# A. Title Page

Lewis University  
CPSC 50900: Database Systems   
Spring 2025 Term Project

Online Food Ordering and Delivery System

Avinash Varma Saidam, [AvinashVarmaSaidam@lewisu.edu](mailto:AvinashVarmaSaidam@lewisu.edu) – Section: SP25-CPSC-50900-001

Sravanthi Domakonda, [SravanthiDomakonda@lewisu.edu](mailto:SravanthiDomakonda@lewisu.edu) – Section: SP25-CPSC-50900-002

Work products stored in the Github repository : https://github.com/Avinash462/DB\_Project.git

Table of Contents

[A. Title Page 1](#_Toc192963406)

[Schedule of Milestones 2](#_Toc192963407)

[B. Initial Proposal 2](#_Toc192963408)

[**1. What data will you be storing?** 3](#_Toc192963409)

[**2. Why are you interested in this data?** 4](#_Toc192963410)

[**3. Why is it important?** 4](#_Toc192963411)

[**4. Where will the data come from?** 4](#_Toc192963412)

[**5. Who will use this data?** 5](#_Toc192963413)

[**6. What kind of application do you plan to build with it?** 5](#_Toc192963414)

[C. Data Sources 5](#_Toc192963415)

[**Overview** 6](#_Toc192963416)

[**Data Files Overview** 6](#_Toc192963417)

[**Data File Descriptions and Structure** 6](#_Toc192963418)

[**3.1 customers.csv (Customer Data)** 6](#_Toc192963419)

[**3.2 restaurants.json (Restaurant Data)** 7](#_Toc192963420)

[**3.3 menu\_items.csv (Menu Items Data)** 7](#_Toc192963421)

[**3.4 orders.csv (Orders Data)** 8](#_Toc192963422)

[**3.5 order\_items.csv (Order Items Data)** 8](#_Toc192963423)

[**3.6 payments.xml (Payment Data)** 8](#_Toc192963424)

[**3.7 delivery\_personnel.csv (Delivery Personnel Data)** 9](#_Toc192963425)

[**3.8 reviews\_ratings.csv (Customer Feedback Data)** 9](#_Toc192963426)

[**Database Schema with Data Types and Constraints** 9](#_Toc192963427)

[**1. Customers Table** 9](#_Toc192963428)

[**2. Restaurants Table** 10](#_Toc192963429)

[**3. Menu\_Items Table** 10](#_Toc192963430)

[**4. Orders Table** 11](#_Toc192963431)

[**5. Order\_Items Table** 11](#_Toc192963432)

[**6. Payments Table** 11](#_Toc192963433)

[**7. Delivery\_Personnel Table** 12](#_Toc192963434)

[**8. Reviews\_Ratings Table** 12](#_Toc192963435)

[D. Alternative Ways to Store the Data 13](#_Toc192963436)

[**Hierarchical Database Model** 13](#_Toc192963437)

[E. Conceptual and Logical Models 16](#_Toc192963438)

[F. Physical Model 21](#_Toc192963439)

[G. Populate the database with data 26](#_Toc192963440)

[H. Data Manipulation Language (DML) Scripts 33](#_Toc192963441)

[I. Indexes 41](#_Toc192963442)

[J. Views 45](#_Toc192963443)

[K. Stored Programs (Stored Procedures, Stored Functions, Triggers) 48](#_Toc192963444)

[L. Transactions 50](#_Toc192963445)

[M. Database Security 52](#_Toc192963446)

[N. Locking and Concurrent Access 58](#_Toc192963447)

[O. Backing Up Your Database 61](#_Toc192963448)

[P. Programming 76](#_Toc192963449)

[Q. Suggested Future Work 82](#_Toc192963450)

[R. Activity Log 85](#_Toc192963451)

# Schedule of Milestones

Here is a schedule that shows when each milestone is due and what sections comprise it.

|  |  |
| --- | --- |
| Deadline | Sections for which you must demonstrate significant progress |
| February 4 at 11:59pm | a. Title page  b. Initial proposal  c. Data sources  d. Alternative ways to store the data  r. Activity Log – at least six entries covering the first two weeks |
| February 18  at 11:59pm | e. Conceptual and logical models  f. Physical model  g. Populate the database with data  r. Activity Log – at least six entries covering the past two weeks |
| March 4 at 11:59pm | h. Data manipulation language (DML) scripts  i. Indexes  j. Views  l. Transactions  m. Security  r. Activity Log – at least six entries covering the past two weeks |

The remaining sections – Triggers, Locking and Concurrency, Backup, and Programming, will be turned in with the final report, which is due March 16 at 11:59pm.

# B. Initial Proposal

*Description: You will describe the data you aim to store. What data will be storing? Why are you interested in this data? Why is it important? Where will the data come from? Who will use this data? What kind of application do you plan to build with it?*

*Rubric: Your response to each of these six questions will be graded out of 3 points.*

* *3 points: clear, complete descriptions that convey the importance and meaning of your data*
* *2 points: mostly clear descriptions, although some additional data would have helped in some sections*
* *1 point: necessary details are lacking in many of your responses.*

*You will also earn 2 additional points for coming up with a descriptive title for your project.*

*As you consider various ideas for your project, keep in mind that your database is going to have to store data for at least 8 different types of things. Each of these different “types of things” will become a table in the database you design and build. So, the idea can’t be so narrow that you can’t identify at least eight different types of things in it that you’d store data about.*

*Total points possible: 20*

ENTER YOUR INITIAL PROPOSAL HERE

The Online Food Ordering and Delivery System is a structured platform that enables customers to browse restaurant menus, place food orders, make payments, and track deliveries in real-time. Three important entities people, restaurants, and delivery drivers are linked by the system. Food items are readily chosen by consumers; restaurants can effectively handle incoming orders and change menu availability; delivery staff members receive and finish orders using a system of optimal tracking.

Stored information on users, restaurant offers, order history, payments, delivery tracking, customer comments, and promotions, the database forms the central focus of this system. The system guarantees seamless order processing, safe transactions, and effective delivery coordination via a well-organized structure, thereby raising customer happiness and corporate profitability.

**1. What data will you be storing?**

The system will keep numerous types of data to guarantee effective running conditions:

* **Customer Information**: This covers consumer names, phone numbers, delivery locations, order history, payment preferences, and saved favorite restaurants.
* **Restaurant Details**: The system will save restaurant names, branch locations, cuisine styles, menu items, food costs, business hours, ratings, customer comments.
* **Menu Items**: Every food item will feature information including name, description, ingredients, cost, and availability status.
* **Order Details**: Every order placed will be entered with specifics including the order ID, customer ID, restaurant ID, chosen food items, total cost, order status (pending, preparing, out for delivery, or completed), and timestamps.
* **Payment Transactions**: Payment records will show transaction IDs, payment methods digital wallet, credit or debit card, cash on delivery discounts applied, and payment status successful, pending, failed.
* **Delivery Personnel Information**: The database will keep driver names, phone numbers, assigned orders, vehicle information, real-time location monitoring, order completion history.
* **Order Tracking Data**: Maintaining information on order pickup time, projected delivery time, current order status, and live delivery tracking would help.
* **Customer Reviews & Ratings**: Reviews and ratings left by consumers for delivery companies and restaurants will be kept for study.
* **Promotions & Discount Offers**: The system will record exceptional discounts, loyalty programs, and promotional codes so that consumers may get tailored offers.

**2. Why are you interested in this data?**

The fast-expanding online meal delivery market requires companies to have a disciplined database to effectively handle orders, payments, and delivery. The gathered information will enable restaurants to maximize food preparation, delivery methods to increase order fulfillment, and consumer experiences to be flawless.

For example, by means of order history, restaurants can find popular meals and modify their menus. While real-time delivery tracking guarantees consumers know exactly when their meal will arrive, customer tastes and comments can direct focused marketing initiatives. Additionally used by the system will be payment records to handle refunds, stop fraud, and guarantee flawless transactions.

**3. Why is it important?**

Improving restaurant operations, enhancing customer experience, and maximizing delivery logistics depend on a reliable online food ordering system.

* **For Customers**: The system guarantees correct real-time tracking, seamless order placement, and safe transactions, so providing an interesting user experience.
* **For Restaurants**: Companies can effectively handle arriving orders, monitor inventory levels, examine sales patterns, and stop food shortages.
* **For Delivery Personnel**: Effective route navigation and order assignments will guarantee quicker delivery and help to lower errors.
* **For Business Growth**: Data insights will enable restaurants to change their menu, modify their pricing policies, and design focused marketing efforts.

Apart from that, safe payment tracking stops financial losses; automated warnings help to reduce delays and missing orders. This technology guarantees seamless cooperation among restaurants, patrons, and delivery staff generally.

**4. Where will the data come from?**

Multiple sources will gather data for the system from including:

1. **Customer Input**: Users' preferences and information will be kept in the database when they register, peruse menus, order something, and post reviews.
2. **Restaurant Updates**: Restaurants will change menus, pricing, and food availability to guarantee that patrons only view current information.
3. **Payment Gateway Integrations**: From digital wallets, credit and debit cards, and cash-on-delivery confirmations, transactions will be noted.
4. **GPS Tracking & Delivery Updates**: While GPS tracking will offer real-time order position information, delivery staff will update the state of orders.
5. **Customer Feedback & Review Submissions**: Customer ratings and reviews will be gathered and kept for study of restaurant performance.

This methodical data collecting guarantees accuracy, real-time ordering tracking, and seamless customer, restaurant, and delivery staff collaboration.

**5. Who will use this data?**

This system will be used by four primary groups:

1. **Customers**: Users will track their food deliveries in real-time, place orders, make payments, and peruse restaurant lists.
2. **Restaurants**: Staff members and proprietors of restaurants will handle incoming orders, change menu items, record income, and examine sales trends.
3. **Delivery Personnel**: Drivers will get order assignments, use real-time GPS, and change order statuses upon delivery.
4. **System Administrators**: Platform operations will be supervised by administrators, who also handle data security, customer service concerns, promotions and marketing plans.

Every organization will engage with the database in such a way that improves food delivery services' general effectiveness.

**6. What kind of application do you plan to build with it?**

The database will provide a fully functional web and mobile application meant for flawless meal ordering and delivery processes.

* **For Customers**: To search restaurants, choose meals, create special orders, and monitor delivery, the app will have a simple interface. AI-driven suggestions based on past orders will provide food selections together with integrated secure payment methods.
* **For Restaurants**: Companies will have a dashboard to handle incoming orders, change menus, handle payments, and examine customer behavior in order to enhance their offers of services.
* **For Delivery Personnel**: A mobile app will let users track deliveries on an interactive map, offer real-time order updates, and lead drivers via ideal paths.
* **For Administrators**: The backend will let managers track data, handle security, settle conflicts, and maximize the platform for improved effectiveness.

# C. Data Sources

*Description:* *Gather your data in text files. The text files may be csv, tab-delimited, xml, json, or some other custom format. Not all the files need be of the same type. Identify what each file contains by indicating where it came from, explaining in detail how it is structured, and describing how you will reorganize the data into a relational database. Post your data files to your GitHub repository, and provide samples of the data in your Word doc.*

*Rubric: Your work will be graded as follows:*

* *5 points: you gathered multiple data files that contain the data that will populate your databases. If you do not use multiple data files, you will not receive credit.*
* *5 points: you described the contents of the data files in detail, including referencing their origin and explaining how they were structured.*
* *3 points: you identify which fields you plan to include in your database, including their data types and any constraints you expect to impose on the data or steps you'll have to take to clean up the data.*
* *2 points: you post the data files to your GitHub account and make it possible for me to see them.*

*Total points possible: 15*

ENTER YOUR DATA SOURCES DESCRIPTION HERE

## **Overview**

This project creates an online food ordering and delivery system's relational database. The system will save and handle a variety of consumer, restaurant, order, menu item, payment, delivery, personnel data points. At least eight organizations in the database help to guarantee thorough coverage of the ordering and delivery process.

## **Data Files Overview**

|  |  |  |
| --- | --- | --- |
| **File Name** | **Format** | **Description** |
| **customers.csv** | CSV | Contains registered customer details such as names, contact information, and addresses. |
| **restaurants.json** | JSON | Stores restaurant details including name, location, and cuisine type. |
| **menu\_items.csv** | CSV | Includes restaurant-specific menu items, prices, and availability. |
| **orders.csv** | CSV | Tracks customer orders, including restaurant ID, items ordered, and order status. |
| **order\_items.csv** | CSV | Contains details of individual items in each order, mapping orders to menu items. |
| **payments.xml** | XML | Stores payment transactions, including method of payment and amount. |
| **delivery\_personnel.csv** | CSV | Stores details of delivery personnel such as name, assigned orders, and contact details. |
| **reviews\_ratings.csv** | CSV | Contains customer reviews and ratings for restaurants and delivery services. |

## **Data File Descriptions and Structure**

### **3.1 customers.csv (Customer Data)**

* **Source:** Simulated dataset of Indian customers.
* **Format:** CSV
* **Structure:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Customer\_ID** | **First\_Name** | **Last\_Name** | **Phone\_Number** | **Email** | **Address** | **City** | **Zip\_Code** |
| 101 | Rajesh | Sharma | 9876543210 | rajesh.sharma@gmail.com | 12, MG Road, Bengaluru | Bengaluru | 560001 |
| 102 | Priya | Iyer | 9823456789 | [priya.iyer@yahoo.com](mailto:priya.iyer@yahoo.com) | 45, Juhu Beach, Mumbai | Mumbai | 400049 |
| 103 | Akash | Mehta | 9987654321 | akash.mehta@outlook.com | 67, Salt Lake, Kolkata | Kolkata | 700091 |

* **Database Table:** **Customers**
* **Key Fields:** Customer\_ID (Primary Key), Phone\_Number (Unique)

### **3.2 restaurants.json (Restaurant Data)**

* **Source:** Simulated dataset for Indian restaurants.
* **Format:** JSON
* **Structure:**

json

CopyEdit

[

{

"restaurant\_id": 201,

"name": "Tandoori Nights",

"cuisine": "North Indian",

"location": "Connaught Place, Delhi",

"rating": 4.5

},

{

"restaurant\_id": 202,

"name": "Dosa Delights",

"cuisine": "South Indian",

"location": "Indiranagar, Bengaluru",

"rating": 4.7

}

]

* **Database Table:** **Restaurants**
* **Key Fields:** Restaurant\_ID (Primary Key), Name (Unique)

### **3.3 menu\_items.csv (Menu Items Data)**

* **Source:** Simulated restaurant menu data.
* **Format:** CSV
* **Structure:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Menu\_Item\_ID** | **Restaurant\_ID** | **Name** | **Price** | **Availability** |
| 401 | 201 | Butter Chicken | 350.00 | Available |
| 402 | 201 | Naan | 50.00 | Available |
| 403 | 202 | Masala Dosa | 120.00 | Available |

* **Database Table:** **Menu\_Items**
* **Key Fields:** Menu\_Item\_ID (Primary Key), Restaurant\_ID (Foreign Key)

### **3.4 orders.csv (Orders Data)**

* **Source:** Simulated order placement data.
* **Format:** CSV
* **Structure:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Order\_ID** | **Customer\_ID** | **Restaurant\_ID** | **Order\_Status** | **Order\_Date** |
| 301 | 101 | 201 | Delivered | 2025-01-30 19:30:00 |
| 302 | 102 | 202 | Out for Delivery | 2025-01-30 20:00:00 |

* **Database Table:** **Orders**
* **Key Fields:** Order\_ID (Primary Key), Customer\_ID (Foreign Key), Restaurant\_ID (Foreign Key)

### **3.5 order\_items.csv (Order Items Data)**

* **Source:** Simulated breakdown of food items in an order.
* **Format:** CSV
* **Structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Order\_Item\_ID** | **Order\_ID** | **Menu\_Item\_ID** | **Quantity** |
| 501 | 301 | 401 | 1 |
| 502 | 301 | 402 | 2 |

* **Database Table:** **Order\_Items**
* **Key Fields:** Order\_Item\_ID (Primary Key), Order\_ID (Foreign Key), Menu\_Item\_ID (Foreign Key)

### **3.6 payments.xml (Payment Data)**

* **Source:** Simulated payment transactions.
* **Format:** XML
* **Structure:**

xml

CopyEdit

<payments>

<payment>

<transaction\_id>601</transaction\_id>

<order\_id>301</order\_id>

<amount>550.00</amount>

<payment\_method>Credit Card</payment\_method>

<status>Completed</status>

</payment>

</payments>

* **Database Table:** **Payments**
* **Key Fields:** Transaction\_ID (Primary Key), Order\_ID (Foreign Key)

### **3.7 delivery\_personnel.csv (Delivery Personnel Data)**

* **Source:** Simulated delivery staff information.
* **Format:** CSV
* **Structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Delivery\_ID** | **Name** | **Phone\_Number** | **Vehicle\_Type** |
| 701 | Sandeep | 9876543211 | Bike |
| 702 | Rahul | 9823456712 | Scooter |

* **Database Table:** **Delivery\_Personnel**
* **Key Fields:** Delivery\_ID (Primary Key), Phone\_Number (Unique)

### **3.8 reviews\_ratings.csv (Customer Feedback Data)**

* **Source:** Simulated customer reviews and ratings.
* **Format:** CSV
* **Structure:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Review\_ID** | **Customer\_ID** | **Restaurant\_ID** | **Rating** | **Comments** |
| 801 | 101 | 201 | 5 | Excellent food! |
| 802 | 102 | 202 | 4 | Great taste! |

* **Database Table:** **Reviews\_Ratings**
* **Key Fields:** Review\_ID (Primary Key), Customer\_ID (Foreign Key), Restaurant\_ID (Foreign Key)

### **Database Schema with Data Types and Constraints**

The relational database structure for the Online Food Ordering and Distribution System is presented in this part. Every table features constraints to guarantee data integrity, main keys, foreign keys, and data types.

## **1. Customers Table**

Stores information about registered customers.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Customer\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| First\_Name | VARCHAR(50) | NOT NULL |
| Last\_Name | VARCHAR(50) | NOT NULL |
| Phone\_Number | VARCHAR(10) | UNIQUE, NOT NULL |
| Email | VARCHAR(100) | UNIQUE, NOT NULL |
| Address | VARCHAR(255) | NOT NULL |
| City | VARCHAR(50) | NOT NULL |
| Zip\_Code | VARCHAR(10) | NOT NULL, CHECK (Zip\_Code REGEXP '^[0-9]{6}$') |

* **Constraints:**
  + Customer\_ID is the **Primary Key** (Unique Identifier).
  + Phone\_Number and Email must be **unique**.
  + Zip\_Code must be a **6-digit numeric value** (Indian ZIP codes).

## **2. Restaurants Table**

Stores restaurant details.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Restaurant\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Name | VARCHAR(100) | UNIQUE, NOT NULL |
| Cuisine | VARCHAR(50) | NOT NULL |
| Location | VARCHAR(255) | NOT NULL |
| Rating | DECIMAL(2,1) | CHECK (Rating BETWEEN 1.0 AND 5.0) |

* **Constraints:**
  + Restaurant\_ID is the **Primary Key**.
  + Name must be **unique**.
  + Rating is limited to values **between 1.0 and 5.0**.

## **3. Menu\_Items Table**

Stores menu items offered by restaurants.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Menu\_Item\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Restaurant\_ID | INT | FOREIGN KEY REFERENCES Restaurants(Restaurant\_ID) ON DELETE CASCADE |
| Name | VARCHAR(100) | NOT NULL |
| Price | DECIMAL(10,2) | CHECK (Price >= 0) |
| Availability | ENUM('Available', 'Unavailable') | DEFAULT 'Available' |

* **Constraints:**
  + Menu\_Item\_ID is the **Primary Key**.
  + Restaurant\_ID is a **Foreign Key** referencing the **Restaurants** table.
  + Price must be **non-negative**.
  + Availability can be either **'Available' or 'Unavailable'**.

## **4. Orders Table**

Stores customer order details.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Order\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Customer\_ID | INT | FOREIGN KEY REFERENCES Customers(Customer\_ID) ON DELETE CASCADE |
| Restaurant\_ID | INT | FOREIGN KEY REFERENCES Restaurants(Restaurant\_ID) ON DELETE CASCADE |
| Order\_Status | ENUM('Pending', 'Preparing', 'Out for Delivery', 'Delivered', 'Cancelled') | DEFAULT 'Pending' |
| Order\_Date | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

* **Constraints:**
  + Order\_ID is the **Primary Key**.
  + Customer\_ID and Restaurant\_ID are **Foreign Keys**.
  + Order\_Status has **predefined values**.

## **5. Order\_Items Table**

Stores individual menu items for each order.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Order\_Item\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Order\_ID | INT | FOREIGN KEY REFERENCES Orders(Order\_ID) ON DELETE CASCADE |
| Menu\_Item\_ID | INT | FOREIGN KEY REFERENCES Menu\_Items(Menu\_Item\_ID) ON DELETE CASCADE |
| Quantity | INT | CHECK (Quantity > 0) |

* **Constraints:**
  + Order\_Item\_ID is the **Primary Key**.
  + Order\_ID and Menu\_Item\_ID are **Foreign Keys**.
  + Quantity must be **greater than 0**.

## **6. Payments Table**

Stores order payment transactions.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Transaction\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Order\_ID | INT | FOREIGN KEY REFERENCES Orders(Order\_ID) ON DELETE CASCADE |
| Amount | DECIMAL(10,2) | CHECK (Amount >= 0) |
| Payment\_Method | ENUM('Credit Card', 'Debit Card', 'UPI', 'Net Banking', 'Cash on Delivery') | NOT NULL |
| Status | ENUM('Completed', 'Pending', 'Failed') | DEFAULT 'Pending' |

* **Constraints:**
  + Transaction\_ID is the **Primary Key**.
  + Order\_ID is a **Foreign Key**.
  + Amount must be **non-negative**.
  + Payment\_Method and Status have **predefined values**.

## **7. Delivery\_Personnel Table**

Stores information about delivery personnel.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Delivery\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Name | VARCHAR(50) | NOT NULL |
| Phone\_Number | VARCHAR(10) | UNIQUE, NOT NULL |
| Vehicle\_Type | VARCHAR(50) | NOT NULL |

* **Constraints:**
  + Delivery\_ID is the **Primary Key**.
  + Phone\_Number must be **unique**.

## **8. Reviews\_Ratings Table**

Stores customer feedback and ratings.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Review\_ID | INT | PRIMARY KEY, AUTO\_INCREMENT |
| Customer\_ID | INT | FOREIGN KEY REFERENCES Customers(Customer\_ID) ON DELETE CASCADE |
| Restaurant\_ID | INT | FOREIGN KEY REFERENCES Restaurants(Restaurant\_ID) ON DELETE CASCADE |
| Rating | DECIMAL(2,1) | CHECK (Rating BETWEEN 1.0 AND 5.0) |
| Comments | TEXT | NULL DEFAULT NULL |

* **Constraints:**
  + Review\_ID is the **Primary Key**.
  + Customer\_ID and Restaurant\_ID are **Foreign Keys**.

# D. Alternative Ways to Store the Data

*Description: We will study alternatives to storing data in a relational database. Some of the alternatives come from several decades ago, including the hierarchical and network models. Some are newer options, such as NoSQL databases that use JSON or some other encoding. Describe in detail how to store the data using two alternatives to relational databases. Be sure to describe how you would implement the alternatives and the advantages and disadvantages of each.*

*Rubric: Your work will be graded as follows*

* *5 points for clearly describing how your data could be stored using one alternative to relational databases and what the advantages and disadvantages of that approach would be.*
* *5 points for clearly describing how your data could be stored using another alternative to relational databases and what the advantages and disadvantages of that approach would be.*

*Total points possible: 10*

ENTER YOUR ALTERNATIVE DATA STORAGE IDEAS HERE

**Hierarchical Database Model**

A **hierarchical database model** structures data in a tree-like format, where each parent node can have multiple child nodes, but each child node has only one parent. This structure is ideal when dealing with data that follows a strict hierarchy.

For a **restaurant management system**, the hierarchical model can be structured as follows:

* **Root Level (Top Node):** Restaurant
  + **First Child Level:** Menu Categories (e.g., Appetizers, Main Course, Desserts)
    - **Second Child Level:** Menu Items (e.g., Biryani, Butter Chicken, Paneer Tikka)
      * **Third Child Level:** Ingredients (e.g., Rice, Chicken, Spices)
  + **Parallel to Menu Categories:** Orders
    - **Child Level:** Customers (each customer order is linked to a specific restaurant)
      * **Child Level:** Payments (each payment is linked to an order)

This structure ensures that every restaurant has its own separate hierarchy of menu categories and orders, making it easy to retrieve data specific to a particular restaurant.

**Advantages**

* **Efficient Data Retrieval:** Since the hierarchical model follows a **predefined structure**, fetching menu items from a specific restaurant is straightforward.
* **Data Integrity:** Ensures that every **menu item belongs to a valid category**, and every **order is linked to a specific restaurant**.
* **Logical Representation:** The parent-child relationship closely resembles real-world data organization (e.g., a restaurant has categories, which have menu items).

**Disadvantages**

* **Lack of Flexibility:** If a **restaurant wants to add a special category or change menu item hierarchy**, the entire structure may need to be modified.
* **One-to-Many Relationship Limitation:** The **model does not support many-to-many relationships**, such as multiple customers ordering from multiple restaurants.
* **Complex Queries:** Searching for specific **customer orders** across multiple restaurants can be cumbersome.

**Example Query for Data Retrieval**

A query in IBM IMS or XML-based hierarchical storage can fetch all menu items under a specific restaurant:

|  |
| --- |
| <Restaurant name="The Great Punjab">  <MenuCategory name="Main Course">  <MenuItem name="Butter Chicken">  <Ingredients>  <Ingredient>Chicken</Ingredient>  <Ingredient>Tomato Puree</Ingredient>  <Ingredient>Spices</Ingredient>  </Ingredients>  </MenuItem>  </MenuCategory>  </Restaurant> |

**NoSQL Document-Based Database (MongoDB) for the Restaurant Management System**

**Implementation Approach**

Unlike relational databases, MongoDB (a document-based NoSQL database) stores data in JSON-like documents instead of structured tables. Each document contains all relevant information, reducing the need for joins.

For the restaurant management system, a MongoDB implementation would look like this:

|  |
| --- |
| {  "Restaurant\_ID": 4,  "Restaurant\_Name": "The Great Punjab",  "Menu": [  {  "Category": "Main Course",  "Items": [  {  "Item\_Name": "Butter Chicken",  "Price": 12.99,  "Ingredients": ["Chicken", "Tomato Puree", "Spices"]  }  ]  }  ],  "Orders": [  {  "Order\_ID": 102,  "Customer": {  "Customer\_ID": 5,  "Name": "Rahul Sharma"  },  "Items": ["Butter Chicken"],  "Total\_Cost": 12.99,  "Payment\_Status": "Completed"  }  ]  } |

**Advantages**

* **Scalability:** Since MongoDB supports horizontal scaling, it can efficiently handle a growing number of restaurants and orders.
* **Faster Read and Write Operations:** Retrieving a customer's order details is faster since all related data is embedded in a single document.
* **Flexible Schema:** New fields can be added without modifying the entire database, making it easy to accommodate changes in menu structures.

**Disadvantages**

* **Data Duplication:** If a menu item appears in multiple orders, its data may be stored redundantly, increasing storage costs.
* **Lack of Strong ACID Compliance:** Transactions such as payments require strict consistency, which MongoDB lacks by default.
* **Limited Querying Capabilities:** Unlike SQL joins, querying across multiple collections in MongoDB requires additional processing.

**Example Query for Data Retrieval**

A MongoDB query to fetch all menu items under the restaurant "The Great Punjab":

|  |
| --- |
| db.restaurants.find(  { "Restaurant\_Name": "The Great Punjab" },  { "Menu.Items": 1, "\_id": 0 }  ) |

This query efficiently retrieves all menu items under a specific restaurant without requiring multiple joins.

**Comparison Between Hierarchical and NoSQL Approaches**

|  |  |  |
| --- | --- | --- |
| Feature | Hierarchical Model | NoSQL (MongoDB) |
| Structure | Tree-based parent-child relationships | JSON-like document storage |
| Query Performance | Fast for hierarchical queries | Fast for direct document lookups |
| Schema Flexibility | Rigid structure | Highly flexible |
| Scalability | Limited scaling | Supports horizontal scaling |
| Complex Relationships | Not ideal for many-to-many relationships | Can store relationships but may lead to duplication |
| ACID Compliance | Strong | Weaker than relational databases |

While both Hierarchical Databases and NoSQL Document Stores offer alternatives to relational databases, each has unique strengths and limitations.

* The hierarchical model is ideal when data naturally follows a parent-child structure, such as restaurants and their menus, but struggles with complex queries.
* MongoDB (NoSQL document store) is more flexible and scalable, allowing faster data retrieval with embedded documents. However, data redundancy and weaker transactional consistency pose challenges.

For a restaurant management system, a hybrid approach could be beneficial:

* Use MongoDB for menu storage and customer interactions due to its flexibility and performance.
* Use a relational database for payments and transactional data, ensuring strong consistency and data integrity.

By leveraging the strengths of both approaches, the system can be scalable, efficient, and reliable.

# E. Conceptual and Logical Models

*Description: First, come up with a conceptual model. The conceptual model identifies the entity sets and the relationships among them. For each relationship, identify the connectivity and the participation (optional or mandatory).*

*Now that you know the entity sets, the next step is to develop the logical model by adding attributes. For each entity set, identify the attributes that describe the entity set. This may include references to other entity sets that are involved in relationships. Then, identify the functional dependencies that exist among them. For each functional dependency, identify the determinants and the fields they determine, like this:*

*determinant, or, determinants  attributes, they, determine*

*This becomes the basis for identifying your entity sets, which will become your tables when we move to the physical model in the next section. The attributes listed on the left of the arrows are candidates to become your primary key attributes. Attributes that are references to other entity sets are candidates to become the foreign keys.*

*For entity sets that have multi-attribute determinants, replace them with surrogate keys. This makes it easier to identify each entity in the set and to define foreign keys.*

*Then apply normalization to make sure that your design satisfies First, Second, and Third Normal forms. For 1st Normal Form, make sure that all attributes are indivisible. This may require adding an entity set that lists values that appear in comma-separated lists as individual entities. For 2nd Normal Form, make sure there are no partial dependencies (this won’t be a problem if all your entity sets have single-attribute determinants). Finally, make sure all your entity sets are in 3rd Normal Form. This means that you have to split transitive dependencies into separate entity sets and add relationships between the original entity set and the new ones.*

*Finally, draw the logical model as an ERD. At this point, your design will have entity sets, their relationships, and their attributes. M:N relationships are acceptable at this point, as we’ll remove them in the physical model.*

*Rubric: Your work will be graded as follows:*

* *5 points for identifying all entity sets*
* *5 points for writing each relationship between entity sets as two sentences and correctly identifying their connectivity and participation.*
* *5 points for adding attributes to entity sets and writing the functional dependencies correctly. Replace multi-attribute determinants with surrogate keys.*
* *4 points for performing the normalization steps. Make sure your design is in 3rd Normal Form.*
* *5 points for drawing the ERD for the logical model. At this point, the ERD will show entity sets, relationships, attributes, and primary identifiers. The design may include M:N relationships at this point. We’ll get rid of those in the physical model.*

*Total points possible: 24*

ENTER YOUR RELATIONAL DATABASE DESIGN DESCRIPTION HERE. INCLUDE A PICTURE OF YOUR ERD.

The conceptual model, which is a representation of the whole structure of the system, is in charge of spotting necessary entity sets and the interactions among them. It abstracts additional technical elements simultaneously while emphasizing specifying entities, linkages, connectedness, and participation constraints.

**Entity Set:**

The Online Food Ordering and Delivery System consists of the following major entity sets:

1. **Customers:** Individuals who place food orders.
2. **Restaurants:** Businesses that provide food services.
3. **Menu\_Items:** Food items available at various restaurants.
4. **Orders:** Records of customer orders placed at restaurants.
5. **Order\_Items:** The individual items included in each order.
6. **Payments:** Transactions made for orders.
7. **Delivery\_Personnel:** Employees responsible for delivering orders.
8. **Reviews\_Ratings:** Customer feedback on restaurant services.

**Relationships and Their Connectivity & Participation**

|  |  |  |  |
| --- | --- | --- | --- |
| Relationship | Connectivity | Participation | Description |
| Customers place Orders | 1:M (One-to-Many) | Mandatory on Orders, Optional on Customers | A customer can place multiple orders, but an order must be linked to exactly one customer. |
| Restaurants receive Orders | 1:M (One-to-Many) | Mandatory on Orders, Optional on Restaurants | A restaurant can receive multiple orders, but an order must belong to only one restaurant. |
| Orders contain Menu\_Items | M:N (Many-to-Many) | Mandatory on Order\_Items, Mandatory on Orders & Menu\_Items | An order consists of multiple menu items, and a menu item can appear in multiple orders. |
| Orders have Payments | 1:1 (One-to-One) | Mandatory on Payments, Optional on Orders | Each order has exactly one payment transaction, and a payment is associated with one order. |
| Orders are assigned to Delivery Personnel | 1:M (One-to-Many) | Mandatory on Orders, Optional on Delivery Personnel | A delivery person can handle multiple orders, but an order must be assigned to one delivery person. |
| Customers leave Reviews for Restaurants | M:N (Many-to-Many) | Optional on both | A customer can leave multiple reviews for different restaurants, and a restaurant can have multiple reviews from different customers. |

**Functional Dependencies**

|  |  |  |
| --- | --- | --- |
| Entity Set | Determinant(s) | Attributes Determined |
| Customers | Customer\_ID | First\_Name, Last\_Name, Phone\_Number, Email, Address, City, Zip\_Code |
|  | Phone\_Number | Customer\_ID *(Each phone number uniquely identifies a customer.)* |
| Restaurants | Restaurant\_ID | Name, Cuisine, Location, Rating |
|  | Name | Restaurant\_ID *(Each restaurant has a unique identifier.)* |
| Menu\_Items | Menu\_Item\_ID | Restaurant\_ID, Name, Price, Availability |
|  | (Restaurant\_ID, Name) | Menu\_Item\_ID *(Each menu item is unique within a restaurant.)* |
| Orders | Order\_ID | Customer\_ID, Restaurant\_ID, Order\_Status, Order\_Date |
|  | (Customer\_ID, Order\_Date) | Order\_ID *(A customer can place only one order at a given time.)* |
| Order\_Items | Order\_Item\_ID | Order\_ID, Menu\_Item\_ID, Quantity |
|  | (Order\_ID, Menu\_Item\_ID) | Order\_Item\_ID *(Each item in an order is uniquely identified.)* |
| Payments | Transaction\_ID | Order\_ID, Amount, Payment\_Method, Status |
|  | Order\_ID | Transaction\_ID *(Each order has one payment transaction.)* |
| Delivery\_Personnel | Delivery\_ID | Name, Phone\_Number, Vehicle\_Type |
|  | Phone\_Number | Delivery\_ID *(Each delivery person has a unique phone number.)* |
| Reviews\_Ratings | Review\_ID | Customer\_ID, Restaurant\_ID, Rating, Comments |
|  | (Customer\_ID, Restaurant\_ID) | Review\_ID *(A customer can leave only one review per restaurant per order.)* |

Normalization:

1st NF:

The database included repeated groups and multi-valued attributes in its unnormalized form (UNF), which caused retrieval inefficiencies and data duplication. For instance, the Customers table included phone numbers and addresses that can have several values in one field. Likewise, the Orders table includes an Items\_Ordered field to non-atomicly record several menu items for an order. Against the atomicity principle, the Delivery\_Personnel table also had several phone numbers and a list of allocated orders.  
Multi-valued attributes were broken out into distinct tables to convert the schema into 1NF and guarantee that every column included just atomic values. Each phone number connected to a single customer in the separate Customer\_Phone\_Numbers table created from splitting the Customers table. In a similar vein, Order\_Items was developed to hold unique menu items within an order rather than listing several things in one field. Likewise, Assigned\_Orders was developed to handle several orders placed to a delivery person, therefore guaranteeing organized data storage.

**Schema after 1NF:**

* Customers(Customer\_ID, First\_Name, Last\_Name, Email, Address, City, Phone\_Number)
* Restaurants(Restaurant\_ID, Name, Cuisine, Location, Rating)
* Menu\_Items(Menu\_Item\_ID, Restaurant\_ID, Name, Price, Availability)
* Orders(Order\_ID, Customer\_ID, Restaurant\_ID, Order\_Status, Order\_Date)
* Order\_Items(Order\_Item\_ID, Order\_ID, Menu\_Item\_ID, Quantity)
* Payments(Transaction\_ID, Order\_ID, Payment\_Method\_ID, Amount, Status)
* Payment\_Method(Payment\_Method\_ID, Payment\_Method)
* Delivery\_Personnel(Delivery\_ID, Name, Phone\_Number, Vehicle\_Type)
* Reviews\_Ratings(Review\_ID, Customer\_ID, Restaurant\_ID, Rating, Comments)

2nd NF:

To convert scheam into 2nd NF, we need to remove partial key dependencies existing in the schema. As we see the schema we can observe that there is no partial key dependencies, because in each table, every attribute is uniquely identified by only 1 primary key, there is no attribute that depends on the composite key. So we can say our schema is already in 2nd NF.

**Schema after 2NF:**

* Customers(Customer\_ID, First\_Name, Last\_Name, Email, Address, City, Phone\_Number)
* Restaurants(Restaurant\_ID, Name, Cuisine, Location, Rating)
* Menu\_Items(Menu\_Item\_ID, Restaurant\_ID, Name, Price, Availability)
* Orders(Order\_ID, Customer\_ID, Restaurant\_ID, Order\_Status, Order\_Date)
* Order\_Items(Order\_Item\_ID, Order\_ID, Menu\_Item\_ID, Quantity)
* Payments(Transaction\_ID, Order\_ID, Payment\_Method\_ID, Amount, Status)
* Payment\_Method(Payment\_Method\_ID, Payment\_Method)
* Delivery\_Personnel(Delivery\_ID, Name, Phone\_Number, Vehicle\_Type)
* Reviews\_Ratings(Review\_ID, Customer\_ID, Restaurant\_ID, Rating, Comments)

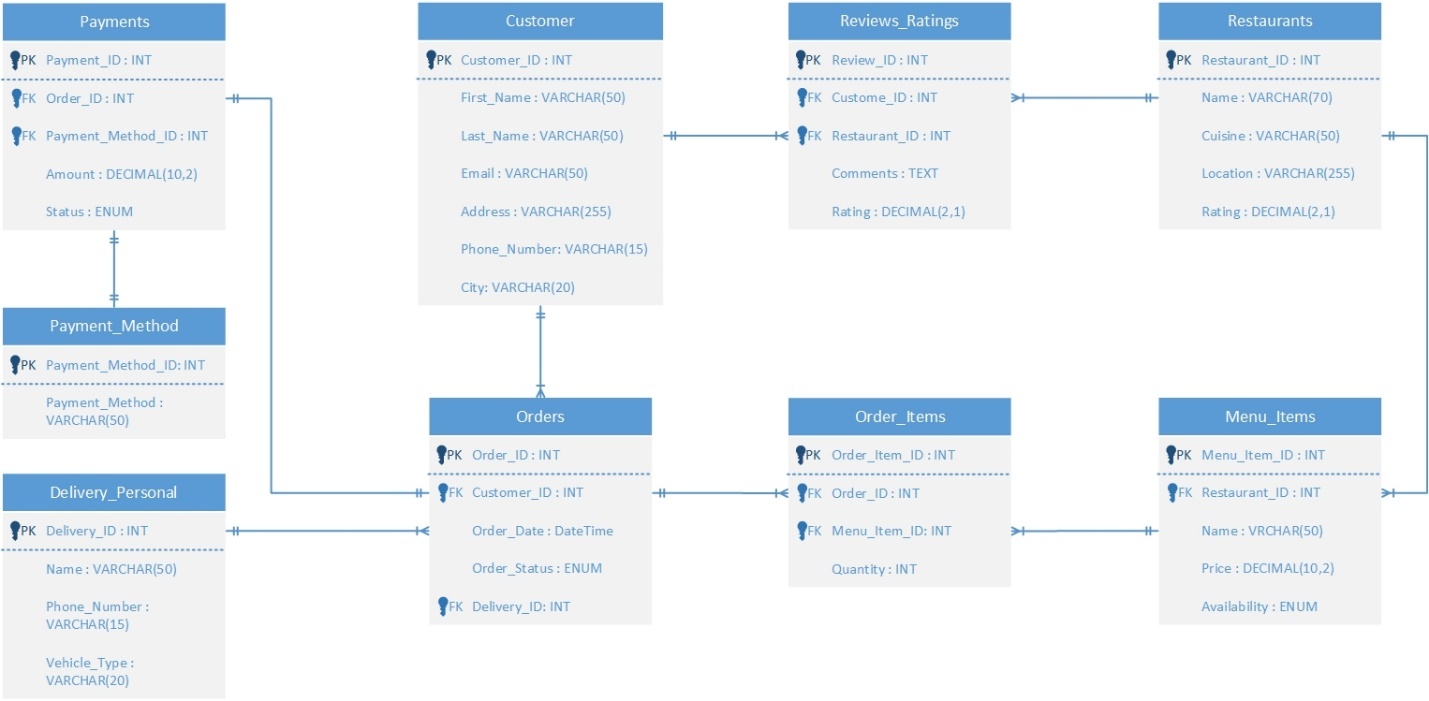
3rd NF:

Now we need to check transitive dependency, to make sure schema is in 3rd NF. And we can see no transitive dependency

**Schema after 3NF:**

* Customers(Customer\_ID, First\_Name, Last\_Name, Email, Address, City, Phone\_Number)
* Restaurants(Restaurant\_ID, Name, Cuisine, Location, Rating)
* Menu\_Items(Menu\_Item\_ID, Restaurant\_ID, Name, Price, Availability)
* Orders(Order\_ID, Customer\_ID, Restaurant\_ID, Order\_Status, Order\_Date)
* Order\_Items(Order\_Item\_ID, Order\_ID, Menu\_Item\_ID, Quantity)
* Payments(Transaction\_ID, Order\_ID, Payment\_Method\_ID, Amount, Status)
* Payment\_Method(Payment\_Method\_ID, Payment\_Method)
* Delivery\_Personnel(Delivery\_ID, Name, Phone\_Number, Vehicle\_Type)
* Reviews\_Ratings(Review\_ID, Customer\_ID, Restaurant\_ID, Rating, Comments)

ERD:



# F. Physical Model

*Description: This is where you will complete your database design. Add data types, including size constraints, uniqueness constraints, and auto-incrementing for all attributes. Implement relationships using foreign keys. Replace many-to-many relationships with two one-to-many relationships using bridge entity sets. Add additional entity sets that you think could be helpful for storing the acceptable values of particular attributes. (For example, if you were storing student data, valid student statuses might include Good Standing, Graduated, On Probation, Expelled. Put those in a table and create a relationship back to the student table). Draw the ERD for the physical model.*

*Using the final ERD, write the SQL DDL statements needed to create the database, its tables, and the relationships among them. Run these statements in MySQL to build your database. Provide screen shots that show the database you built in MySQL, including its tables and descriptions of some of the tables. To show a list of databases and a list of the tables in a particular database, use the show command. To see a description for a table, use the describe command.*

*Rubric: Your work will be graded as follows:*

* *3 points for introducing bridge entity sets (if necessary)*
* *3 points for adding data types and other constraints on the data.*
* *3 points for introducing other entity sets and their relationships that help enforce what values can be assigned to particular attributes (if necessary)*
* *5 points for drawing the ERD for the physical model. If you used Vertabelo, the resulting ERD must be free of errors and warnings*
* *6 points for generating the SQL scripts that build the database and then running the script in mysql. Demonstrate that the script built the database and its tables with screenshots that show that you ran the show and describe commands.*

*You will be penalized 4 points if your database doesn’t have at least 8 appropriately defined tables.*

*Total points possible: 20*

DESCRIBE THE STEPS YOU TOOK TO COMPLETE THE PHYSICAL MODEL. THEN SHOW THE ERD FOR THE PHYSICAL MODEL. THEN SHOW THE SQL COMMANDS THAT BUILD THE DATABASE. (FOR THIS, YOU MAY REFER TO A PARTICULAR FILE IN YOUR GITHUB REPOSITORY. MAKE SURE YOU INVITE ME AS A COLLABORATOR ON YOUR REPOSITORY SO THAT I CAN ACCESS THE SCRIPT.) FINALLY, SHOW SCREEN SHOTS THAT PROVE THAT YOU BUILT THE DATABASE AND ITS STRUCTURES IN MYSQL.

**Data Types and Constraints:**

Each column in the tables is assigned an appropriate **data type** based on:

* **Size requirements** (e.g., VARCHAR(50), VARCHAR(255))
* **Precision requirements** (e.g., DECIMAL(10,2) for currency fields)
* **Domain constraints** (e.g., ENUM for predefined categories)

**Implementation of Constraints**

1. **Primary Keys (PRIMARY KEY)**
   * Every table has a **unique identifier**.
   * Example: Customer\_ID in Customers, Order\_ID in Orders.
2. **Foreign Keys (FOREIGN KEY)**
   * Enforce **referential integrity** between related tables.
   * Example: Zip\_Code in Customers references City(Zip\_Code).
3. **Unique Constraints (UNIQUE)**
   * Applied where necessary (e.g., Email in Customers, Phone\_Number in Delivery\_Personnel).
4. **Check Constraints (CHECK)**
   * Used for **data validation**.
   * Example: CHECK (Rating BETWEEN 1.0 AND 5.0) ensures valid ratings.
5. **Auto-Incrementing Keys (AUTO\_INCREMENT)**
   * Used for all **primary keys** where values are generated automatically.

**Additional Entity Sets:**

**Payment\_Methods Table**

* Ensures **only valid payment methods** can be selected.
* Prevents inconsistent payment method entries.

**ENUM-Based Constraints**

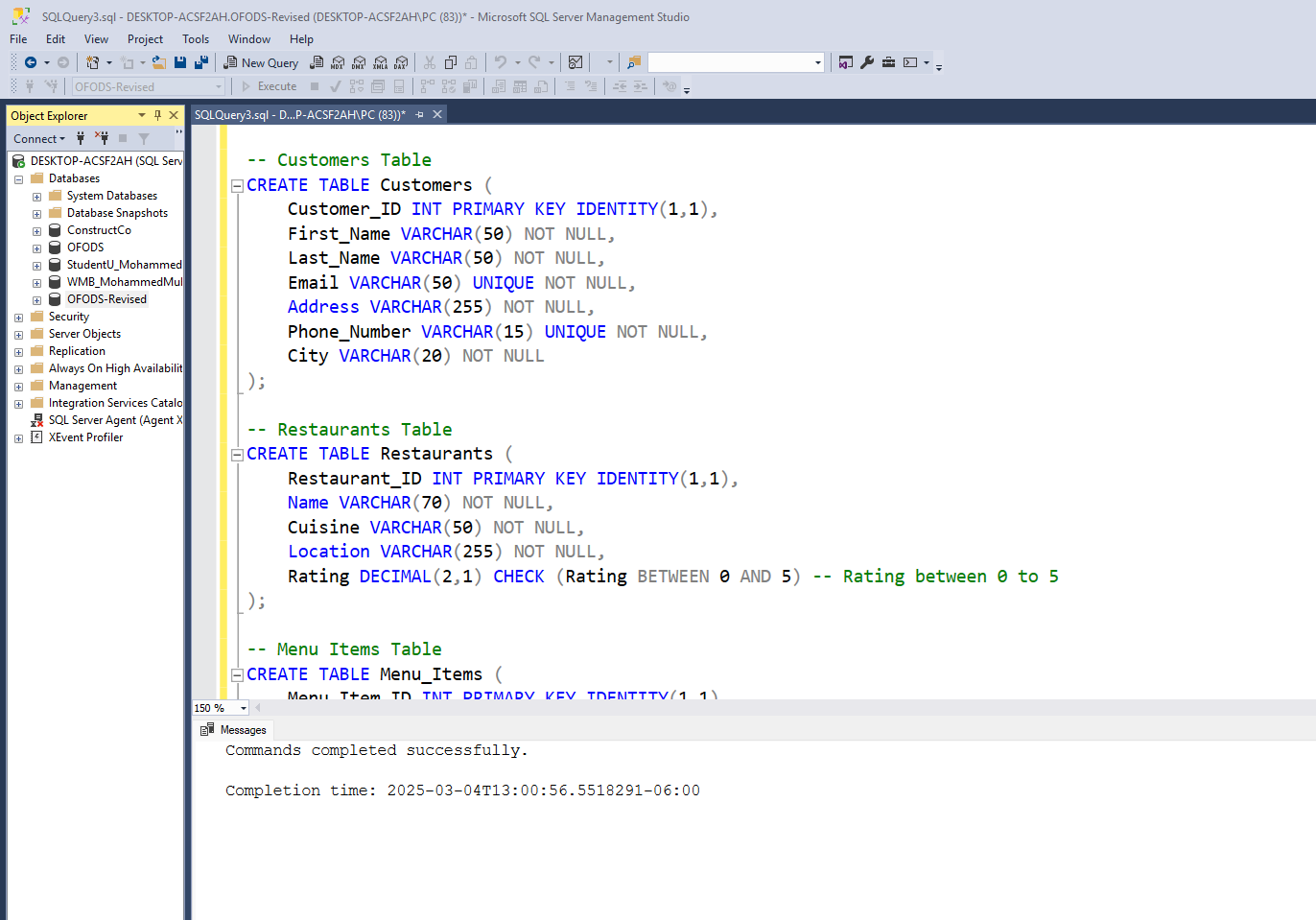
* **Order\_Status ENUM** ensures orders can only be in predefined states (Pending, Delivered, etc.).
* **Availability ENUM** in Menu\_Items ensures a menu item can only be Available or Unavailable.

A diagram of a computer

AI-generated content may be incorrect.

**SQL DDL Statements:**

|  |
| --- |
| -- Customers Table  CREATE TABLE Customers (  Customer\_ID INT PRIMARY KEY IDENTITY(1,1),  First\_Name VARCHAR(50) NOT NULL,  Last\_Name VARCHAR(50) NOT NULL,  Email VARCHAR(50) UNIQUE NOT NULL,  Address VARCHAR(255) NOT NULL,  Phone\_Number VARCHAR(15) UNIQUE NOT NULL,  City VARCHAR(20) NOT NULL  );  -- Restaurants Table  CREATE TABLE Restaurants (  Restaurant\_ID INT PRIMARY KEY IDENTITY(1,1),  Name VARCHAR(70) NOT NULL,  Cuisine VARCHAR(50) NOT NULL,  Location VARCHAR(255) NOT NULL,  Rating DECIMAL(2,1) CHECK (Rating BETWEEN 0 AND 5) -- Rating between 0 to 5  );  -- Menu Items Table  CREATE TABLE Menu\_Items (  Menu\_Item\_ID INT PRIMARY KEY IDENTITY(1,1),  Restaurant\_ID INT NOT NULL,  Name VARCHAR(50) NOT NULL,  Price DECIMAL(10,2) NOT NULL CHECK (Price >= 0), -- Price cannot be negative  Availability VARCHAR(15) NOT NULL CHECK (Availability IN ('Available', 'Not Available')),  FOREIGN KEY (Restaurant\_ID) REFERENCES Restaurants(Restaurant\_ID) ON DELETE CASCADE  );  -- Delivery Personnel Table  CREATE TABLE Delivery\_Personal (  Delivery\_ID INT PRIMARY KEY IDENTITY(1,1),  Name VARCHAR(50) NOT NULL,  Phone\_Number VARCHAR(15) UNIQUE NOT NULL,  Vehicle\_Type VARCHAR(20) NOT NULL  );  -- Orders Table  CREATE TABLE Orders (  Order\_ID INT PRIMARY KEY IDENTITY(1,1),  Customer\_ID INT NOT NULL,  Order\_Date DATETIME DEFAULT GETDATE(),  Order\_Status VARCHAR(15) NOT NULL CHECK (Order\_Status IN ('Pending', 'Completed', 'Cancelled')),  Delivery\_ID INT NULL,  FOREIGN KEY (Customer\_ID) REFERENCES Customers(Customer\_ID) ON DELETE CASCADE,  FOREIGN KEY (Delivery\_ID) REFERENCES Delivery\_Personal(Delivery\_ID) ON DELETE SET NULL  );  -- Order Items Table  CREATE TABLE Order\_Items (  Order\_Item\_ID INT PRIMARY KEY IDENTITY(1,1),  Order\_ID INT NOT NULL,  Menu\_Item\_ID INT NOT NULL,  Quantity INT NOT NULL CHECK (Quantity > 0), -- Quantity must be positive  FOREIGN KEY (Order\_ID) REFERENCES Orders(Order\_ID) ON DELETE CASCADE,  FOREIGN KEY (Menu\_Item\_ID) REFERENCES Menu\_Items(Menu\_Item\_ID) ON DELETE CASCADE  );  -- Payment Method Table  CREATE TABLE Payment\_Method (  Payment\_Method\_ID INT PRIMARY KEY IDENTITY(1,1),  Payment\_Method VARCHAR(50) NOT NULL UNIQUE  );  -- Payments Table  CREATE TABLE Payments (  Payment\_ID INT PRIMARY KEY IDENTITY(1,1),  Order\_ID INT NOT NULL,  Payment\_Method\_ID INT NOT NULL,  Amount DECIMAL(10,2) NOT NULL CHECK (Amount >= 0), -- Amount cannot be negative  Status VARCHAR(15) NOT NULL CHECK (Status IN ('Pending', 'Completed', 'Failed')),  FOREIGN KEY (Order\_ID) REFERENCES Orders(Order\_ID) ON DELETE CASCADE,  FOREIGN KEY (Payment\_Method\_ID) REFERENCES Payment\_Method(Payment\_Method\_ID) ON DELETE CASCADE  );  -- Reviews and Ratings Table  CREATE TABLE Reviews\_Ratings (  Review\_ID INT PRIMARY KEY IDENTITY(1,1),  Customer\_ID INT NOT NULL,  Restaurant\_ID INT NOT NULL,  Rating DECIMAL(2,1) CHECK (Rating BETWEEN 0 AND 5), -- Rating range 0 to 5  Comments TEXT,  FOREIGN KEY (Customer\_ID) REFERENCES Customers(Customer\_ID) ON DELETE CASCADE,  FOREIGN KEY (Restaurant\_ID) REFERENCES Restaurants(Restaurant\_ID) ON DELETE CASCADE  ); |



# G. Populate the database with data

*Description: You built the database in section F, and it now exists in mysql. Now populate it with your data. Take your original data source or sources and generate insert statements from them. Store the insert statements in a text file, and then use the mysql source command to run these insert statements to populate the various table structures. Generating the necessary insert statements may require writing Python scripts or manipulating Excel databases to convert the data from your original data sources.*

*Rubric: Your work will be grades as follows:*

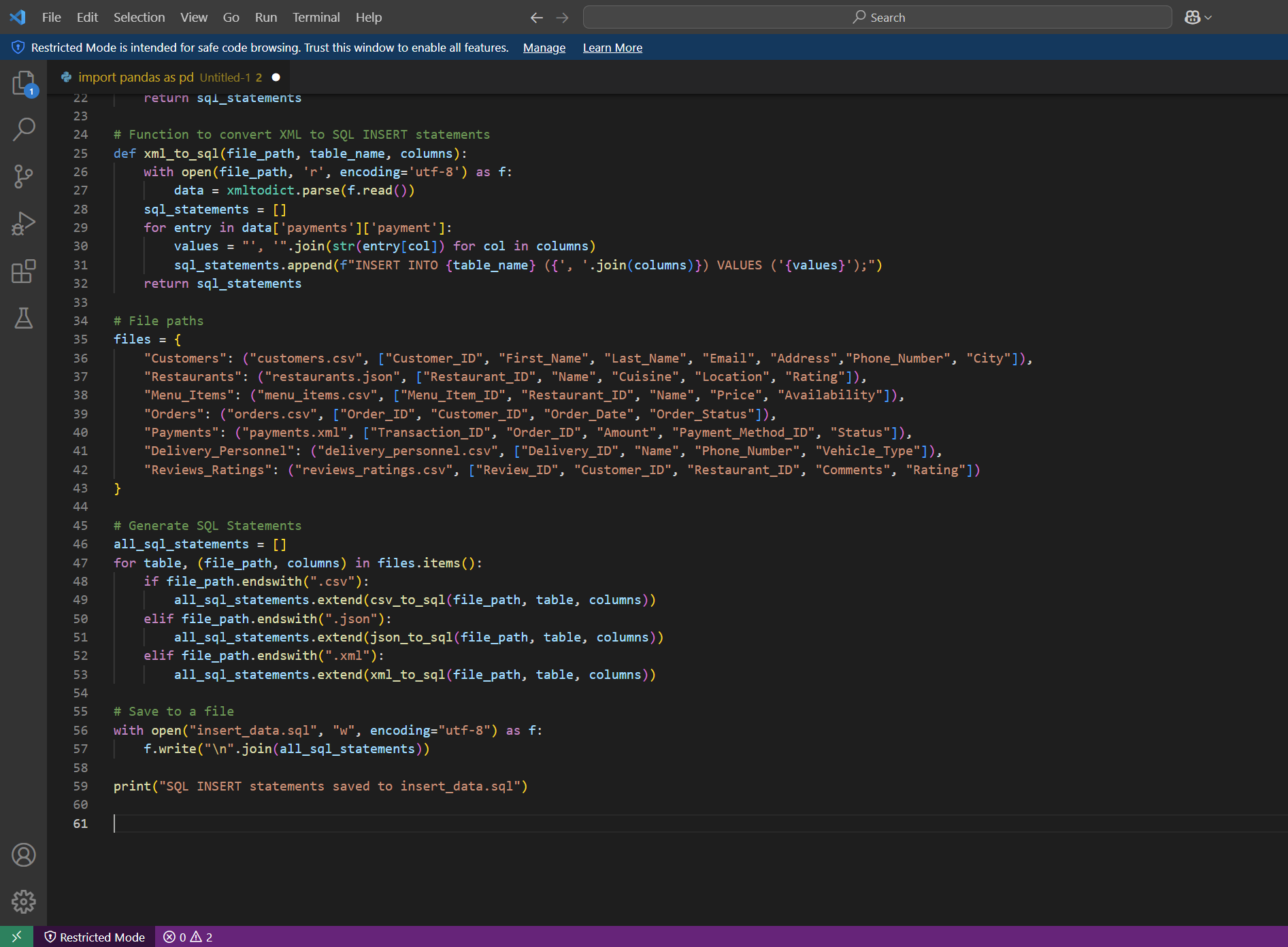
* *Explain step-by-step and very clearly how you created the required SQL statements from your initial data. Write it as a set of instructions. 5 points*
* *Show the file of insert statements that you ran in MySQL. You may do this either by including the listing in this report or by identifying the file in your GitHub that contains the insert statements. Make sure I have access to your GitHub repository. 4 points*
* *Show screenshots of the data in your MySQL database. To do this, run select statements for each table and show screen shots of what is displayed: 5 points*

*Total points possible: 14*

ENTER YOUR DDL WORK HERE

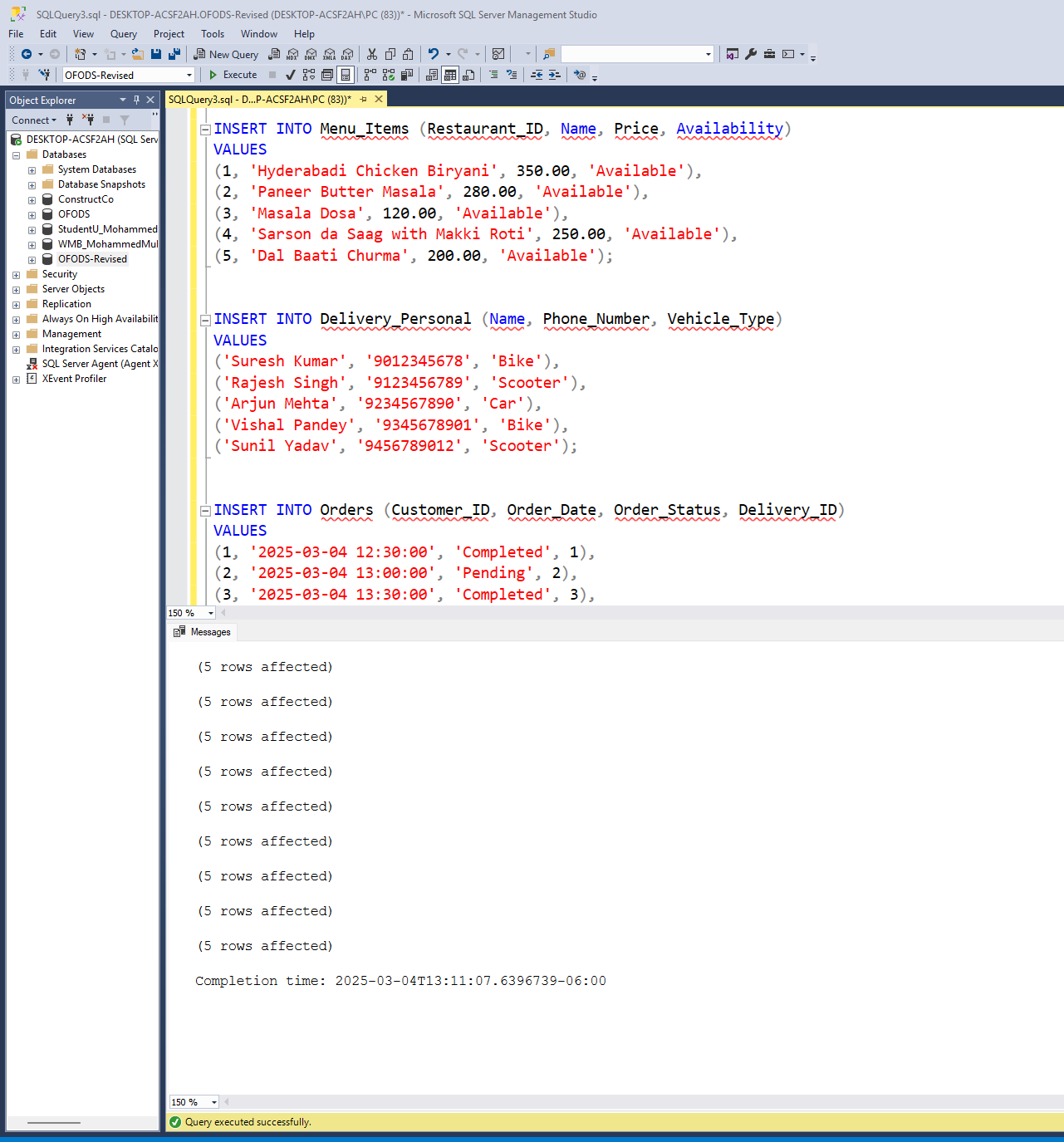
Generating insert statements from the initial data using python script.

|  |
| --- |
| import pandas as pd  import json  import xmltodict  # Function to convert CSV to SQL INSERT statements  def csv\_to\_sql(file\_path, table\_name, columns):      df = pd.read\_csv(file\_path)      sql\_statements = []      for \_, row in df.iterrows():          values = "', '".join(str(row[col]) for col in columns)          sql\_statements.append(f"INSERT INTO {table\_name} ({', '.join(columns)}) VALUES ('{values}');")      return sql\_statements  # Function to convert JSON to SQL INSERT statements  def json\_to\_sql(file\_path, table\_name, columns):      with open(file\_path, 'r', encoding='utf-8') as f:          data = json.load(f)      sql\_statements = []      for entry in data:          values = "', '".join(str(entry[col]) for col in columns)          sql\_statements.append(f"INSERT INTO {table\_name} ({', '.join(columns)}) VALUES ('{values}');")      return sql\_statements  # Function to convert XML to SQL INSERT statements  def xml\_to\_sql(file\_path, table\_name, columns):      with open(file\_path, 'r', encoding='utf-8') as f:          data = xmltodict.parse(f.read())      sql\_statements = []      for entry in data['payments']['payment']:          values = "', '".join(str(entry[col]) for col in columns)          sql\_statements.append(f"INSERT INTO {table\_name} ({', '.join(columns)}) VALUES ('{values}');")      return sql\_statements  # File paths  files = {      "Customers": ("customers.csv", ["Customer\_ID", "First\_Name", "Last\_Name", "Email", "Address","Phone\_Number", "City"]),      "Restaurants": ("restaurants.json", ["Restaurant\_ID", "Name", "Cuisine", "Location", "Rating"]),      "Menu\_Items": ("menu\_items.csv", ["Menu\_Item\_ID", "Restaurant\_ID", "Name", "Price", "Availability"]),      "Orders": ("orders.csv", ["Order\_ID", "Customer\_ID", "Order\_Date", "Order\_Status"]),      "Payments": ("payments.xml", ["Transaction\_ID", "Order\_ID", "Amount", "Payment\_Method\_ID", "Status"]),      "Delivery\_Personnel": ("delivery\_personnel.csv", ["Delivery\_ID", "Name", "Phone\_Number", "Vehicle\_Type"]),      "Reviews\_Ratings": ("reviews\_ratings.csv", ["Review\_ID", "Customer\_ID", "Restaurant\_ID", "Comments", "Rating"])  }  # Generate SQL Statements  all\_sql\_statements = []  for table, (file\_path, columns) in files.items():      if file\_path.endswith(".csv"):          all\_sql\_statements.extend(csv\_to\_sql(file\_path, table, columns))      elif file\_path.endswith(".json"):          all\_sql\_statements.extend(json\_to\_sql(file\_path, table, columns))      elif file\_path.endswith(".xml"):          all\_sql\_statements.extend(xml\_to\_sql(file\_path, table, columns))  # Save to a file  with open("insert\_data.sql", "w", encoding="utf-8") as f:      f.write("\n".join(all\_sql\_statements))  print("SQL INSERT statements saved to insert\_data.sql") |



Insert Statements

|  |
| --- |
| INSERT INTO City (Zip\_Code, City) VALUES  ('10001', 'New York'),  ('60601', 'Chicago'),  ('90001', 'Los Angeles'),  ('94101', 'San Francisco'),  ('75201', 'Dallas');  INSERT INTO Customers (First\_Name, Last\_Name, Email, Address, Zip\_Code) VALUES  ('John', 'Doe', 'john.doe@email.com', '123 Main St', '10001'),  ('Jane', 'Smith', 'jane.smith@email.com', '456 Oak St', '60601'),  ('Michael', 'Johnson', 'michael.j@email.com', '789 Pine St', '90001'),  ('Emily', 'Davis', 'emily.d@email.com', '321 Elm St', '94101'),  ('Daniel', 'Brown', 'daniel.b@email.com', '654 Maple St', '75201');  INSERT INTO Customer\_PhoneNumber (Customer\_ID, Phone\_Number) VALUES  (1, '555-1234'),  (1, '555-5678'),  (2, '555-8765'),  (3, '555-4321'),  (4, '555-6789');  INSERT INTO Restaurants (Name, Cuisine, Location, Zip\_Code, Rating) VALUES  ('Pizza Palace', 'Italian', '123 Pizza Ave', '10001', 4.5),  ('Sushi World', 'Japanese', '456 Sushi Blvd', '60601', 4.7),  ('Taco Town', 'Mexican', '789 Taco St', '90001', 4.3),  ('Burger Haven', 'American', '321 Burger Rd', '94101', 4.2),  ('Vegan Delight', 'Vegan', '654 Healthy Ln', '75201', 4.6);  INSERT INTO Menu\_Items (Restaurant\_ID, Name, Price, Availability) VALUES  (1, 'Margherita Pizza', 12.99, 'Available'),  (1, 'Pepperoni Pizza', 14.99, 'Available'),  (2, 'California Roll', 10.99, 'Available'),  (2, 'Spicy Tuna Roll', 11.99, 'Unavailable'),  (3, 'Taco Supreme', 8.99, 'Available');  INSERT INTO Orders (Customer\_ID, Order\_Date, Order\_Status, Total\_Cost) VALUES  (1, '2024-02-15 18:30:00', 'Delivered', 25.98),  (2, '2024-02-16 19:45:00', 'Preparing', 21.98),  (3, '2024-02-17 12:15:00', 'Out for Delivery', 8.99);  INSERT INTO Order\_Items (Order\_ID, Menu\_Item\_ID, Quantity) VALUES  (1, 1, 2),  (2, 3, 1),  (2, 4, 1),  (3, 5, 1);  INSERT INTO Payment\_Methods (Payment\_Method) VALUES  ('Credit Card'),  ('Debit Card'),  ('PayPal'),  ('Cash on Delivery');  INSERT INTO Payments (Order\_ID, Payment\_Method\_ID, Amount, Status) VALUES  (1, 1, 25.98, 'Completed'),  (2, 2, 21.98, 'Pending'),  (3, 4, 8.99, 'Completed');  INSERT INTO Delivery\_Personnel (Name, Phone\_Number, Vehicle\_Type) VALUES  ('James Walker', '555-2468', 'Car'),  ('Sarah Adams', '555-1357', 'Bike');  INSERT INTO Reviews\_Ratings (Customer\_ID, Restaurant\_ID, Comments, Rating) VALUES  (1, 1, 'Delicious pizza, fast delivery!', 5.0),  (2, 2, 'Great sushi, but a little expensive.', 4.5),  (3, 3, 'Tacos were fresh and tasty!', 4.8);  INSERT INTO Order\_Details (Order\_ID, Restaurant\_ID) VALUES  (1, 1), -- Order 1 is from Pizza Palace  (2, 2), -- Order 2 is from Sushi World  (3, 3); -- Order 3 is from Taco Town |



Select Statements:

|  |
| --- |
| -- Select all records from Customers  SELECT \* FROM Customers;  -- Select all records from Restaurants  SELECT \* FROM Restaurants;  -- Select all records from Menu\_Items  SELECT \* FROM Menu\_Items;  -- Select all records from Delivery\_Personal  SELECT \* FROM Delivery\_Personal;  -- Select all records from Orders  SELECT \* FROM Orders;  -- Select all records from Order\_Items  SELECT \* FROM Order\_Items;  -- Select all records from Payment\_Method  SELECT \* FROM Payment\_Method;  -- Select all records from Payments  SELECT \* FROM Payments;  -- Select all records from Reviews\_Ratings  SELECT \* FROM Reviews\_Ratings; |

A screenshot of a computer

AI-generated content may be incorrect.

# H. Data Manipulation Language (DML) Scripts

*Description: Write the SQL commands for twelve queries. Two queries should be insert statements, two should update statements, one should be a delete statement, one should be a simple select statement that selects a subset of the rows and columns from one table, two should be a select statements that select data from a joining of two tables, two should use summary functions to generate statistics about the data, one should be a multi-table query, and one should be another query of your choice. Show the queries and screenshots of the results in your Word document, and save your queries in a commented sql script to GitHub.*

*Rubric: Your work will be graded as follows:*

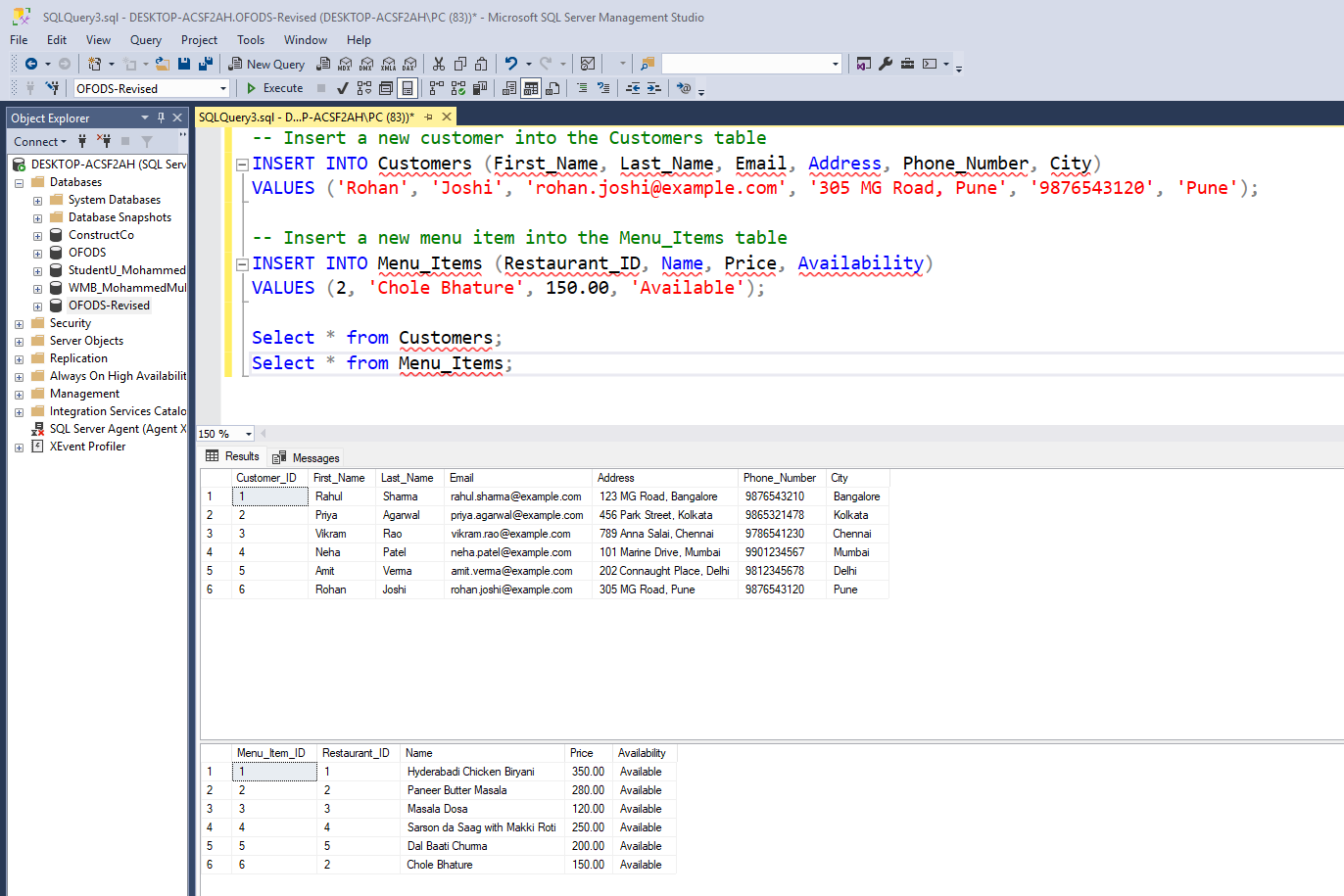
* *1 point each for the two insert statements*
* *1 point each for the two update statements*
* *1 point for the delete statement*
* *1 point for the simple select statement*
* *2 points each for the 2 join statements*
* *2 points each for the two that use summary statements*
* *2 points for the multi-table query*
* *2 points for the query of your choice.*
* *6 points for showing the query and a screenshot of the corresponding result set back-to-back for each of these queries in your Word document.*

*Total points possible: 24*

ENTER DML WORK HERE

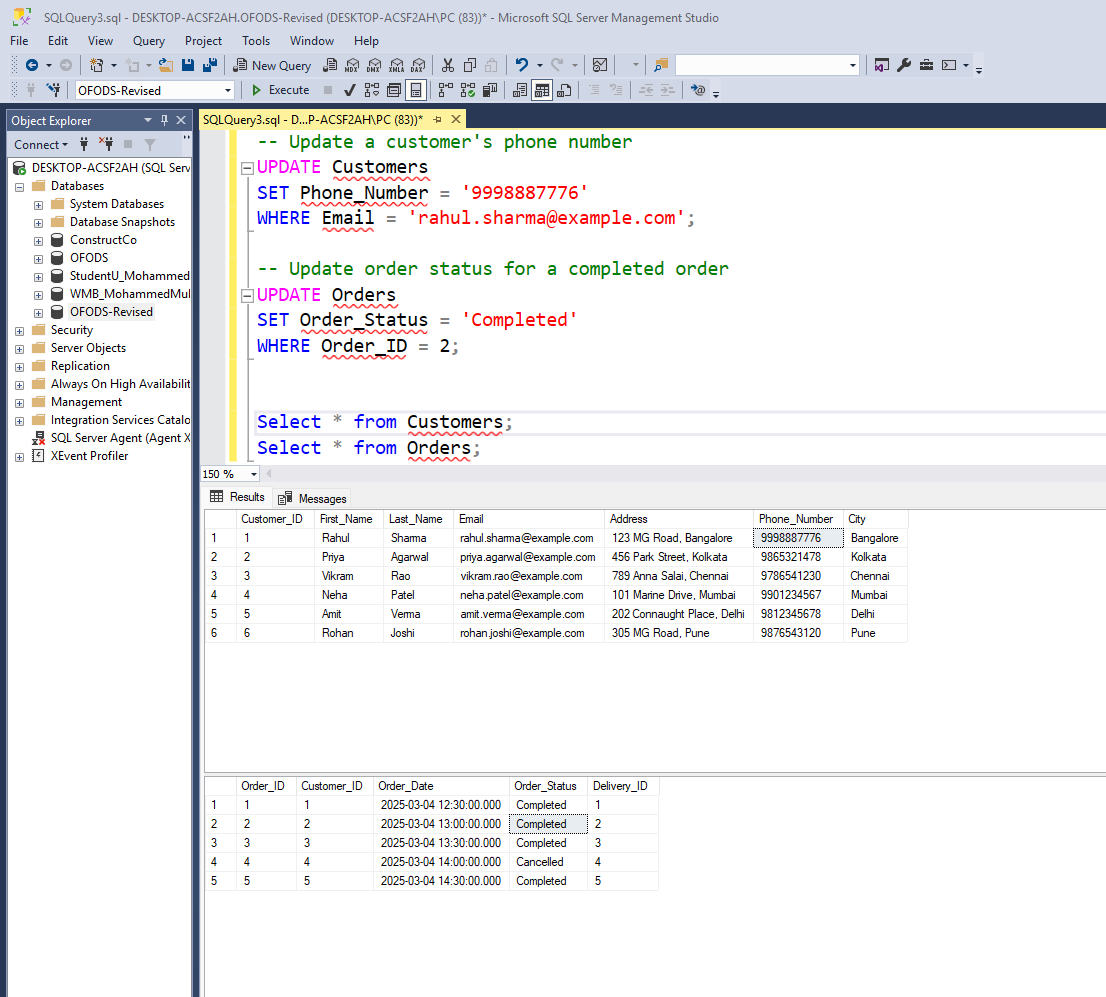
Insertion queries:

|  |
| --- |
| -- Insert a new customer into the Customers table  INSERT INTO Customers (First\_Name, Last\_Name, Email, Address, Phone\_Number, City)  VALUES ('Rohan', 'Joshi', 'rohan.joshi@example.com', '305 MG Road, Pune', '9876543120', 'Pune');  -- Insert a new menu item into the Menu\_Items table  INSERT INTO Menu\_Items (Restaurant\_ID, Name, Price, Availability)  VALUES (2, 'Chole Bhature', 150.00, 'Available'); |



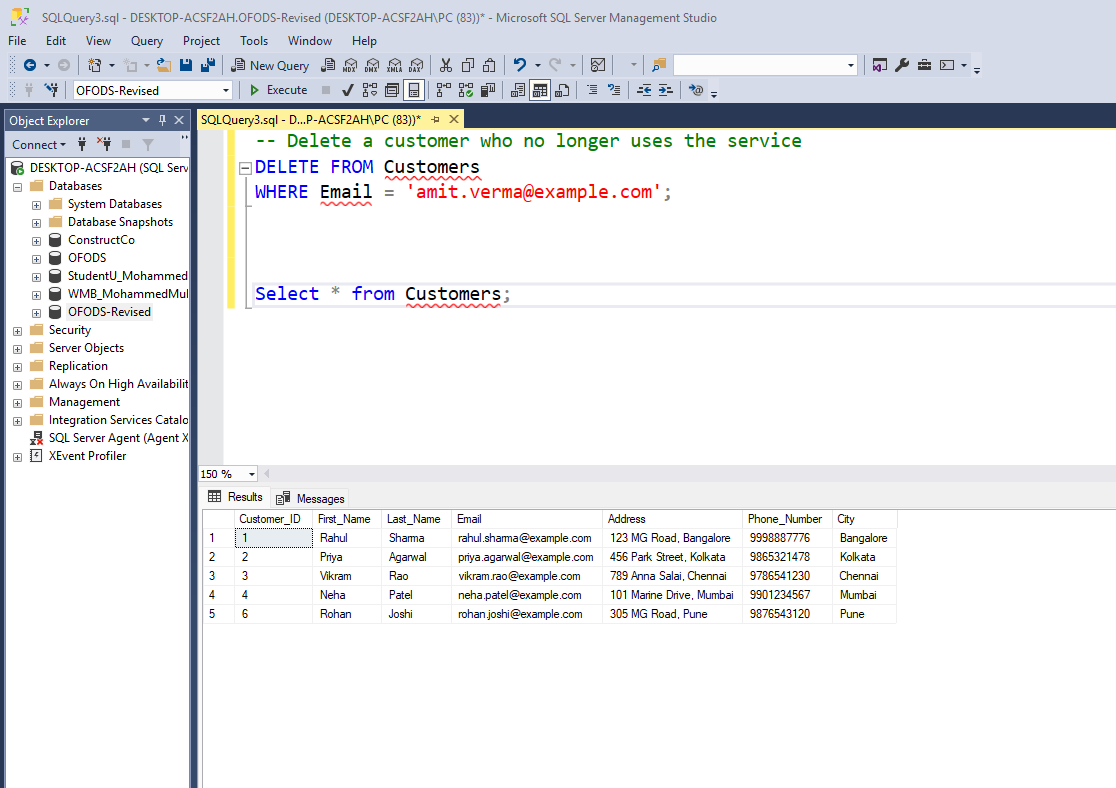
Update Queries:

|  |
| --- |
| -- Update a customer's phone number  UPDATE Customers  SET Phone\_Number = '9998887776'  WHERE Email = 'rahul.sharma@example.com';  -- Update order status for a completed order  UPDATE Orders  SET Order\_Status = 'Completed'  WHERE Order\_ID = 2; |



Delete Query:

|  |
| --- |
| -- Delete a customer who no longer uses the service  DELETE FROM Customers  WHERE Email = 'amit.verma@example.com'; |



Select Query:

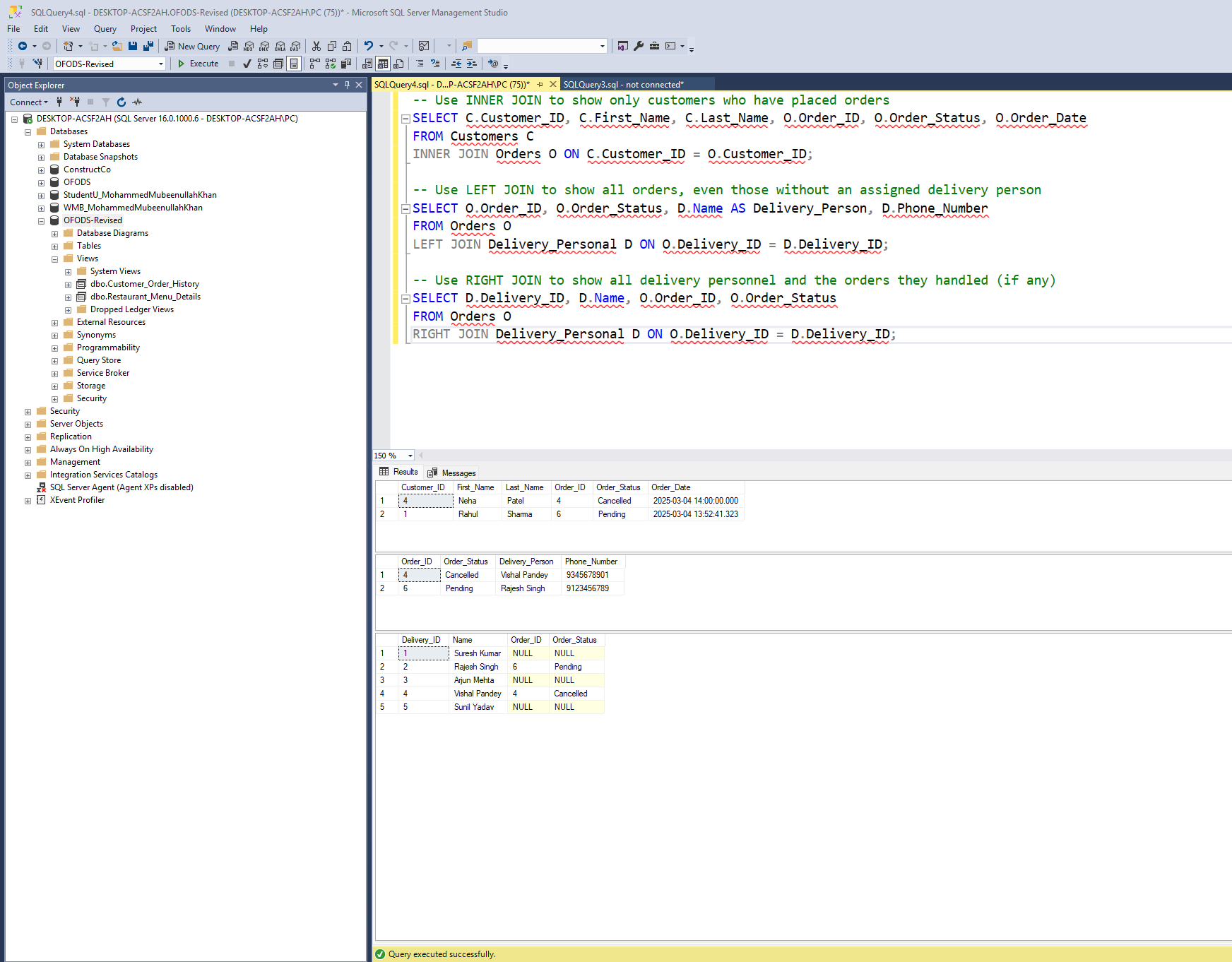
|  |
| --- |
| -- Select specific columns from the Customers table for customers in Bangalore  SELECT Customer\_ID, First\_Name, Last\_Name, Phone\_Number  FROM Customers  WHERE City = 'Bangalore'; |

A screenshot of a computer

AI-generated content may be incorrect.

Join Queries:

|  |
| --- |
| -- Use **INNER JOIN** to show only customers who have placed orders  SELECT C.Customer\_ID, C.First\_Name, C.Last\_Name, O.Order\_ID, O.Order\_Status, O.Order\_Date  FROM Customers C  INNER JOIN Orders O ON C.Customer\_ID = O.Customer\_ID;  -- Use **LEFT JOIN** to show all orders, even those without an assigned delivery person  SELECT O.Order\_ID, O.Order\_Status, D.Name AS Delivery\_Person, D.Phone\_Number  FROM Orders O  LEFT JOIN Delivery\_Personal D ON O.Delivery\_ID = D.Delivery\_ID;  -- Use **RIGHT JOIN** to show all delivery personnel and the orders they handled (if any)  SELECT D.Delivery\_ID, D.Name, O.Order\_ID, O.Order\_Status  FROM Orders O  RIGHT JOIN Delivery\_Personal D ON O.Delivery\_ID = D.Delivery\_ID; |



Summary Functions:

|  |
| --- |
| -- Get the total number of orders placed  SELECT COUNT(Order\_ID) AS Total\_Orders FROM Orders;  -- Get the average order amount from the Payments table  SELECT AVG(Amount) AS Average\_Order\_Amount FROM Payments; |

A screenshot of a computer

AI-generated content may be incorrect.

Multi table Query:

|  |
| --- |
| -- Show customer details, their orders, and the ordered menu items with quantity  SELECT C.First\_Name, C.Last\_Name, O.Order\_ID, M.Name AS Menu\_Item, OI.Quantity  FROM Customers C  JOIN Orders O ON C.Customer\_ID = O.Customer\_ID  JOIN Order\_Items OI ON O.Order\_ID = OI.Order\_ID  JOIN Menu\_Items M ON OI.Menu\_Item\_ID = M.Menu\_Item\_ID; |

A screenshot of a computer

AI-generated content may be incorrect.

Queries of our choice:

|  |
| --- |
| -- Find the top-rated restaurants with an average rating greater than 4.5  SELECT R.Name AS Restaurant\_Name, AVG(RR.Rating) AS Avg\_Rating  FROM Restaurants R  JOIN Reviews\_Ratings RR ON R.Restaurant\_ID = RR.Restaurant\_ID  GROUP BY R.Name  HAVING AVG(RR.Rating) > 4.5; |

A screenshot of a computer

AI-generated content may be incorrect.

# I. Indexes

*Description: Improve the performance of your design by adding indexes to various tables. Show the SQL needed to add the indexes. Explain why you chose the ones you added. Explain how you would demonstrate the impact the indexes had on the performance of various queries.*

*Rubric: Your work will be graded as follows:*

* *3 points for clearly defining at least three indexes and explaining why you chose them.*
* *3 points for showing the sql needed to generate the indexes*
* *2 points for explaining how you would demonstrate the performance improvement afforded by the indexes.*

*Total points possible: 8*

ENTER YOUR INDEX WORK HERE

Chosen Indexes:

**Index on Customers (Email)**

Reason:

* Email is frequently used in queries for searching customers (WHERE Email = '...').
* Since it’s already **unique**, indexing will make lookups much faster.

**Index on Orders (Customer\_ID, Order\_Date)**

Reason:

* Customer\_ID is a foreign key frequently used in joins with Customers.
* Order\_Date is often used in date-based searches (ORDER BY Order\_Date DESC).
* A composite index on (Customer\_ID, Order\_Date) speeds up retrieval of customer orders.

**Index on Menu\_Items (Restaurant\_ID, Name)**

Reason:

* Restaurant\_ID is a foreign key used in joins (JOIN Restaurants ON ...).
* Name is often searched for menu item lookups (WHERE Name LIKE '...').
* A composite index improves searches by restaurant and item name.

SQL Indexing Statements:

|  |
| --- |
| -- Index on Customers for faster email lookups  CREATE INDEX idx\_Customers\_Email  ON Customers (Email);  -- Composite Index on Orders for searching by Customer and sorting by Order Date  CREATE INDEX idx\_Orders\_Customer\_Date  ON Orders (Customer\_ID, Order\_Date);  -- Composite Index on Menu\_Items for filtering by Restaurant and searching by Name  CREATE INDEX idx\_MenuItems\_Restaurant\_Name  ON Menu\_Items (Restaurant\_ID, Name); |

A screenshot of a computer

AI-generated content may be incorrect.

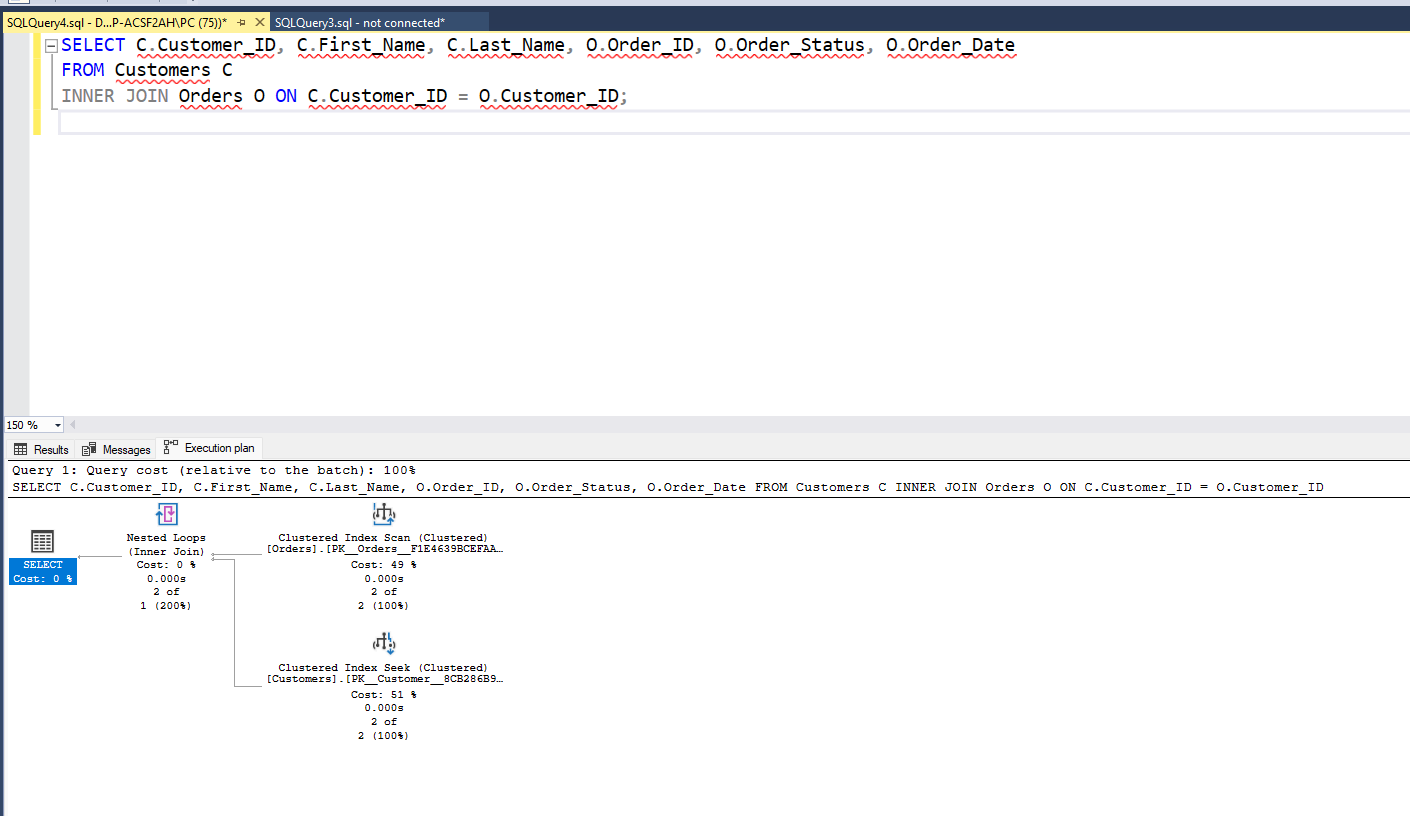
Query Performance Analysis Using Execution Plan:

The execution plan shows that SQL Server used a Nested Loop Join for this query.

The Orders table used a Clustered Index Scan, while the Customers table used a Clustered Index Seek.

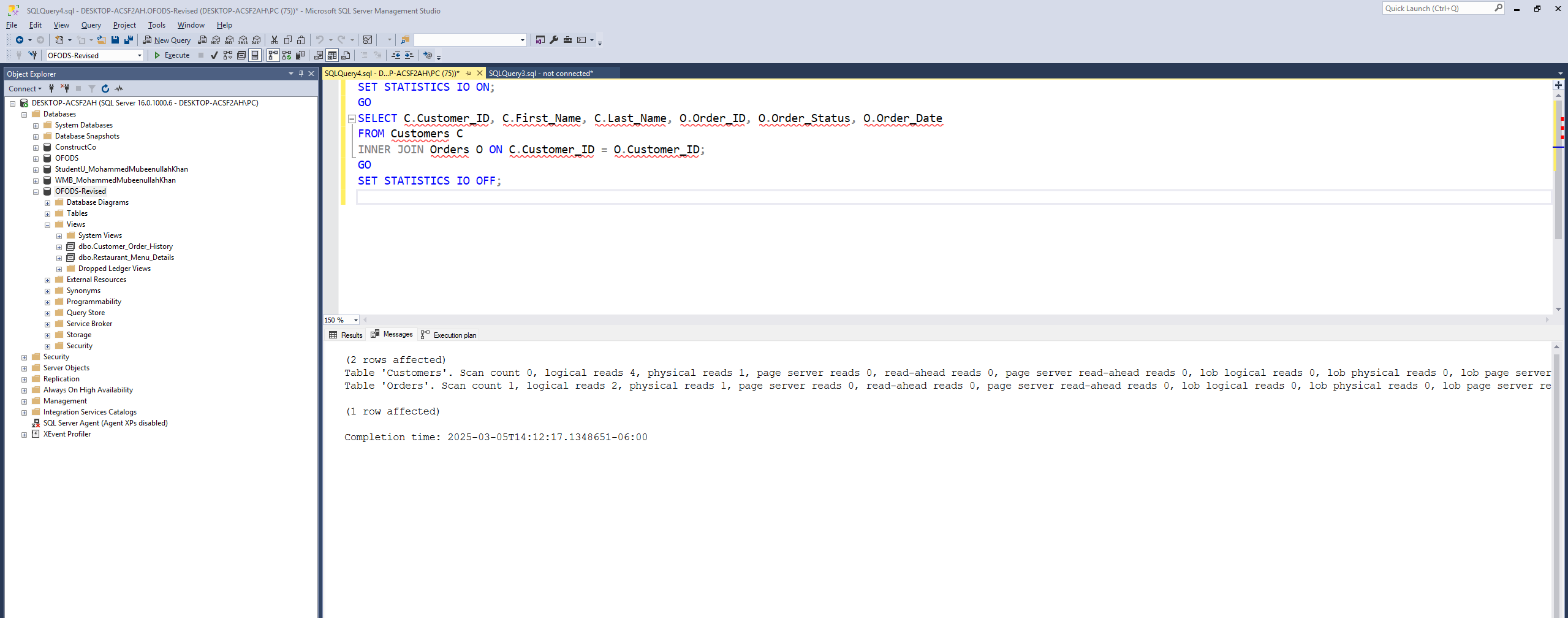
This indicates that the Orders table's clustered index was scanned in full, which could impact performance for large datasets.

Optimizing by adding an index on Customer\_ID in Orders can improve performance.



Query Performance Analysis Using SET STATISTICS IO:

* Customers Table:
  + Scan count 0: SQL Server did not perform a full table scan on Customers, indicating it used an index.
  + Logical reads 4: The number of 8KB pages read from the buffer cache.
  + Physical reads 1: Pages read from disk (not from memory).
* Orders Table:
  + Scan count 1: A full table scan was required.
  + Logical reads 2: SQL Server read 2 pages from the cache.
  + This indicates that indexing on Customer\_ID in Orders could further optimize the join performance.



# J. Views

*Description: Add two views to your database to provide easy access to combinations of data from multiple tables.*

*Rubric: Your work will be graded as follows:*

* *3 points for including the SQL for generating the two views in your Word document*
* *3 points for including screenshots for the data contained in each view in your Word document*
* *3 points for explaining why each view is a valuable addition to your database*

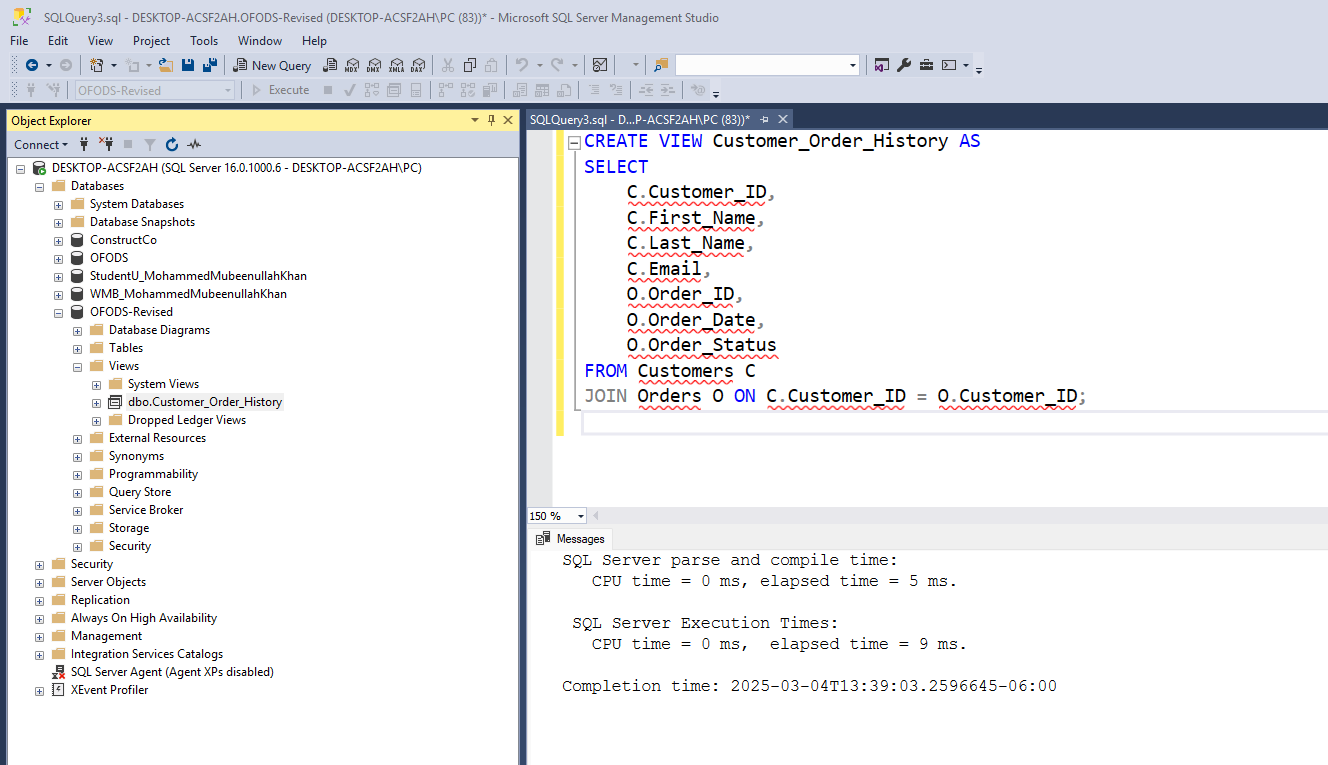
*Total points possible: 9*

ENTER YOUR WORK WITH VIEWS HERE

Customer\_Order\_History View:

This view joins Customers and Orders to show customer order history. It provides a quick lookup of which customers placed orders, their latest order date, and the status. Useful for customer support and tracking orders.

|  |
| --- |
| CREATE VIEW Customer\_Order\_History AS  SELECT  C.Customer\_ID,  C.First\_Name,  C.Last\_Name,  C.Email,  O.Order\_ID,  O.Order\_Date,  O.Order\_Status  FROM Customers C  JOIN Orders O ON C.Customer\_ID = O.Customer\_ID; |



Querying the view:

|  |
| --- |
| SELECT \* FROM Customer\_Order\_History; |

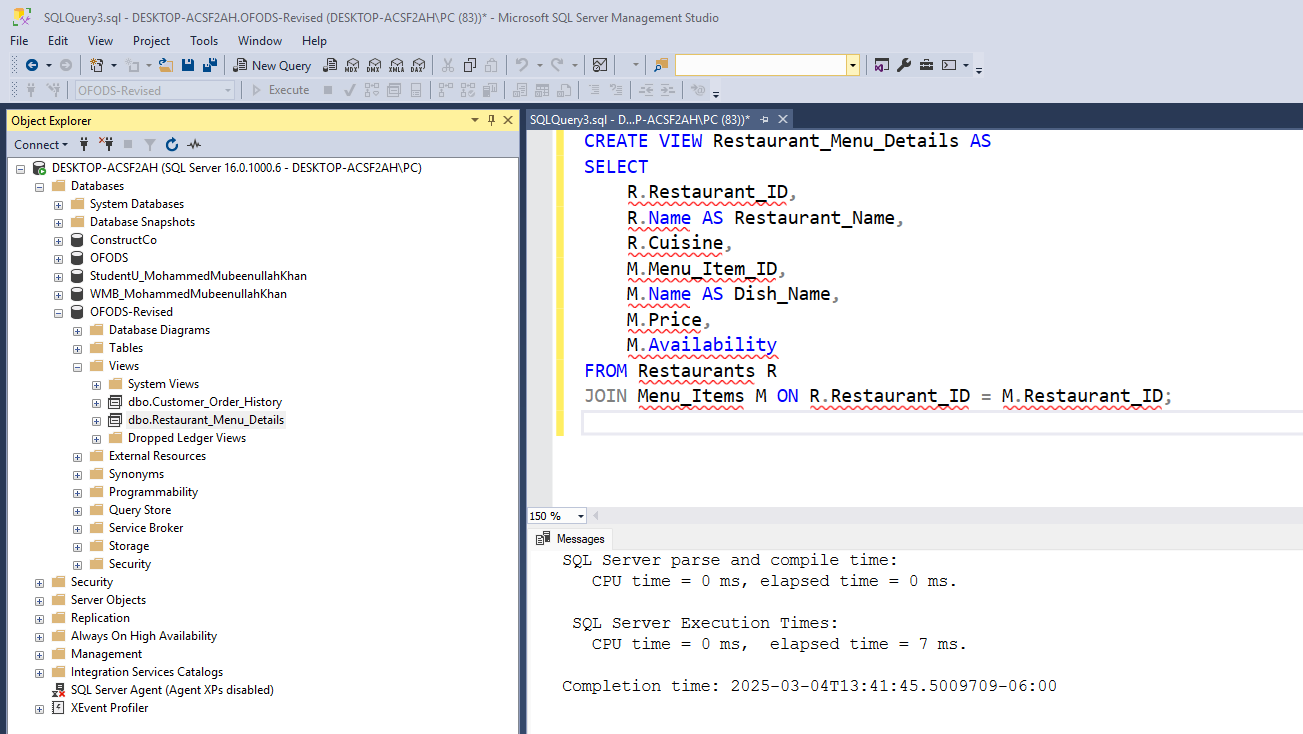
A screenshot of a computer

AI-generated content may be incorrect.

Restaurant\_Menu\_Details View:

This view joins Restaurants and Menu\_Items to show menu details by restaurant. It allows quick retrieval of available dishes at each restaurant, along with pricing. Useful for frontend applications, restaurant management, and menu updates.

|  |
| --- |
| CREATE VIEW Restaurant\_Menu\_Details AS  SELECT  R.Restaurant\_ID,  R.Name AS Restaurant\_Name,  R.Cuisine,  M.Menu\_Item\_ID,  M.Name AS Dish\_Name,  M.Price,  M.Availability  FROM Restaurants R  JOIN Menu\_Items M ON R.Restaurant\_ID = M.Restaurant\_ID; |



Querying the view:

|  |
| --- |
| SELECT \* FROM Restaurant\_Menu\_Details; |

A screenshot of a computer

AI-generated content may be incorrect.

# K. Stored Programs (Stored Procedures, Stored Functions, Triggers)

*Description: Add a stored procedure, stored function or trigger to a table and demonstrate using it.*

*Rubric: Your work will be graded as follows:*

* *3 points for including the SQL for the stored program (procedure, function, or trigger in your Word document*
* *3 points for clearly explaining the purpose of the stored program*
* *3 points for a screenshot and explanation that shows the stored program in action.*

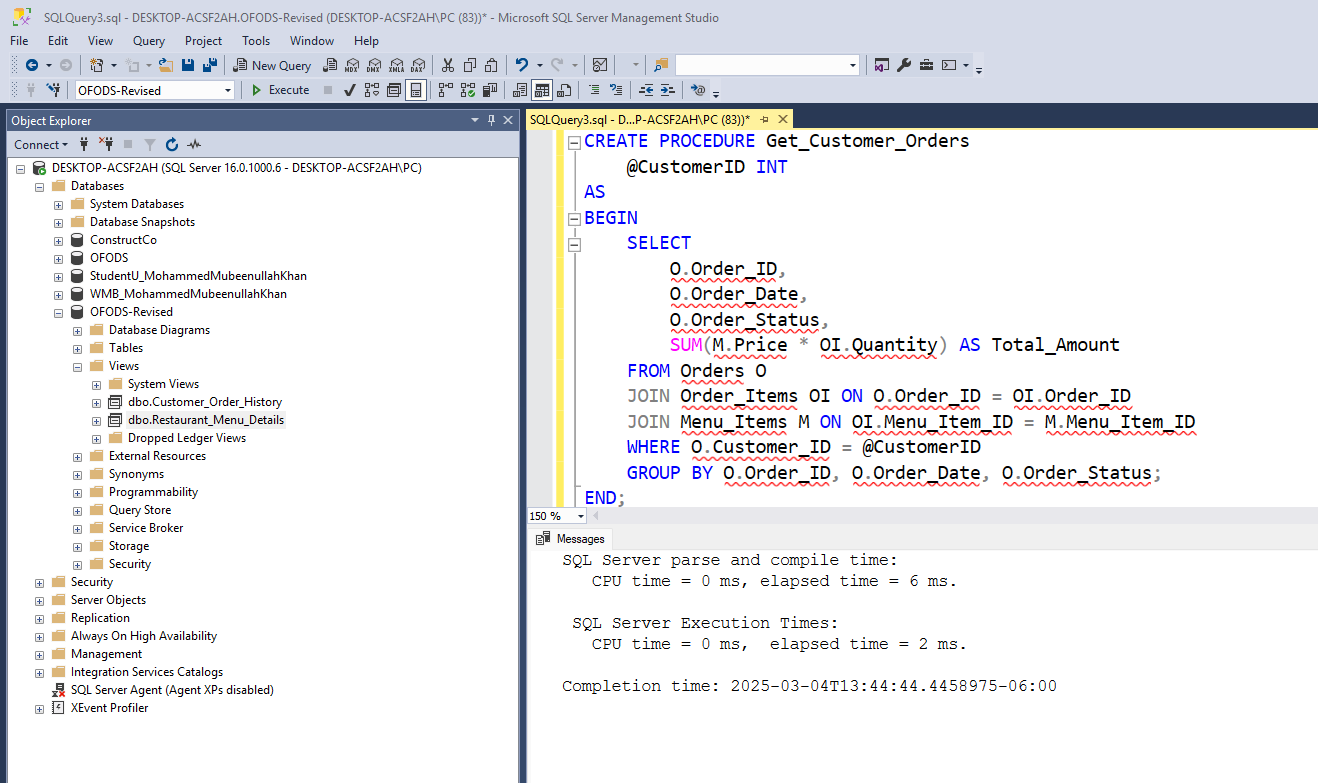
*Total points possible: 9*

ENTER YOUR WORK WITH STORED PROGRAMS HERE

Get\_Customer\_Orders Stored Procedure:

This stored procedure retrieves order details for a given customer based on Customer\_ID. It simplifies order tracking, allowing customer service or analytics teams to quickly fetch data. Instead of writing complex queries repeatedly, users can call this procedure with a single line.

|  |
| --- |
| CREATE PROCEDURE Get\_Customer\_Orders  @CustomerID INT  AS  BEGIN  SELECT  O.Order\_ID,  O.Order\_Date,  O.Order\_Status,  SUM(M.Price \* OI.Quantity) AS Total\_Amount  FROM Orders O  JOIN Order\_Items OI ON O.Order\_ID = OI.Order\_ID  JOIN Menu\_Items M ON OI.Menu\_Item\_ID = M.Menu\_Item\_ID  WHERE O.Customer\_ID = @CustomerID  GROUP BY O.Order\_ID, O.Order\_Date, O.Order\_Status;  END; |



|  |
| --- |
| EXEC Get\_Customer\_Orders @CustomerID = 1; |

A screenshot of a computer

AI-generated content may be incorrect.

The procedure returns all orders placed by a customer. It automatically calculates the total amount spent in each order. It improves query efficiency as users don’t have to write complex joins manually.

# L. Transactions

*Description: Demonstrate that you know how to define and use a transaction. Why are transactions important for ensuring ACID behavior?*

*Rubric: Your work will be graded as follows:*

* *5 points for clearly explaining the importance of transactions to ensuring ACID behavior*
* *3 points for including a screenshot and accompanying explanation of a MySQL transaction.*

*Total points possible: 8*

ENTER YOUR WORK WITH TRANSACTIONS HERE

Transaction:

A transaction in SQL is a sequence of operations that are executed as a single unit of work. Transactions ensure data integrity by following the ACID (Atomicity, Consistency, Isolation, Durability) properties.

Importance:

The ACID Properties:

Atomicity:

* Ensures all operations in a transaction are completed or none at all.
* If a failure occurs midway, all previous changes are rolled back.

Consistency

* Ensures database remains valid before and after a transaction.
* Prevents database corruption due to partial updates.

Isolation

* Ensures concurrent transactions don’t interfere with each other.
* Prevents dirty reads, non-repeatable reads, and phantom reads.

Durability

* Ensuring once a transaction is committed, changes are permanently saved, even in system failures.

**Order Payment Processing Transaction:**

If a customer places an order, both the order details and payment should be inserted. If either fails, the entire transaction must roll back.

|  |
| --- |
| -- Begin transaction  BEGIN TRANSACTION;  -- Insert a new order (assume Customer\_ID = 1)  INSERT INTO Orders (Customer\_ID, Order\_Date, Order\_Status, Delivery\_ID)  VALUES (1, GETDATE(), 'Pending', 2);  -- Get the last inserted Order\_ID  DECLARE @OrderID INT;  SET @OrderID = SCOPE\_IDENTITY();  -- Insert payment for the order  INSERT INTO Payments (Order\_ID, Payment\_Method\_ID, Amount, Status)  VALUES (@OrderID, 1, 500.00, 'Completed');  -- Commit if all operations succeed  COMMIT; |



The transaction executed successfully in SQL Server Management Studio (SSMS), demonstrating the correct implementation of atomic database operations. The purpose of this transaction was to insert a new order into the Orders table, retrieve its Order\_ID, and then insert a corresponding payment record into the Payments table. This ensures that an order and its payment are always recorded together, maintaining data integrity.

# M. Database Security

*Description: Identify the different kinds of users who will use your database. Write GRANT statements to define the privileges for these different kinds of users.*

*Rubric: Your work will be graded as follows:*

* *4 points for clearly identifying and describing the various kinds of users who will use the databases and identifying and justifying what privileges each should have.*
* *4 points for writing GRANT statements that assign privileges to these different kinds of users.*
* *4 points for demonstrating with screenshots that your GRANT statements do distinguish among different kinds of users in regard to what they can do with the database.*

*Total points possible: 12*

ENTER YOUR WORK WITH DATABASE SECURITY HERE

Identification of Different User Roles and Their Privileges

**Administrator (admin\_user)**

**Responsibilities:**  
The administrator is responsible for managing the entire database system, including creating and modifying database objects, overseeing security policies, and maintaining data integrity.

**Privileges:**

* Full access to the database, including SELECT, INSERT, UPDATE, DELETE, and privilege management.
* Ability to create, modify, and drop tables, views, and users.

**Justification:**  
As the highest-level user, the administrator requires unrestricted access to manage all database objects and ensure the system operates efficiently.

**Restaurant Manager (restaurant\_manager)**

**Responsibilities:**  
Restaurant managers oversee menu listings and track customer orders placed at their respective restaurants.

**Privileges:**

* SELECT, INSERT, UPDATE, and DELETE permissions on Menu\_Items to manage menu offerings.
* SELECT permission on Orders to view order details related to their restaurant.

**Justification:**  
Restaurant managers should have full control over menu management but should not be able to modify customer information, payments, or other sensitive data.

**Customer (customer\_user)**

**Responsibilities:**  
Customers use the database to place orders, make payments, and provide feedback on restaurant services.

**Privileges:**

* SELECT, INSERT permissions on Orders to place new orders.
* SELECT, INSERT permissions on Payments to process transactions.
* SELECT, INSERT permissions on Reviews\_Ratings to submit restaurant reviews.

**Justification:**  
Customers should be able to interact with the ordering system but should not modify, delete, or access sensitive restaurant or administrative data.

**Delivery Personnel (delivery\_user)**

**Responsibilities:**  
Delivery personnel are responsible for updating the status of orders assigned to them.

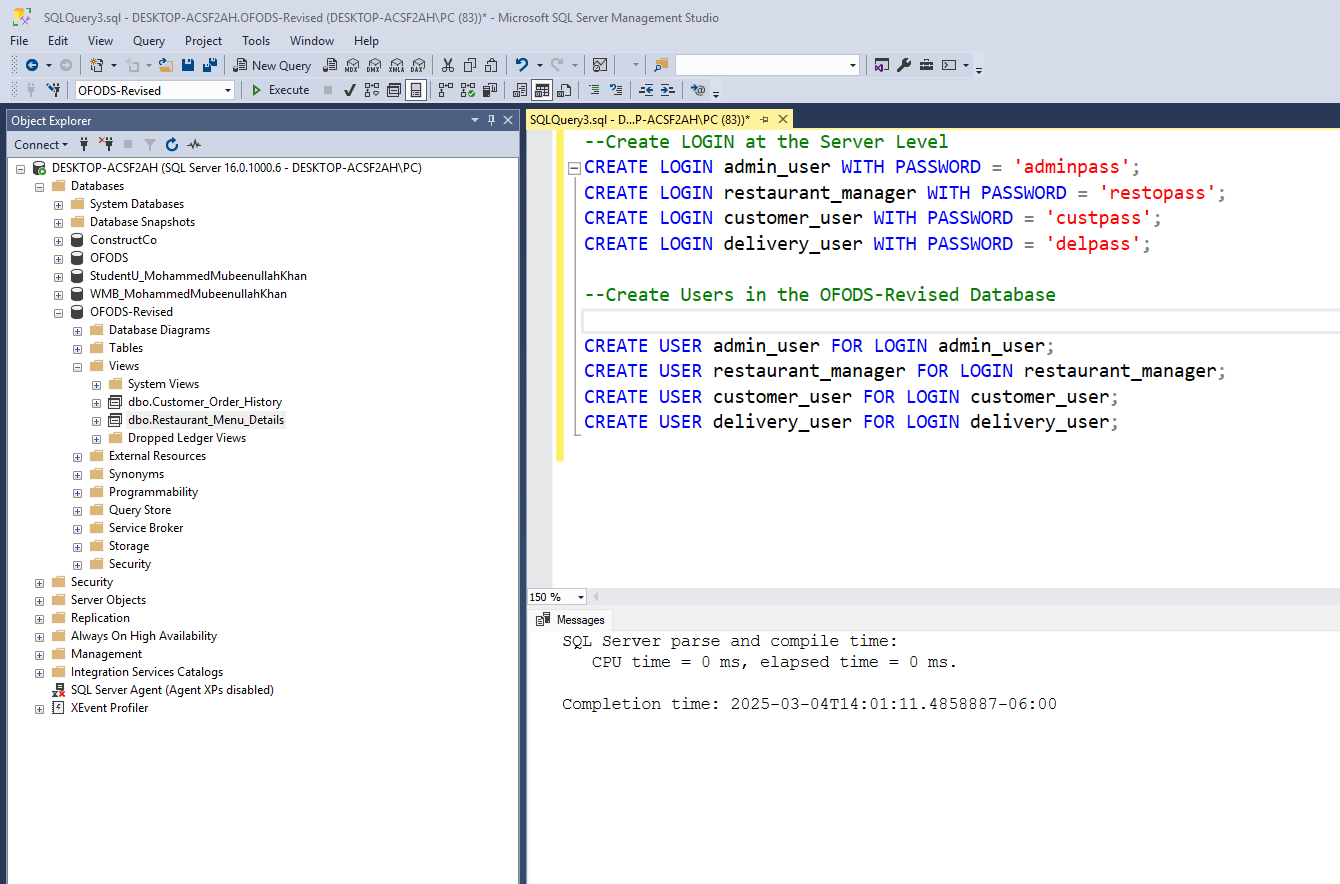
**Privileges:**

* SELECT, UPDATE permissions on Orders to update the status of deliveries.

**Justification:**  
Delivery personnel require access to view and modify order statuses but should not have the ability to alter menu items, payments, or customer details.

Users created:

|  |
| --- |
| --Create LOGIN at the Server Level  CREATE LOGIN admin\_user WITH PASSWORD = 'adminpass';  CREATE LOGIN restaurant\_manager WITH PASSWORD = 'restopass';  CREATE LOGIN customer\_user WITH PASSWORD = 'custpass';  CREATE LOGIN delivery\_user WITH PASSWORD = 'delpass';  --Create Users in the OFODS-Revised Database  CREATE USER admin\_user FOR LOGIN admin\_user;  CREATE USER restaurant\_manager FOR LOGIN restaurant\_manager;  CREATE USER customer\_user FOR LOGIN customer\_user;  CREATE USER delivery\_user FOR LOGIN delivery\_user; |



Verification of user creation:

|  |
| --- |
| SELECT name FROM sys.database\_principals WHERE type = 'S'; |

A screenshot of a computer

AI-generated content may be incorrect.

Grant permissions:

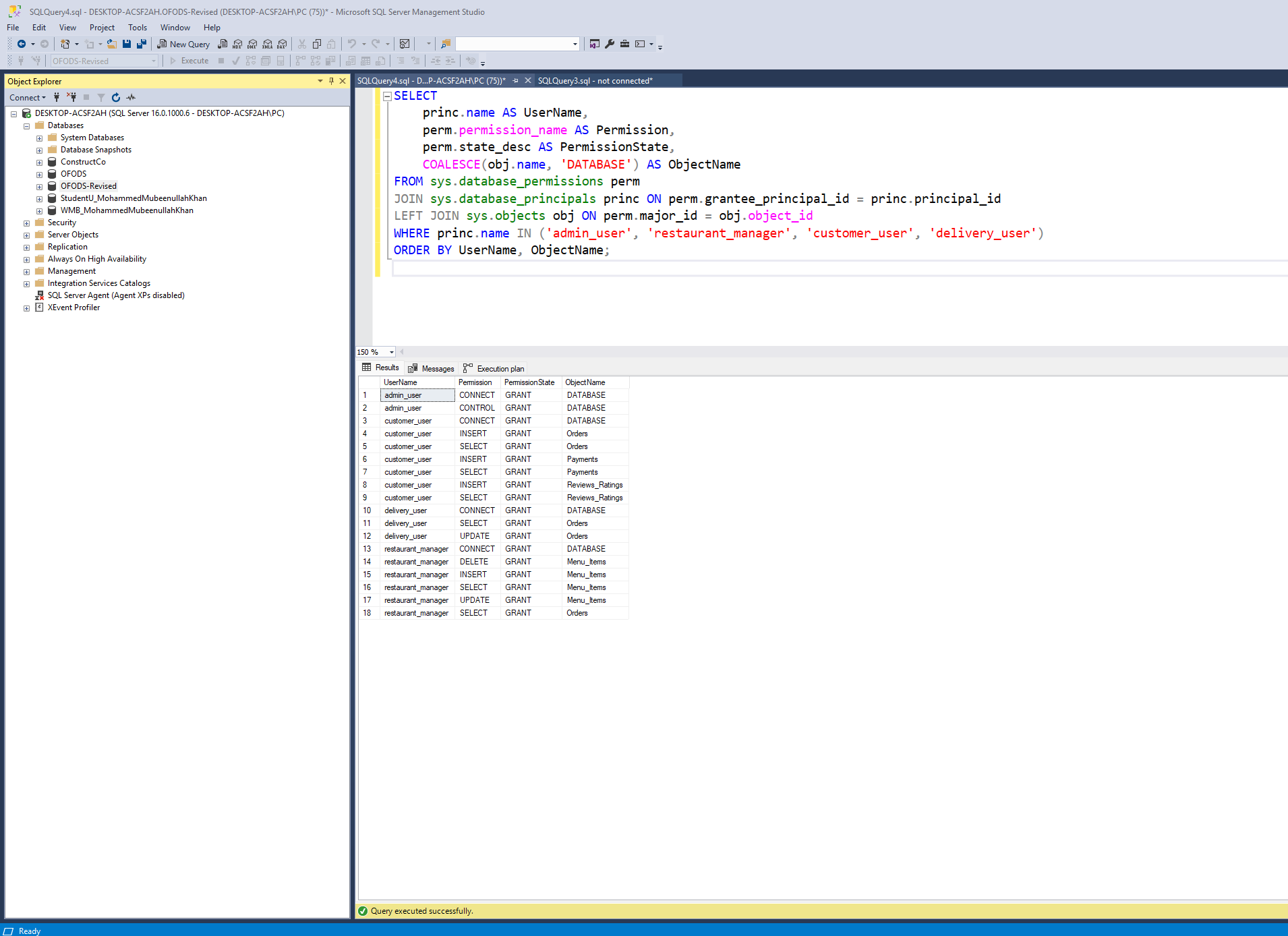
|  |
| --- |
| -- Grant full privileges to admin  GRANT CONTROL ON DATABASE::[OFODS-Revised] TO admin\_user;  -- Grant restaurant manager access to menu and orders  GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Menu\_Items TO restaurant\_manager;  GRANT SELECT ON dbo.Orders TO restaurant\_manager;  -- Grant customer access to place orders and write reviews  GRANT SELECT, INSERT ON dbo.Orders TO customer\_user;  GRANT SELECT, INSERT ON dbo.Payments TO customer\_user;  GRANT SELECT, INSERT ON dbo.Reviews\_Ratings TO customer\_user;  -- Grant delivery personnel access to update order statuses  GRANT SELECT, UPDATE ON dbo.Orders TO delivery\_user; |

A screenshot of a computer

AI-generated content may be incorrect.

Verification of permissions granted to each user:

|  |
| --- |
| SELECT  princ.name AS UserName,  perm.permission\_name AS Permission,  perm.state\_desc AS PermissionState,  COALESCE(obj.name, 'DATABASE') AS ObjectName  FROM sys.database\_permissions perm  JOIN sys.database\_principals princ ON perm.grantee\_principal\_id = princ.principal\_id  LEFT JOIN sys.objects obj ON perm.major\_id = obj.object\_id  WHERE princ.name IN ('admin\_user', 'restaurant\_manager', 'customer\_user', 'delivery\_user')  ORDER BY UserName, ObjectName; |



# N. Locking and Concurrent Access

*Description: Explain the purpose of locking tables and show how to do that to prevent inconsistencies that may arise in your data when concurrent transactions take place.*

*Rubric: Your work will be graded as follows:*

* *3 points for clearly explaining an example that shows why you should lock tables to prevent inconsistencies.*
* *3 points for providing a screenshot and accompanying explanation of locking tables.*

*Total points possible: 5*

ENTER YOUR WORK WITH LOCKING AND CONCURRENT ACCESS HERE

Table Locking:

Table locking is a mechanism used in SQL Server to prevent concurrent transactions from modifying data inconsistently. When multiple users or processes access the same table simultaneously, locking ensures that data integrity is maintained by preventing conflicting operations.

Example - Preventing Double Payment Processing:

A customer is making a payment for an order, and at the same time, another transaction tries to update the payment status.

* If both transactions proceed without locking, the system might process duplicate payments or inconsistent updates.

**Without locking:**

Assume two users execute the following queries simultaneously:

|  |
| --- |
| SELECT Status FROM Payments WHERE Payment\_ID = 201;  -- Both transactions see Status = 'Pending' |

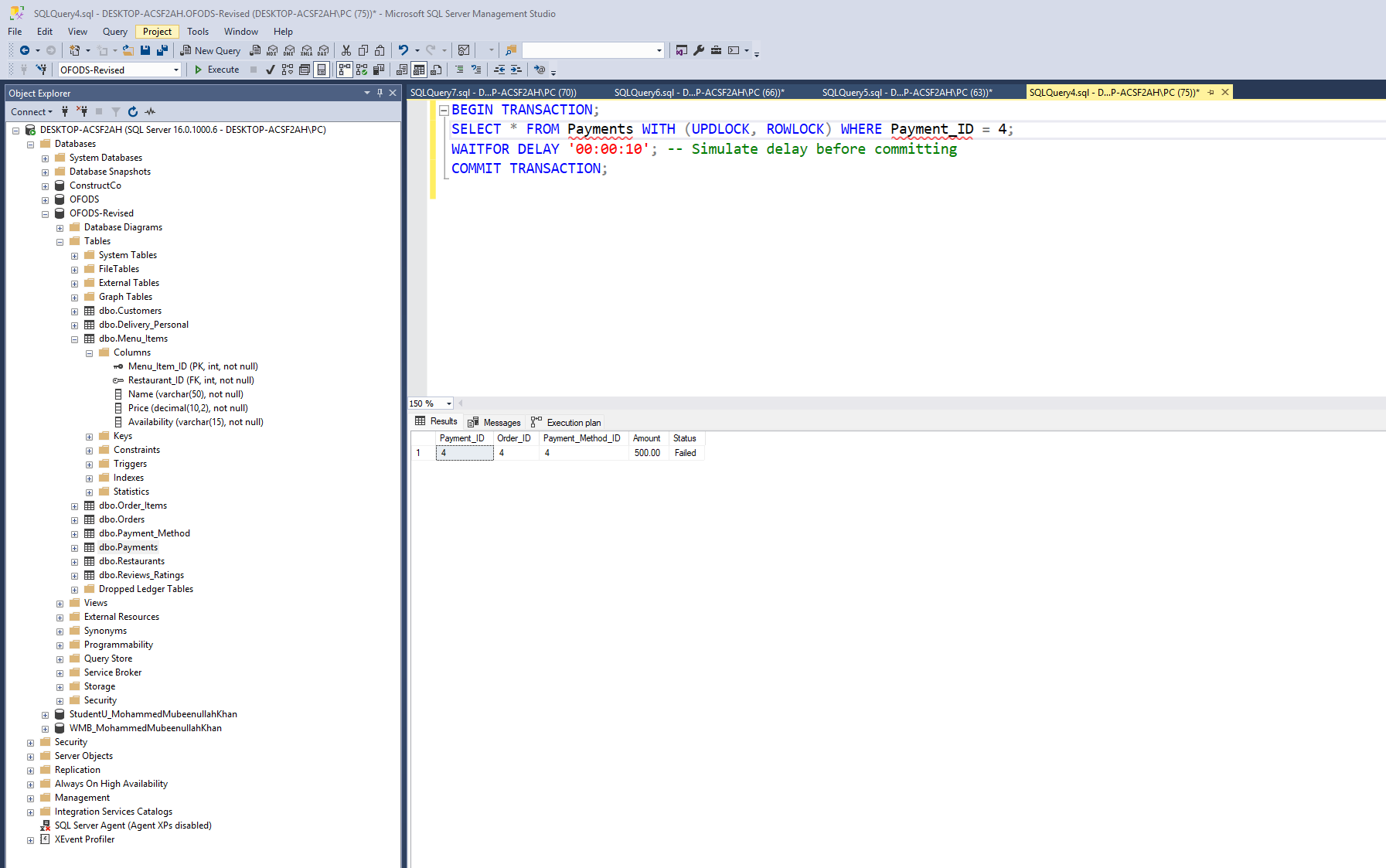
Then they both try to process the payment:

|  |
| --- |
| UPDATE Payments SET Status = 'Completed' WHERE Payment\_ID = 201;  -- Both succeed, potentially causing duplicate processing |

If payment processing logic depends on the Pending status, both transactions might execute, causing double processing.

I executed these two queries one by one, and 2nd query took long time to execute, further explanation what happened is explained below with the respective queries.

|  |
| --- |
| BEGIN TRANSACTION;  SELECT \* FROM Payments WITH (UPDLOCK, ROWLOCK) WHERE Payment\_ID = 4;  WAITFOR DELAY '00:00:10'; -- Simulate delay before committing  COMMIT TRANSACTION; |



The UPDLOCK and ROWLOCK ensure that this particular row (Payment\_ID = 4) is locked for updates.

The WAITFOR DELAY simulates a real-world delay before committing.

Any other transaction that tries to update the same row will be blocked until the first transaction completes.

|  |
| --- |
| UPDATE Payments SET Status = 'Completed' WHERE Payment\_ID = 4; |

A screenshot of a computer

AI-generated content may be incorrect.

The second transaction was forced to wait until the first transaction was committed.

Once the first transaction released the lock, this update successfully executed (1 row affected).

# O. Backing Up Your Database

*Description: How you will back up your database. What commands will you issue? How frequently will the commands run? How can they be automated? Where will the backups be stored?*

*Rubric: Your work will be graded as follows:*

* *6 points for clearly explaining and justifying your database backup strategy, including the frequency with which you will back up the database, how you will automate backups, where you will store them, and how you will secure them. You will earn three points for addressing each factor (frequency, location, automation, and security)*
* *2 points for providing a screenshot of the command you would issue to back up the database and for including a portion of the resulting file.*

*Total points possible: 8*

ENTER YOUR WORK ON DATABASE BACKUPS HERE

**Importance of Database Backup**

Backing up the database is critical to ensure data availability, recovery from failures, and protection against accidental deletions or cyberattacks. A well-defined backup strategy should consider:

* Backup Frequency
* Automation
* Storage Location
* Security Measures

**Backup Frequency:**

|  |  |  |
| --- | --- | --- |
| **Backup Type** | **Frequency** | **Justification** |
| Full Backup | Daily (Midnight, off-peak hours) | Ensures complete data recovery in case of failure. |
| Differential Backup | Every 4 Hours | Reduces recovery time by storing changes since the last full backup. |
| Transaction Log Backup | Every 15 Minutes | Ensures point-in-time recovery and minimizes data loss. |

* Full backups ensure complete database recovery.
* Differential backups allow faster restoration than full backups.
* Transaction log backups enable point-in-time recovery.

**Automating Backups:**

SQL Server Agent Job (Scheduled Backups)

Full Database Backup (Daily)

|  |
| --- |
| BACKUP DATABASE [OFODS-Revised]  TO DISK = 'C:\SQLBackups\OFODS-Revised\_Full.bak'  WITH FORMAT, INIT,  NAME = 'Full Database Backup',  SKIP, NOREWIND, NOUNLOAD, STATS = 10; |

FORMAT, INIT → Overwrites existing backup files.

STATS = 10 → Provides progress updates.

Differential Backup (Every 4 Hours)

|  |
| --- |
| BACKUP DATABASE [OFODS-Revised]  TO DISK = 'C:\SQLBackups\OFODS-Revised\_Diff.bak'  WITH DIFFERENTIAL, INIT,  NAME = 'Differential Backup',  STATS = 10; |

Backs up only changes since the last full backup, reducing storage use.

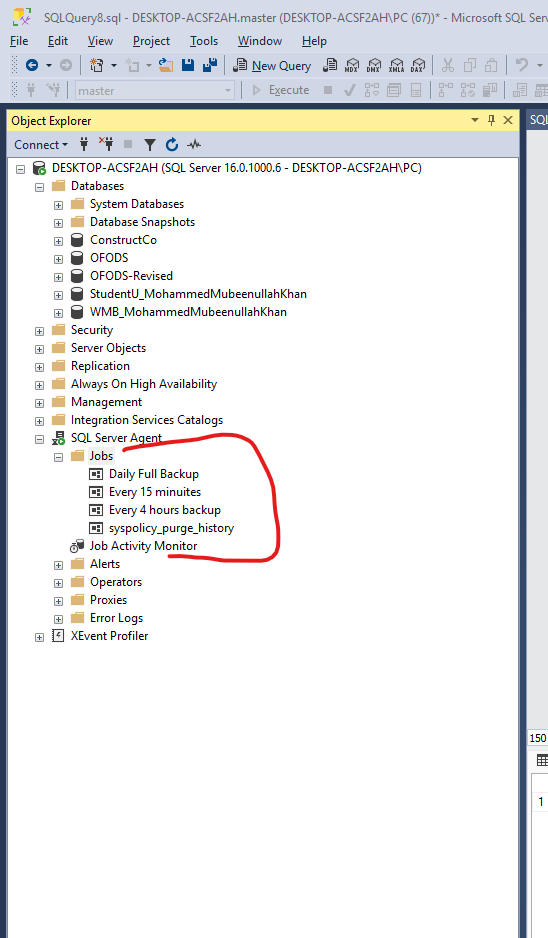
Transaction Log Backup (Every 15 Minutes)

|  |
| --- |
| BACKUP LOG [OFODS-Revised]  TO DISK = 'C:\SQLBackups\OFODS-Revised\_Log.trn'  WITH INIT, NAME = 'Transaction Log Backup'; |

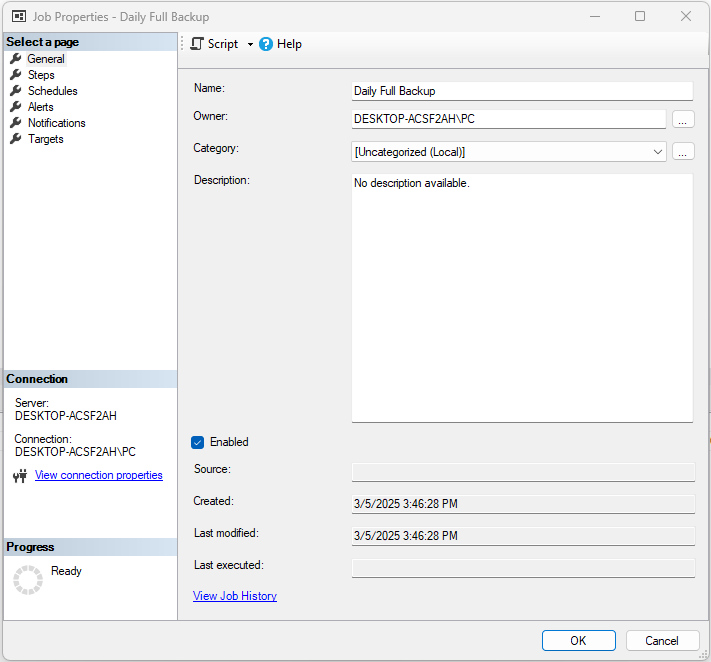
Ensures point-in-time recovery.

Automating Backups Using SQL Server Agent:

* Open SSMS → Expand "SQL Server Agent" → Right-click "Jobs" → New Job.
* Enter Job Name: e.g., Daily Full Backup.
* Go to "Steps" → Click "New" → Paste Backup Command.
* Go to "Schedules" → Create a Schedule:
  + Full Backup: Every day at 12:00 AM.
  + Differential Backup: Every 4 hours.
  + Transaction Log Backup: Every 15 minutes.
* Click OK → Start the Job.



Daily backup:

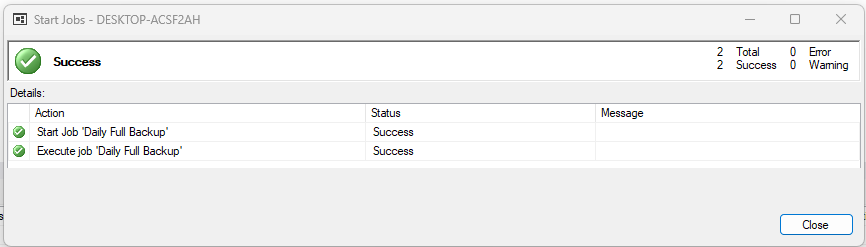


A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

Every 4 hours backup:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Every 15 minutes backup:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Storage Location for Backups:**

|  |  |  |
| --- | --- | --- |
| **Storage Type** | **Description** | **Justification** |
| Local Disk (C:\SQLBackups) | Stores recent backups for quick access. | Useful for immediate recovery. |
| Network Storage (NAS/Shared Drive) | Copies backups to a secure network location. | Ensures backups remain accessible if the server crashes. |
| Cloud Backup (Azure/AWS S3/Google Cloud) | Offsite storage for disaster recovery. | Protects against local hardware failures and ransomware. |

* Local backups allow quick recovery.
* Network backups ensure redundancy.
* Cloud backups provide disaster recovery protection.

**Security Measures for Backup Protection:**

|  |  |  |
| --- | --- | --- |
| **Security Method** | **Description** | **Justification** |
| Backup Encryption | Use ENCRYPTION option in BACKUP DATABASE. | Prevents unauthorized access to backup files. |
| Access Control | Restrict backup folder access to admins only. | Ensures only authorized users manage backups. |
| Offsite Storage | Store copies in a secure cloud storage. | Prevents loss from physical damage or ransomware. |
| Automated Integrity Checks | Use RESTORE VERIFYONLY after each backup. | Ensures backups are valid and usable. |

Encrypting Backups:

|  |
| --- |
| BACKUP DATABASE [OFODS-Revised]  TO DISK = 'C:\SQLBackups\OFODS-Revised\_Encrypted.bak'  WITH ENCRYPTION (ALGORITHM = AES\_256, SERVER CERTIFICATE = MyBackupCert); |

# P. Programming

*Description: Write a Python, Java, or PHP program that generates a report that contains a subset of the data from your database. Include the code for your Python program in your Word document, and also post the program to your GitHub repository.*

*Rubric: Your work will be graded as follows:*

* *10 points for writing a Python script (and including its code in the Word doc) that will pull data from a database and store it to a text file and present it to the screen. Your code must have comments in it that explain how it works. You will be awarded 3 points for successfully connecting to the database, 3 points for successfully querying it, and 4 points for presenting the data to the screen and to a file. Internal comments count for 2 points.*
* *2 points for posting the code to GitHub*
* *6 points for showing a screenshot of your running the script and showing the results it produces on the screen.*

*Total points possible: 18*

ENTER YOUR PYTHON, PHP, or JAVA DATABASE PROGRAMMING WORK HERE

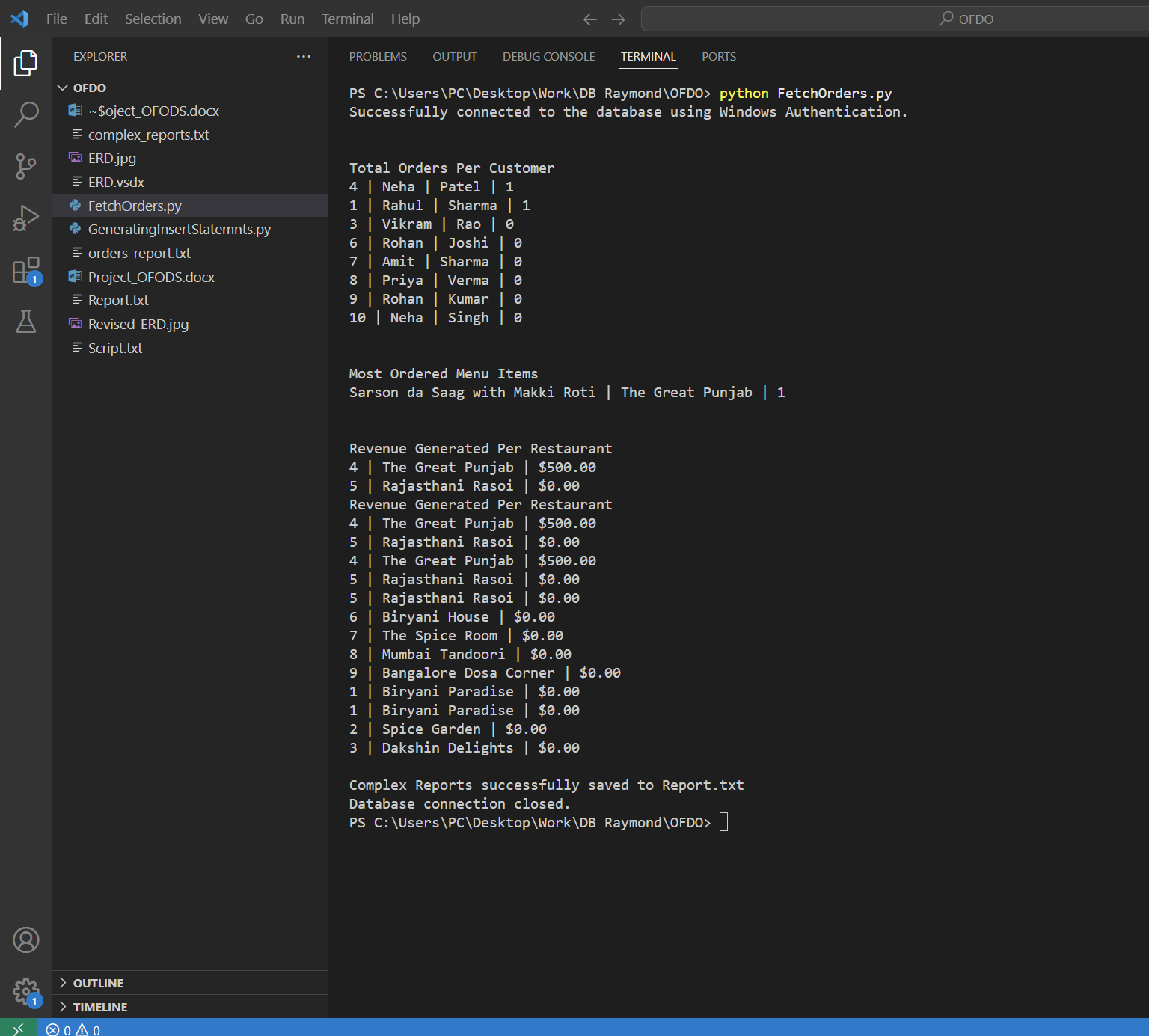
Code:

|  |
| --- |
| import pyodbc  # Define database connection parameters (Windows Authentication)  server = 'DESKTOP-ACSF2AH'  # Your SQL Server name  database = 'OFODS-Revised'  # Your database name  try:      # Establish connection using Windows Authentication      conn = pyodbc.connect(          f'DRIVER={{SQL Server}};SERVER={server};DATABASE={database};Trusted\_Connection=yes;'      )      cursor = conn.cursor()      print("Successfully connected to the database using Windows Authentication.\n")      # QUERY 1: Get total orders per customer (Removing HAVING clause to ensure all customers appear)      query1 = """      SELECT C.Customer\_ID, C.First\_Name, C.Last\_Name, COUNT(O.Order\_ID) AS Total\_Orders      FROM Customers C      LEFT JOIN Orders O ON C.Customer\_ID = O.Customer\_ID      GROUP BY C.Customer\_ID, C.First\_Name, C.Last\_Name      ORDER BY Total\_Orders DESC;      """      # QUERY 2: Get the most ordered menu items (Removing TOP 5 for testing)      query2 = """      SELECT MI.Name AS Menu\_Item, R.Name AS Restaurant, COUNT(OI.Menu\_Item\_ID) AS Times\_Ordered      FROM Order\_Items OI      JOIN Menu\_Items MI ON OI.Menu\_Item\_ID = MI.Menu\_Item\_ID      JOIN Restaurants R ON MI.Restaurant\_ID = R.Restaurant\_ID      GROUP BY MI.Name, R.Name      ORDER BY Times\_Ordered DESC;      """      # QUERY 3: Get revenue generated per restaurant (Fixing alias issue)      query3 = """      SELECT R.Restaurant\_ID, R.Name AS Restaurant\_Name, ISNULL(SUM(P.Amount), 0) AS Total\_Revenue      FROM Restaurants R      LEFT JOIN Menu\_Items MI ON R.Restaurant\_ID = MI.Restaurant\_ID      LEFT JOIN Order\_Items OI ON MI.Menu\_Item\_ID = OI.Menu\_Item\_ID      LEFT JOIN Orders O ON OI.Order\_ID = O.Order\_ID      LEFT JOIN Payments P ON O.Order\_ID = P.Order\_ID      GROUP BY R.Restaurant\_ID, R.Name      ORDER BY Total\_Revenue DESC;      """      # Execute queries      cursor.execute(query1)      results1 = cursor.fetchall()      cursor.execute(query2)      results2 = cursor.fetchall()      cursor.execute(query3)      results3 = cursor.fetchall()      # Define output file      output\_file = "Report.txt"      # Open the file to write results      with open(output\_file, "w", encoding="utf-8") as file:          # Print & Write Query 1 Results          print("\nTotal Orders Per Customer")          file.write("Total Orders Per Customer\n")          file.write("Customer\_ID | First\_Name | Last\_Name | Total\_Orders\n")          file.write("-" \* 60 + "\n")          if results1:              for row in results1:                  line = f"{row[0]} | {row[1]} | {row[2]} | {row[3]}"                  print(line)  # Print to terminal                  file.write(line + "\n")  # Write to file          else:              print("No data found for Total Orders Per Customer.")              file.write("No data found.\n")          print("\n")  # Add spacing between sections          file.write("\n\n")          # Print & Write Query 2 Results          print("Most Ordered Menu Items")          file.write("Most Ordered Menu Items\n")          file.write("Menu\_Item | Restaurant | Times\_Ordered\n")          file.write("-" \* 60 + "\n")          if results2:              for row in results2:                  line = f"{row[0]} | {row[1]} | {row[2]}"                  print(line)                  file.write(line + "\n")          else:              print("No data found for Most Ordered Menu Items.")              file.write("No data found.\n")          print("\n")  # Add spacing between sections          file.write("\n\n")          # Print & Write Query 3 Results          print("Revenue Generated Per Restaurant")          file.write("Revenue Generated Per Restaurant\n")          file.write("Restaurant\_ID | Restaurant\_Name | Total\_Revenue\n")          file.write("-" \* 60 + "\n")          if results3:              for row in results3:                  line = f"{row[0]} | {row[1]} | ${row[2]:.2f}"                  print(line)                  file.write(line + "\n")          else:              print("No data found for Revenue Generated Per Restaurant.")              file.write("No data found.\n")      print(f"\nComplex Reports successfully saved to {output\_file}")  except pyodbc.Error as e:      print(f"Database Error: {e}")  except Exception as e:      print(f"Unexpected Error: {e}")  finally:      # Close the connection      if 'conn' in locals():          conn.close()          print("Database connection closed.") |

A screen shot of a computer

AI-generated content may be incorrect.

Output:



Written to File:

A screenshot of a computer

AI-generated content may be incorrect.

# Q. Suggested Future Work

*Description: Describe the limitations of your current database and explain how you or someone else could improve the design to address these shortcomings. Also describe how you might take advantage of leverage cloud services to increase the performance and availability of your database. Finally, explain the advantages and disadvantages of storing your data in a NoSQL format instead.*

*Rubric: Your work will be graded as follows:*

* *3 points for clearly describing the limitations of your databases*
* *3 points for explaining how you would address these shortcomings*
* *3 points for explaining how you might migrate the database to the cloud and describing what advantages you might gain from doing that.*
* *3 points for explaining the advantages and disadvantages of storing your data in a document-based NoSQL format instead.*

*Total points possible: 12*

ENTER YOUR SUGGESTED FUTURE WORK IDEAS HERE

**Limitations of the Current Database Design**

The current database structure effectively manages restaurant orders, payments, and customer interactions. However, it presents several limitations that may affect performance, scalability, and overall efficiency.

**Scalability Issues**

As the database grows, the reliance on multiple joins across relational tables can cause performance degradation. A higher volume of customers, restaurants, and orders increases query execution time, making the system slower.

**Lack of Real-Time Analytics**

The current system does not support real-time monitoring of orders and payments. Restaurant managers and customers cannot dynamically track order statuses, which affects operational efficiency.

**Limited Fault Tolerance**

The database runs on a single SQL Server instance without built-in failover or replication mechanisms. This increases the risk of downtime in case of server failures.

**High Query Complexity**

Generating reports involves executing complex SQL queries that require multiple joins and aggregations. With large datasets, these queries may take considerable time to process.

**Storage Inefficiency**

Storing historical data such as past orders, customer reviews, and interactions within relational tables can lead to data bloating, slowing down query performance.

**Strategies to Improve Database Performance**

To address these shortcomings, the database design should be optimized using the following approaches.

**Implementing Indexing and Query Optimization**

One of the most effective ways to enhance database performance is through indexing. Indexing frequently queried fields such as Customer\_ID, Order\_ID, and Menu\_Item\_ID can significantly reduce search times. Additionally, denormalizing some tables by storing frequently accessed order details separately can optimize data retrieval speeds. Table partitioning can also be used to distribute large tables across multiple storage locations for better performance.

**Real-Time Data Processing**

To enable real-time tracking of orders, event-driven architectures such as Apache Kafka or RabbitMQ can be implemented. These systems capture updates and distribute them in real time, ensuring that restaurant managers and customers receive live updates. Materialized views can also be created to precompute commonly used reports, reducing the load on the main database.

**Enhancing Fault Tolerance**

Fault tolerance can be improved by setting up database replication mechanisms such as SQL Server Always On Availability Groups or log shipping. This ensures that a backup copy of the database is available, reducing downtime in case of a failure. Regular backups should also be scheduled to prevent data loss.

**Caching Frequently Used Queries**

Caching mechanisms such as Redis or Memcached can be integrated to store frequently used queries. By caching results of queries that do not change frequently, the system reduces the load on SQL Server, leading to faster response times.

**Cloud Migration and Its Benefits**

Migrating the database to a cloud-based platform such as Microsoft Azure SQL Database, Amazon RDS for SQL Server, or Google Cloud SQL provides several advantages.

**Advantages of Cloud Migration**

Cloud-hosted databases offer automatic scaling, allowing resources to be allocated dynamically based on demand. This ensures smooth operation during peak hours. Additionally, cloud solutions provide high availability and disaster recovery, as they include built-in replication, automatic backups, and failover mechanisms. By migrating to a cloud-based solution, the need for manual maintenance is minimized, as cloud providers handle software updates, patches, and infrastructure management. Cloud-hosted databases are also globally accessible, allowing restaurant owners to manage their data across multiple locations.

**Challenges of Cloud Migration**

Despite its benefits, cloud migration introduces certain challenges. Network latency can impact performance due to delays in transmitting data between application servers and cloud databases. Additionally, security concerns arise when storing customer and payment details in the cloud, requiring strict encryption and access control. Vendor lock-in is another issue, as once a database is migrated to a specific cloud provider, switching to another provider may require extensive reconfiguration.

**SQL vs. NoSQL: A Comparative Analysis**

While traditional SQL databases provide a structured approach to data management, **NoSQL databases** (such as MongoDB, Firebase, or CouchDB) offer an alternative method for handling large volumes of unstructured data.

**Advantages of NoSQL Databases**

NoSQL databases provide a flexible schema, making it easier to store dynamic restaurant data, menu items, and promotions. They are optimized for key-value lookups, which makes retrieving frequently accessed data significantly faster. Additionally, NoSQL databases offer horizontal scalability, allowing them to distribute large datasets across multiple servers.

**Disadvantages of NoSQL Databases**

One major drawback of NoSQL databases is that they lack ACID compliance, making them unsuitable for applications that require strong transactional consistency, such as payment processing. NoSQL databases may also store redundant data, leading to increased storage costs. Furthermore, querying capabilities in NoSQL are limited compared to SQL databases, making it difficult to perform complex joins and aggregations.

# R. Activity Log

*Description: As an appendix, the team will keep a frequently updated diary or log of their activity. What did you or your team study in this class each day? What did you learn? What did you accomplish or build or design? You don't have to enter something every day, but there should be at least three entries each week. Since we have eight weeks, that means you should make 3 posts to the Activity Log each week, for a total of at least 24 posts. Each post will be worth 1 point.*

*If you are working as part of a team, make sure you clearly identify which team member worked on which tasks. The Activity Log should help me figure out how each team member contributed to the project. If I cannot discern who worked on what aspects of the project from the activity log, no points will be awarded for it.*

*Total points possible: 24*

MAKE AT LEAST THREE ENTRIES PER WEEK. CLEARLY IDENTIFY WHAT EACH PERSON ON YOUR TEAM ACCOMPLISHED. YOU MUST SHARE THE RESPONSIBILITY OF COMPLETING THE PROJECT.

|  |  |  |
| --- | --- | --- |
| Shortlisted some projects and then after analyzing that we can complete it selected 1 project | Avinash Varma Saidam | Week 1 |
| Shortlisted some projects and then after analyzing that we can complete it selected 1 project | Sravanthi Domakonda | Week 1 |
| Completed the research for the questions for Initial Proposal | Both | Week 1 |
| Found out the data and made files after finding suitable data structures | Avinash Varma Saidam | Week 2 |
| Converted data files into tables | Sravanthi Domakonda | Week 2 |
| Finalized A, B, C and D parts and inserted then in this template | Both | Week 2 |
| Find out the relationships between entities and find functional dependencies | Avinash Varma Saidam | Week 3 |
| Performed the Normalization Step | Sravanthi Domakonda | Week 3 |
| Draw the ERD | Sravanthi Domakonda | Week 3 |
| Python script to generate insert statements | Avinash Varma Saidam | Week 4 |
| Wrote the Table Creation Statements | Sravanthi Domakonda | Week 4 |
| Wrote Insert Statemnts | Avinash Varma Saidam | Week 4 |
| Verified the Database on SSMS | Sravanthi Domakonda | Week 4 |
| Revised the design | Avinash Varma Saidam | Week 5 |
| Write the schema again | Sravanthi Domakonda | Week 5 |
| Populated the Database | Avinash Varma Saidam | Week 5 |
| DML statements | Sravanthi Domakonda Avinash Varma Saidam | Week 6 |
| Indexing | Avinash Varma Saidam | Week 6 |
| Transactions | Sravanthi Domakonda | Week 6 |
| Security | Avinash Varma Saidam | Week 6 |
| Join statements revised | Avinash Varma Saidam | Week 7 |
| Indexing performance revised | Avinash Varma Saidam | Week 7 |
| Stored Procedure | Sravanthi Domakonda | Week 7 |
| Locking Database Tables | Sravanthi Domakonda | Week 7 |
| Backup Database | Sravanthi Domakonda Avinash Varma Saidam | Week 7 |
| Programming | Sravanthi Domakonda Avinash Varma Saidam | Week 7 |
| Future Work and Suggestions | Sravanthi Domakonda Avinash Varma Saidam | Week 7 |