card','Completed'),
(3,66.00,8.00,'Cash','Completed'),
(5,27.50,3.00,'Credit Card','Completed'),
(7,36.30,4.00,'Credit Card','Completed'),
(8,60.50,6.00,'Cash','Completed'),
(10,88.00,10.00,'Credit Card','Completed');

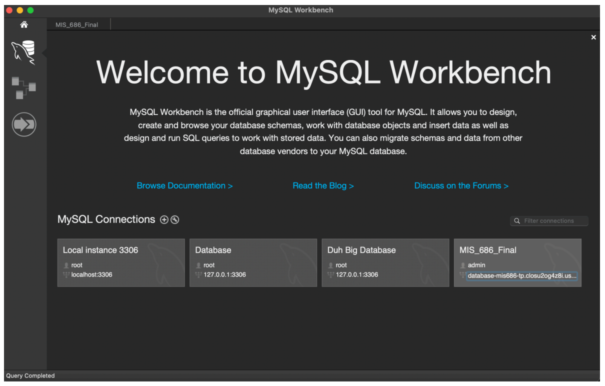
IV. Database Deployment Documentation

Database Deployment and User Access Management

We deployed our database schema using the Relational Database Service (RDS) on Amazon Web Services (AWS). During configuration, we defined the parameter group name and description and the security group, and selected MySQL Community Edition (version 8.0) as our database engine. We also modified the log_bin_trust_function_creators setting, enabling it by setting the value to 1 to allow stored function creation without required elevated privileges.

After configuring the initial parameters, we created the database instance using the same settings and designated AWS Free Tier, as the project environment does not require production-level capacity. We then specified the database name and established the master login credentials, including the username and password.

To ensure accessibility for all team members, we configured the instance to be publicly accessible and updated the associated VPC security group rules to allow appropriate inbound connections. After creating the database on Amazon's infrastructure, we connected to it using the Administrator credentials through MySQL Workbench.

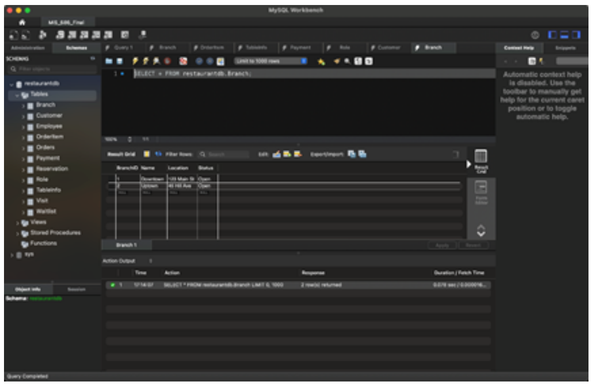


Welcome to MySQL Workbench

MySQL Workbench is the official graphical user interface (GUI) tool for MySQL. It allows you to design, create and browse your database schemas, work with database objects and insert data as well as design and run SQL queries to work with stored data. You can also migrate schemas and data from other database vendors to your MySQL database.

MySQL Connections

Local instance 3306	Database	Duh Big Database	MIS_686_Final
root	root	root	admin
localhost:3306	127.0.0.1:3306	127.0.0.1:3306	database-mis686-tp.closu2og4z8i.us...



```
SELECT * FROM restaurantdb.Branch;
```

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

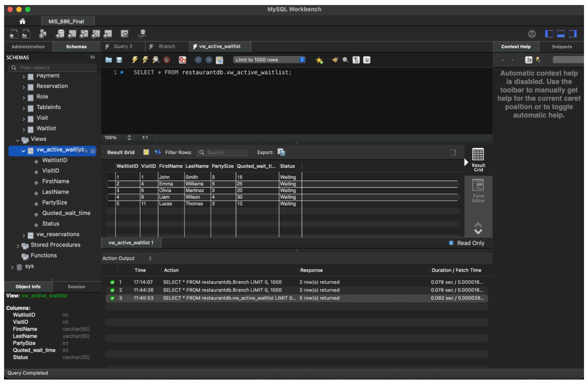
BranchID	Name	Location	Status
1	Downtown	123 Main St	Open
2	Uptown	45 Hill Ave	Open

	Time	Action	Response	Duration / Fetch Time
1	17:14:07	SELECT * FROM restaurantdb.Branch LIMIT 0, 1000	2 row(s) returned	0.078 sec / 0.000016...

Schema in MySQL Workbench when connected to AWS

Views

A view is a saved SQL query that behaves like a virtual table. In our project, we used a view to represent the active waitlist, which includes details such as the customer name, party size, and the time they were quoted.



```
SELECT * FROM restaurantdb.vw_active_waitlist;
```

WaitlistID	VisitID	FirstName	LastName	PartySize	Quoted_wait_ti...	Status
1	1	John	Smith	3	15	Waiting
2	4	Emma	Williams	5	25	Waiting
3	6	Olivia	Martinez	3	20	Waiting
4	9	Liam	Wilson	4	30	Waiting
5	11	Lucas	Thomas	3	10	Waiting

	Time	Action	Response	Duration / Fetch Time
1	17:14:07	SELECT * FROM restaurantdb.Branch LIMIT 0, 1000	2 row(s) returned	0.078 sec / 0.000016...
2	11:44:26	SELECT * FROM restaurantdb.Branch LIMIT 0, 1000	2 row(s) returned	0.078 sec / 0.000016...
3	11:45:53	SELECT * FROM restaurantdb.vw_active_waitlist LIMIT 0...	5 row(s) returned	0.082 sec / 0.000026...

View: vw_active_waitlist

Columns:

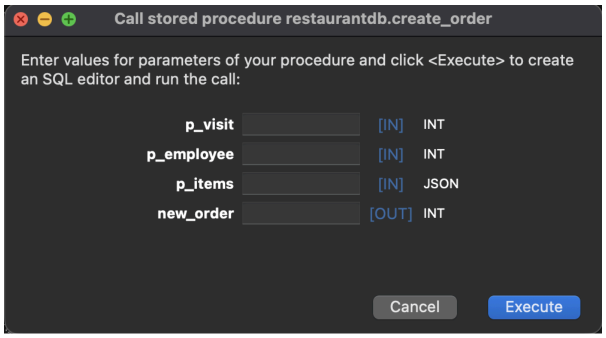
WaitlistID	int
VisitID	int
FirstName	varchar(50)
LastName	varchar(50)
PartySize	int
Quoted_wait_time	int
Status	varchar(20)

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Query Completed

Stored Procedure

A stored procedure is a saved set of SQL commands that can be run whenever needed. In our project, we used a stored procedure to handle creating an order.



Call stored procedure restaurantdb.create_order

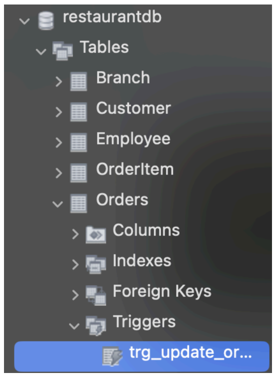
Enter values for parameters of your procedure and click <Execute> to create an SQL editor and run the call:

p_visit		[IN]	INT
p_employee		[IN]	INT
p_items		[IN]	JSON
new_order		[OUT]	INT

Trigger Update

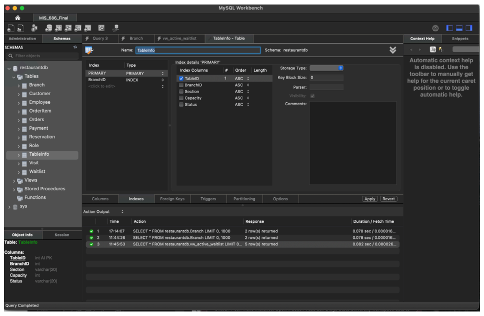
An update trigger is code that automatically runs whenever a row in a table is updated. We have a trigger update associated with orders that updates the timestamp when an order is placed.

trg_update_order_timestamp



Indexes

An index is a tool the database uses to locate data more quickly, similar to how a book index helps you find information fast. In our project, we displayed the indexes in the TableInfo table to show which ones are associated with it.



Name: TableInfo Schema: restaurantdb

Index	Type
PRIMARY	PRIMARY
BranchID	INDEX

Index details 'PRIMARY'

Index Columns	#	Order	Length
✓ TableID	1	ASC	
BranchID		ASC	
Section		ASC	
Capacity		ASC	
Status		ASC	

Key Block Size: 0

	Time	Action	Response	Duration / Fetch Time
1	17:14:07	SELECT * FROM restaurantdb.Branch LIMIT 0, 1000	2 row(s) returned	0.078 sec / 0.000016...
2	11:44:26	SELECT * FROM restaurantdb.Branch LIMIT 0, 1000	2 row(s) returned	0.078 sec / 0.000016...
3	11:45:53	SELECT * FROM restaurantdb.vw_active_waitlist LIMIT 0...	5 row(s) returned	0.082 sec / 0.000026...

Table: TableInfo

Columns:

TableID	int AI PK
BranchID	int
Section	varchar(20)
Capacity	int
Status	varchar(20)

Query Completed

V. Dashboard with Explanations for Analytical Insights

To identify operational best practices and guide future strategic decisions for our restaurant chain, we conducted a comprehensive analysis of our internal database. The goal of this analysis is to better understand customer behavior, branch performance, staffing efficiency, and overall revenue drivers so that we can implement targeted improvements across the organization.

By examining visit patterns, customer loyalty, revenue distribution, wait times, and employee performance, we can determine where to focus promotional efforts, how to optimize staffing, which branches offer the greatest growth potential, and what operational changes may increase profitability. This data-driven approach will also help us assess whether pricing adjustments are feasible and identify opportunities to enhance customer engagement through rewards or incentives.

The following SQL queries were used to extract key insights regarding branch traffic, peak and non-peak hours, customer return frequency, revenue per branch, average customer spending, wait-time performance, employee sales contribution, and staffing distribution across roles. Together, these results will support our long-term strategy for improving operational efficiency and delivering a better customer experience system-wide.

In order to determine what branches we should run a promotion on in order to gain more foot traffic, what is the total number of visits per branch?

```
SELECT
    b.Name AS Branch,
    COUNT(v.VisitID) AS TotalVisits
FROM Visit v
JOIN Branch b ON b.BranchID = v.BranchID
GROUP BY b.Name
ORDER BY TotalVisits DESC;
```

We should implement a system where we should create a discount for people who come in during non-peak hours. We need to determine what hours we should make this. List out the hours of the day and how many people visit during that hour.

```
SELECT
   EXTRACT(HOUR FROM TimeCreated) AS HourOfDay,
   COUNT(*) AS VisitCount
FROM Visit
GROUP BY HourOfDay
ORDER BY VisitCount DESC;
```

Let’s implement a reward system for our loyal customers. We need to determine which customers are returners. List out all the customers who have visited more than once.

```
SELECT
   c.CustomerID,
   c.FirstName || ' ' || c.LastName AS Customer,
   COUNT(v.VisitID) AS VisitCount
FROM Customer c
JOIN Visit v ON v.CustomerID = c.CustomerID
GROUP BY c.CustomerID
HAVING COUNT(v.VisitID) > 1
ORDER BY VisitCount DESC;
```

We need to determine how much money we make at each branch. List out each branch, how much each one made, along with tax, revenue, and tips.

```
SELECT
   b.Name AS Branch,
   SUM(o.SubTotal) AS TotalSubTotal,
   SUM(o.TaxAmount) AS TotalTaxCollected,
   SUM(o.TotalDue) AS TotalRevenue,
   SUM(p.Tip) AS TotalTips
FROM Orders o
JOIN Visit v ON v.VisitID = o.VisitID
JOIN Branch b ON b.BranchID = v.BranchID
LEFT JOIN Payment p ON p.OrderID = o.OrderID
GROUP BY b.Name
ORDER BY TotalRevenue DESC;
```

We need to see if we should increase our prices, and if our customers would be receptive to it. List out our current customers and their average spending.

```
SELECT
   c.CustomerID,
   c.FirstName || ' ' || c.LastName AS Customer,
   ROUND(AVG(o.TotalDue), 2) AS AvgSpending
FROM Customer c
JOIN Visit v ON v.CustomerID = c.CustomerID
JOIN Orders o ON o.VisitID = v.VisitID
GROUP BY c.CustomerID
ORDER BY AvgSpending DESC;
```

We may need to start adding in more tables based on how much the waittime is. Let's determine what that is across all branches.

```
SELECT
   ROUND(AVG(Quoted_wait_time), 2) AS AvgWaitMinutes
FROM Waitlist
WHERE Status = 'Waiting';
```

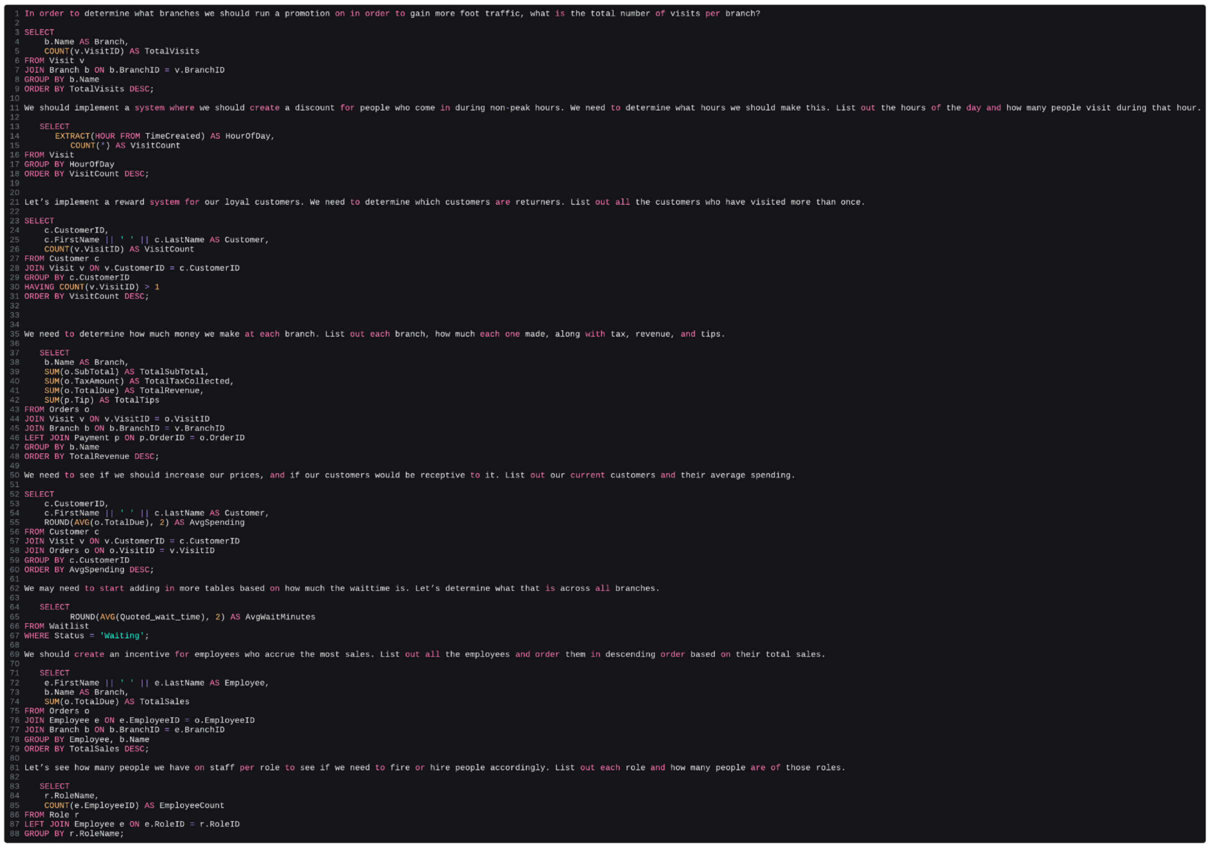
We should create an incentive for employees who accrue the most sales. List out all the employees and order them in descending order based on their total sales.

```
SELECT
   e.FirstName || ' ' || e.LastName AS Employee,
   b.Name AS Branch,
   SUM(o.TotalDue) AS TotalSales
FROM Orders o
JOIN Employee e ON e.EmployeeID = o.EmployeeID
JOIN Branch b ON b.BranchID = e.BranchID
GROUP BY Employee, b.Name
ORDER BY TotalSales DESC;
```

Let's see how many people we have on staff per role to see if we need to fire or hire people accordingly. List out each role and how many people are of those roles.

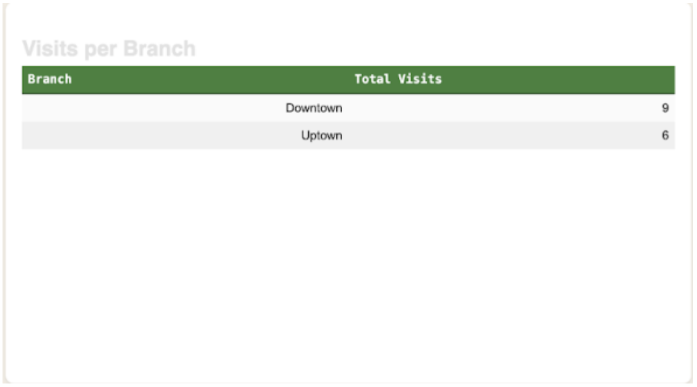
```
SELECT
   r.RoleName,
   COUNT(e.EmployeeID) AS EmployeeCount
FROM Role r
LEFT JOIN Employee e ON e.RoleID = r.RoleID
GROUP BY r.RoleName;
```

The results presented through the dashboard highlight meaningful patterns across our locations—revealing which branches attract the highest foot traffic, when customers are most likely to visit, which patrons demonstrate loyalty through repeat visits, how revenue and tips vary by branch, and how average wait times influence the customer experience. Additionally, insights into employee sales performance and role distribution help guide staffing and incentive decisions.



```
In order to determine what branches we should run a promotion on in order to gain more foot traffic, what is the total number of visits per branch?

SELECT
    b.Name AS Branch,
    COUNT(v.VisitID) AS TotalVisits
FROM Visit v
JOIN Branch b ON b.BranchID = v.BranchID
GROUP BY b.Name
ORDER BY TotalVisits DESC;

We should implement a system where we should create a discount for people who come in during non-peak hours. We need to determine what hours we should make this. List out the hours of the day and how many people visit during that hour.

   SELECT
      EXTRACT(HOUR FROM TimeCreated) AS HourOfDay,
         COUNT(*) AS VisitCount
FROM Visit
GROUP BY HourOfDay
ORDER BY VisitCount DESC;


Let’s implement a reward system for our loyal customers. We need to determine which customers are returners. List out all the customers who have visited more than once.

SELECT
    c.CustomerID,
    c.FirstName || ' ' || c.LastName AS Customer,
    COUNT(v.VisitID) AS VisitCount
FROM Customer c
JOIN Visit v ON v.CustomerID = c.CustomerID
GROUP BY c.CustomerID
HAVING COUNT(v.VisitID) > 1
ORDER BY VisitCount DESC;



We need to determine how much money we make at each branch. List out each branch, how much each one made, along with tax, revenue, and tips.

   SELECT
    b.Name AS Branch,
    SUM(o.SubTotal) AS TotalSubTotal,
    SUM(o.TaxAmount) AS TotalTaxCollected,
    SUM(o.TotalDue) AS TotalRevenue,
    SUM(p.Tip) AS TotalTips
FROM Orders o
JOIN Visit v ON v.VisitID = o.VisitID
JOIN Branch b ON b.BranchID = v.BranchID
LEFT JOIN Payment p ON p.OrderID = o.OrderID
GROUP BY b.Name
ORDER BY TotalRevenue DESC;

We need to see if we should increase our prices, and if our customers would be receptive to it. List out our current customers and their average spending.

SELECT
    c.CustomerID,
    c.FirstName || ' ' || c.LastName AS Customer,
    ROUND(AVG(o.TotalDue), 2) AS AvgSpending
FROM Customer c
JOIN Visit v ON v.CustomerID = c.CustomerID
JOIN Orders o ON o.VisitID = v.VisitID
GROUP BY c.CustomerID
ORDER BY AvgSpending DESC;

We may need to start adding in more tables based on how much the waittime is. Let’s determine what that is across all branches.

   SELECT
         ROUND(AVG(Quoted_wait_time), 2) AS AvgWaitMinutes
FROM Waitlist
WHERE Status = 'Waiting';

We should create an incentive for employees who accrue the most sales. List out all the employees and order them in descending order based on their total sales.

   SELECT
    e.FirstName || ' ' || e.LastName AS Employee,
    b.Name AS Branch,
    SUM(o.TotalDue) AS TotalSales
FROM Orders o
JOIN Employee e ON e.EmployeeID = o.EmployeeID
JOIN Branch b ON b.BranchID = e.BranchID
GROUP BY Employee, b.Name
ORDER BY TotalSales DESC;

Let’s see how many people we have on staff per role to see if we need to fire or hire people accordingly. List out each role and how many people are of those roles.

   SELECT
    r.RoleName,
    COUNT(e.EmployeeID) AS EmployeeCount
FROM Role r
LEFT JOIN Employee e ON e.RoleID = r.RoleID
GROUP BY r.RoleName;
```

1. In order to determine what branches we should run a promotion on in order to gain more foot traffic, what is the total number of visits per branch?

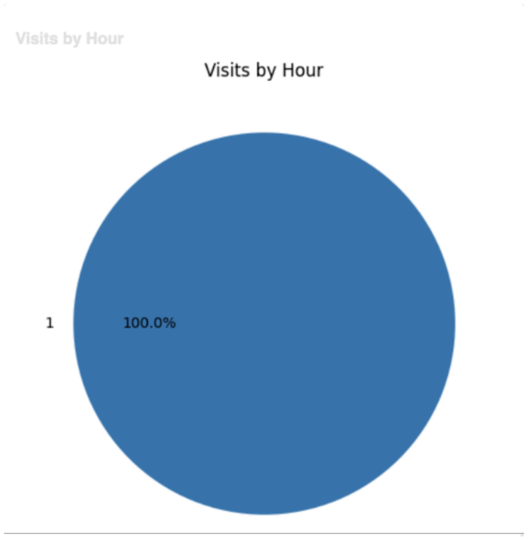


Visits per Branch

Branch	Total Visits
Downtown	9
Uptown	6

Based on the table, it can be seen that there have been 9 visits in the Downtown branch and 6 in the Uptown branch. After seeing this information clearly on a table, we are able to decide on running promotions for the Uptown branch, as it has less foot traffic.

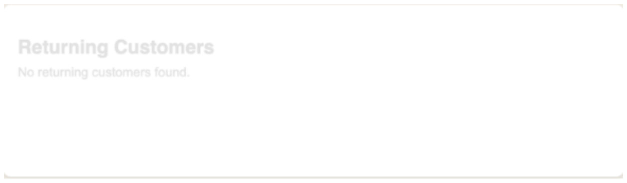
2. We should implement a system where we should create a discount for people who come in during non-peak hours. We need to determine what hours we should make this. List out the hours of the day and how many people visit during that hour.



Peak hours, based on the pie chart, are at 1pm. Once more people visit at different times, the chart will be populated with more data, and the hour with the highest number of visits becomes the peak hour.

3. Let’s implement a reward system for our loyal customers. We need to determine which customers are returners. List out all the customers who have visited more than once.

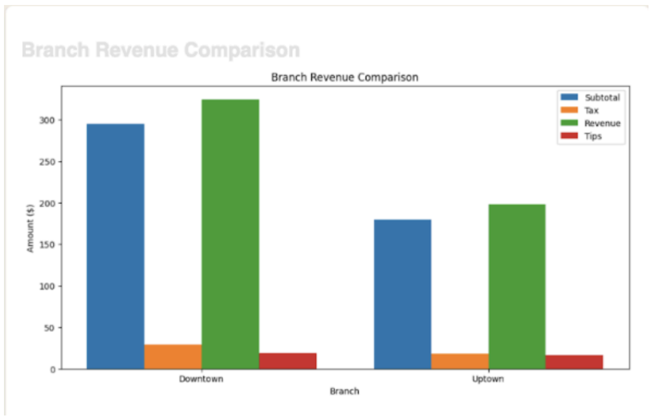
As of now there are no returning customers, as printed on the dashboard.



Returning Customers

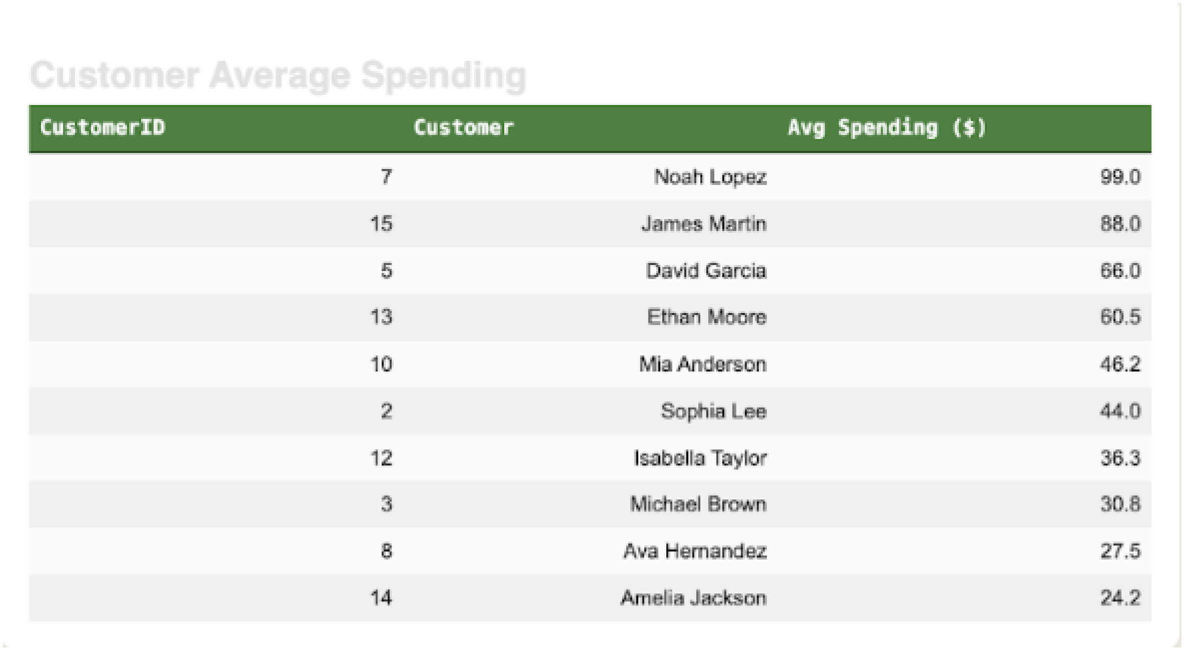
No returning customers found.

4. We need to determine how much money we make at each branch. List out each branch, how much each one made, along with tax, revenue, and tips.



Based on the chart, we can see that the revenue and Subtotal is significantly higher for the Downtown branch. For the tips and tax, both branches have similar amounts.

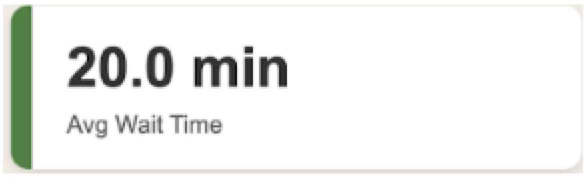
5. We need to see if we should increase our prices, and if our customers would be receptive to it. List out our current customers and their average spending.



Customer Average Spending

CustomerID	Customer	Avg Spending ($)
7	Noah Lopez	99.0
15	James Martin	88.0
5	David Garcia	66.0
13	Ethan Moore	60.5
10	Mia Anderson	46.2
2	Sophia Lee	44.0
12	Isabella Taylor	36.3
3	Michael Brown	30.8
8	Ava Hernandez	27.5
14	Amelia Jackson	24.2

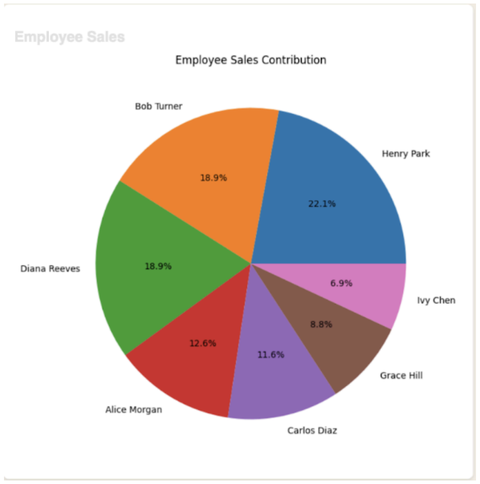
6. We may need to start adding in more tables based on how much the waittime is. Let's determine what that is across all branches.



20.0 min

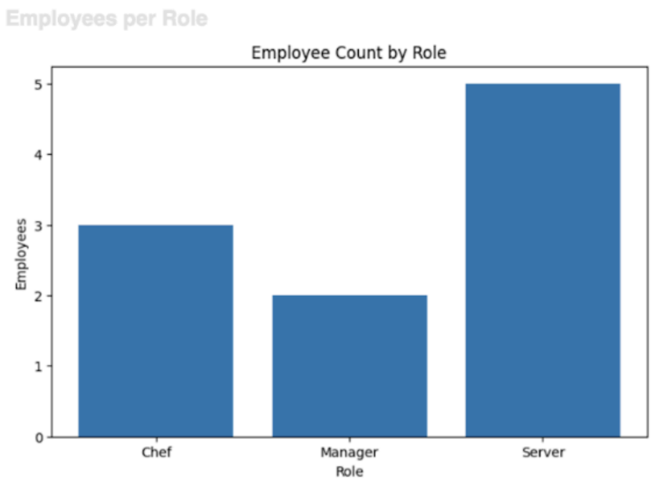
Avg Wait Time

7. We should create an incentive for employees who accrue the most sales. List out all the employees and order them in descending order based on their total sales.



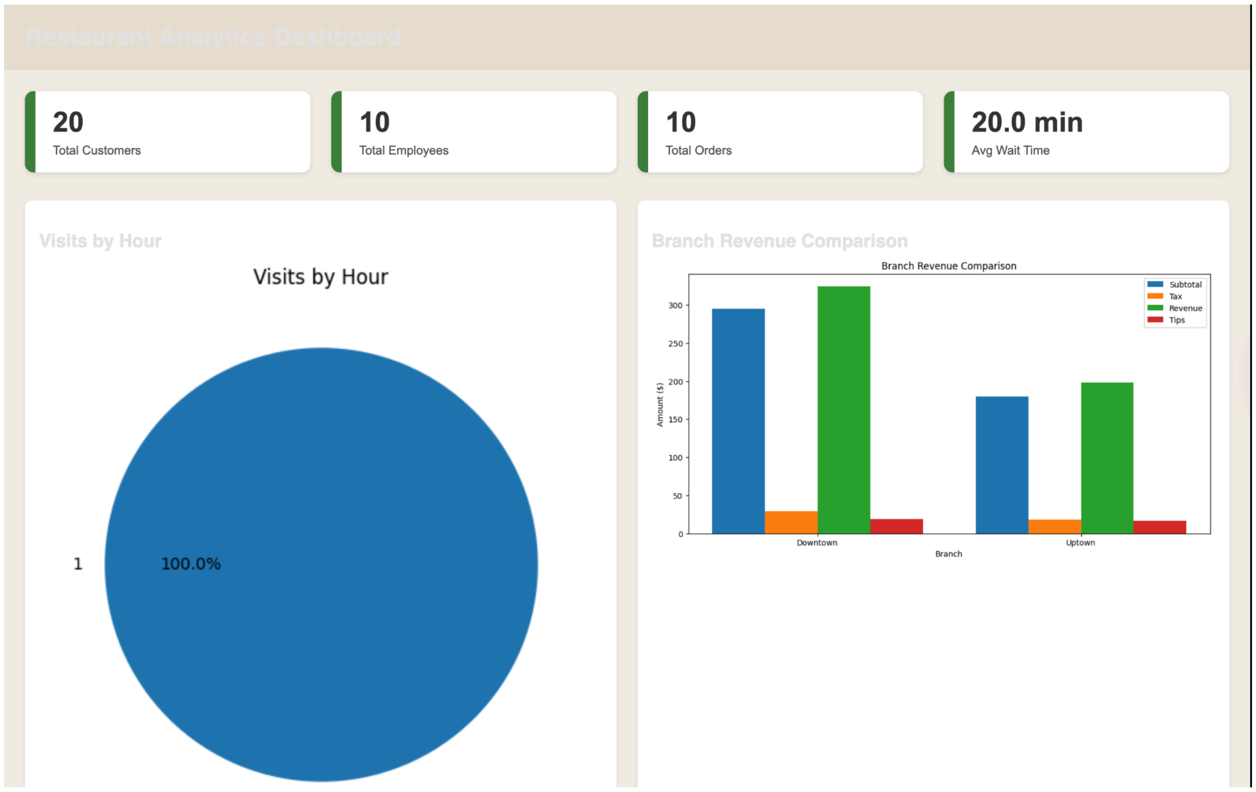
The pie chart shows that Henry Park has made the highest number of sales out of all the employees.

8. Let’s see how many people we have on staff per role to see if we need to fire or hire people accordingly. List out each role and how many people are of those roles.



From the chart we can see that there are 5 servers, 2 managers and 3 chafe working at the restaurants.

Full Dashboard:

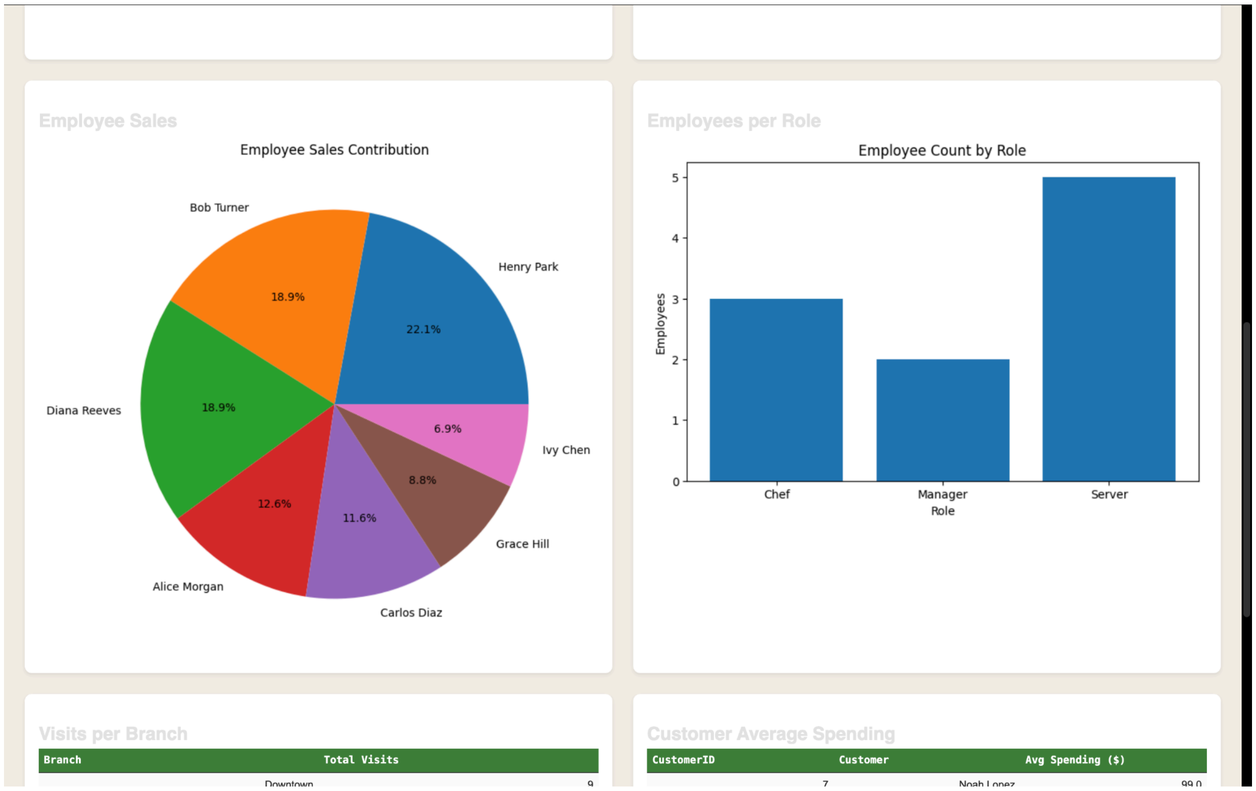


20
Total Customers

10
Total Employees

10
Total Orders

20.0 min
Avg Wait Time



Employee Sales

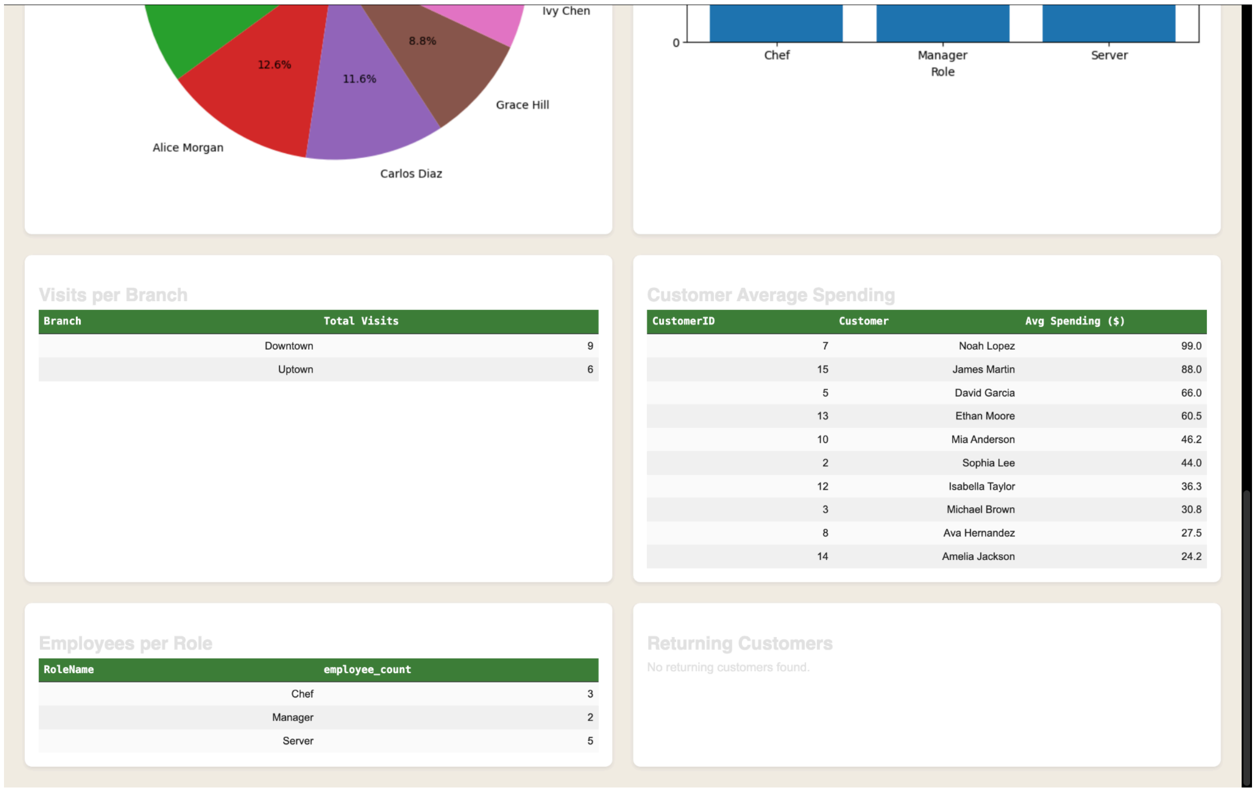
Employees per Role

Visits per Branch

Branch	Total Visits
Downtown	9

Customer Average Spending

CustomerID	Customer	Avg Spending ($)
7	Noah Lopez	99.0



Visits per Branch

Branch	Total Visits
Downtown	9
Uptown	6

Customer Average Spending

CustomerID	Customer	Avg Spending ($)
7	Noah Lopez	99.0
15	James Martin	88.0
5	David Garcia	66.0
13	Ethan Moore	60.5
10	Mia Anderson	46.2
2	Sophia Lee	44.0
12	Isabella Taylor	36.3
3	Michael Brown	30.8
8	Ava Hernandez	27.5
14	Amelia Jackson	24.2

Employees per Role

RoleName	employee_count
Chef	3
Manager	2
Server	5

Returning Customers

No returning customers found.