

***NAME : AINDREE AARYA***

***STUDENT\_ID: @00787449***

***Advanced Databases***

Table of Contents

[***Task 1: Design and Implementation of an Airport E-Boarding Database System*** 5](#_Toc196428570)

[**1. Introduction** 5](#_Toc196428571)

[**2.Part 1: Database Design and Normalization** 5](#_Toc196428572)

[**2.1. Creating the Database** 6](#_Toc196428573)

[**2.2.Creating the Employees Table** 6](#_Toc196428574)

[**2.3.Creating the Passengers Table** 7](#_Toc196428575)

[**2.4.Creating the Flights Table** 8](#_Toc196428576)

[**2.5.Creating the Reservations Table** 8](#_Toc196428577)

[**2.6.Creating the Tickets Table** 8](#_Toc196428578)

[**2.7.Creating the Baggage Table** 9](#_Toc196428579)

[**2.8. Verifying Table Creation** 10](#_Toc196428580)

[**3.Part 2: Inserting Sample Data and Basic Testing** 10](#_Toc196428581)

[**3.1.Employee Table Data Insertion** 11](#_Toc196428582)

[**3.2.Passangers Table Data Insertion** 12](#_Toc196428583)

[**3.3.Flights Table Data Insertion** 13](#_Toc196428584)

[***3.4.Reservations Table Data Insertion*** 13](#_Toc196428585)

[***3.5.Tickets Table Data Insertion*** 14](#_Toc196428586)

[***3.6.Baggage Table Data Insertion*** 15](#_Toc196428587)

[***3.7.Data Verification Across All Tables*** 16](#_Toc196428588)

[**4.Database Structure Summary** 17](#_Toc196428589)

[**4.1.ER Diagram of the Airport Ticketing System** 17](#_Toc196428590)

[**5.(Task 1.2) Task: Identify Passengers with Pending Reservations** 18](#_Toc196428591)

[**6.(Task 1.3) Task: Find Passengers Aged 40 and Above** 19](#_Toc196428592)

[**7.(Task 1.4 a) Create a Stored Procedure to Search Passengers by Last** Name 20](#_Toc196428593)

[**8.(Task 1.4 b) Stored Procedure – Business Class Reservations Made Today** 21](#_Toc196428594)

[**9.(Task 1.4 c): Insert a New Employee Using a Stored Procedure** 22](#_Toc196428595)

[**10.Task 1.4(d) – Stored Procedure to Update Passenger Details If Booked Before** 23](#_Toc196428596)

[**11.Task 1.5 – Enhanced View: Tickets Issued and Total Revenue Including Extras** 25](#_Toc196428597)

[**12.Task 1.6 – Create a Trigger to Automatically Update Seat Status** 26](#_Toc196428598)

[**13.(Task 1.7) – Scalar Function: Checked-In Baggage Count** 27](#_Toc196428599)

[**14.Bonus Query – Flights with the Most Total Baggage Weight** 28](#_Toc196428600)

[**15.Bonus Query – Number of Tickets Issued by Each Employee** 29](#_Toc196428601)

[**16.Making the Database Secure, Reliable, and Recoverable** 30](#_Toc196428602)

[**16.1.Data Integrity** 30](#_Toc196428603)

[**16.2.Security** 31](#_Toc196428604)

[**16.3.Backup and Recovery** 31](#_Toc196428605)

[**17.Creating a Read-Only User for Security** 31](#_Toc196428606)

[**18. Step: Verifying and Granting Permissions to Read-Only User (SQL Authentication)** 32](#_Toc196428607)

[**19.Taking a Backup of the Database** 35](#_Toc196428608)

[**20.Conclusion** 36](#_Toc196428609)

[***Task 2: Online Shopping Database – Data Import, Querying, and Analysis*** 37](#_Toc196428610)

[**1.Introduction** 37](#_Toc196428611)

[**2. Data Import and Table Creation** 38](#_Toc196428612)

[**3.Importing Tables and Verifying Data** 38](#_Toc196428613)

[**3.1.Data Types and Field Design Rationale** 40](#_Toc196428614)

[**4.Creating Relationships and Diagram** 43](#_Toc196428615)

[**4.1. Entity Relationship (ER) Diagram** 45](#_Toc196428616)

[**5.Queries** 46](#_Toc196428617)

[**5.1.Query 2: Customers with Total Order Amount Between £500 and £1000** 46](#_Toc196428618)

[**5.2.Query 3: Total Amount Paid by UK Customers Who Ordered More Than Three Products** 47](#_Toc196428619)

[**5.3.Query 4: Highest and Second Highest VAT-Adjusted Payments from UK or Australia (Rounded)** 48](#_Toc196428620)

[**5.4.Query 5: Total Quantity Purchased for Each Product (Sorted by Quantity)** 49](#_Toc196428621)

[**5.5.Query 6: Stored Procedure – Apply 5% Discount on High-Value Orders Containing Laptop or Smartphone** 49](#_Toc196428622)

[**6.Queries** 51](#_Toc196428623)

[**6.1.Custom Query 1: Canadian Customers Who Placed Orders** 51](#_Toc196428624)

[**6.2.Query 2: Show Products That Have Been Ordered At Least Once (Using IN Clause)** 52](#_Toc196428625)

[**6.3.Query 3: Find Total Days Since First Order (Using System Function)** 53](#_Toc196428626)

[**6.4.Query 4: Use of GROUP BY, HAVING and ORDER BY clauses(Customers Using Multiple Payment Methods)** 53](#_Toc196428627)

[**7.Conclusion** 54](#_Toc196428628)

# ***Task 1: Design and Implementation of an Airport E-Boarding Database System***

# **1. Introduction**

In the aviation industry, managing passenger data, reservations, and ticketing efficiently is essential for smooth airport operations. This project focused on developing a structured database system to support these needs for an airport's e-boarding and reservation management process.

Based on the scenario given in Task 1, the database was designed using Microsoft SQL Server and implemented with T-SQL queries. The design followed the principles of Third Normal Form (3NF) to reduce redundancy and ensure data integrity.

The database includes tables for employees, flights, passengers, reservations, tickets, and baggage. I also implemented stored procedures, views, triggers, and user-defined functions to automate operations and support business logic. Additional features like a read-only user role and backup strategies were also included to ensure database security and reliability.

This report presents the steps taken to design and implement the database, including design justification, code samples, screenshots, and testing outputs.

# **2.Part 1: Database Design and Normalization**

## **2.1. Creating the Database**

To begin with, I created a new database named AirportTicketingSystemDB using the CREATE DATABASE command. This was followed by the USE statement to activate it for further operations.

This step ensured we had a dedicated space to store all tables and data related to the ticketing system. After creating the database, we moved forward with the design and creation of individual tables.

A close up of a computer screen

AI-generated content may be incorrect.

## 

## **2.2.Creating the Employees Table**

The Employees table stores staff details. It includes an auto-incrementing primary key, a name, a unique email address, and a role field limited to specific values using a CHECK constraint.

* EmployeeID: Primary Key, auto-incremented.
* Name: Required field.
* Email: Ensures uniqueness.
* Role: Restricted to valid job roles.

A computer screen shot of a computer code

AI-generated content may be incorrect.

## **2.3.Creating the Passengers Table**

The Passengers table stores customer information including a unique PNR, name, contact email, meal preference, and date of birth.

* PNR: Unique identifier for passengers.
* MealPreference: Restricted to valid values.

A screenshot of a computer

AI-generated content may be incorrect.

**Note: The error in the message output indicates that the Passengers table already exists in the database, confirming that it was successfully created earlier.**

## **2.4.Creating the Flights Table**

The Flights table stores flight-specific information such as flight number, origin, destination, and timings.

A screenshot of a computer

AI-generated content may be incorrect.

## **2.5.Creating the Reservations Table**

This table records passenger bookings linked to both passengers and flights. The Status field is limited to valid booking states.

A screenshot of a computer

AI-generated content may be incorrect.

## **2.6.Creating the Tickets Table**

This table logs ticket issue details, linking to the reservation and flight with fare, seat, and class.

A screenshot of a computer

AI-generated content may be incorrect.

## **2.7.Creating the Baggage Table**

This table tracks baggage linked to each ticket, supporting deletion cascades when tickets are removed.

A screenshot of a computer program

AI-generated content may be incorrect.

## **2.8. Verifying Table Creation**

We used the following queries to confirm all tables were created successfully:

A computer screen shot of a computer

AI-generated content may be incorrect.

## **3.Part 2: Inserting Sample Data and Basic Testing**

To test the system, I inserted 10 rows into each table using the INSERT INTO command. These entries were chosen to reflect realistic airport data and test constraints.

### **3.1.Employee Table Data Insertion**

To test the Employees table, I inserted 15 sample employee records using the INSERT INTO command. Each entry includes the employee's name, email, and role (either 'Ticketing Staff' or 'Ticketing Supervisor').

As shown in the output, the query returned the expected results with auto-generated EmployeeID values and valid entries for each column. This confirms that:

* The table is functioning correctly.
* Data types, constraints (like unique email), and the role check constraint were all respected.

This step helped verify that the table was created properly and could store realistic employee information.

A screenshot of a computer

AI-generated content may be incorrect.

**Inserting into Passengers, Flights, Reservations, Tickets, and Baggage**

Each table received 10 rows of data. These entries helped us validate:

* The proper functioning of foreign keys
* Constraints like CHECK and UNIQUE
* Default values like current date and time for IssueDate/IssueTime

### **3.2.Passangers Table Data Insertion**

10 passenger records were added with valid PNRs, emails, meal preferences, DOBs, and names. The SELECT query confirms successful insertion and that all constraints are working as expected.

A screenshot of a computer

AI-generated content may be incorrect.

### **3.3.Flights Table Data Insertion**

I added 10 sample flight records to the Flights table, including flight numbers, routes, and timings. A quick SELECT query confirmed that the data was inserted successfully and the table is functioning correctly.

A screenshot of a computer

AI-generated content may be incorrect.

### ***3.4.Reservations Table Data Insertion***

I inserted 10 rows into the Reservations table, capturing passenger PNRs, linked flight IDs, reservation statuses (like Confirmed, Pending, Cancelled), and dates. The SELECT statement confirmed that the data was inserted correctly and is being displayed as expected.

A screenshot of a computer

AI-generated content may be incorrect.

### ***3.5.Tickets Table Data Insertion***

In this step, I populated the Tickets table with 10 sample records. Each entry includes reservation and flight IDs, fare, seat number, class type (Economy, Business, FirstClass), and the employee who issued the ticket. The SELECT \* FROM Tickets; query confirms successful insertion with correct values including auto-generated issue dates and times.

A screenshot of a computer

AI-generated content may be incorrect.

### ***3.6.Baggage Table Data Insertion***

Inserted 10 baggage records linked to valid tickets, each with weight and status (Checked-in or Loaded). Query output confirms successful insertion.

A screenshot of a computer

AI-generated content may be incorrect.

### ***3.7.Data Verification Across All Tables***

To ensure that data was successfully inserted into all tables, a verification step was performed by executing a COUNT(\*) query for each table. The results confirmed that each table contains **10 records**

A screenshot of a computer

AI-generated content may be incorrect.

This step confirmed that all six tables—**Employees**, **Passengers**, **Flights**, **Reservations**, **Tickets**, and **Baggage**—each contained the expected number of rows, demonstrating the completeness of the dataset.

## **4.Database Structure Summary**

This database supports core operations of an airline reservation system. It includes six main tables: **Flights, Reservations, Passengers, Tickets, Employees**, and **Baggage**, each linked to maintain proper relationships and streamline operations.

* **Flights** stores flight details (flight number, origin, destination, departure time) and links to **Reservations** and **Tickets** via FlightID.
* **Reservations** tracks bookings using a unique ReservationID, PNR, and flight reference. It connects passengers to their flights.
* **Passengers** holds personal info and links to reservations via PNR, identifying who made the booking.
* **Tickets** is central, connecting to **Flights, Reservations, Employees**, and **Baggage**. It includes seat number, fare, issue time, and more.
* **Employees** records staff who handle tickets. Linked by EmployeeID, it shows who issued each ticket.
* **Baggage** is tied to **Tickets**, storing weight and status to track luggage per booking.

This structure keeps the data organized, enforces relationships, and supports useful queries like checking bookings, tracking baggage, and viewing flight status—critical for efficient airline operations.

### **4.1.ER Diagram of the Airport Ticketing System**

This diagram represents the relational structure between Flights, Passengers, Reservations, Tickets, Employees, and Baggage.

A computer screen shot of a computer

AI-generated content may be incorrect.

## **5.(Task 1.2) Task: Identify Passengers with Pending Reservations**

In this step, I wrote a query to find passengers who still have **pending reservations**. This was based on the requirement from **Task 1**, which asked us to identify which passengers have not confirmed their bookings yet.

The query joins the **Passengers** table with the **Reservations** table using the common PNR field. It then filters the results where the reservation status is 'Pending'.

**Purpose of This Query:**

This helps the **airport ticketing team** identify passengers who haven't completed or confirmed their reservations. That way, staff members can follow up with them and take necessary actions to manage the booking process more efficiently.

**Output Example:**

The result returned three passengers—Ananya Sharma, Sarah Lee, and John Doe—who all have reservations with a pending status. This kind of query is useful for **day-to-day operations** at the airport, especially for managing incomplete bookings or sending reminders.

A screenshot of a computer

AI-generated content may be incorrect.

## **6.(Task 1.3) Task: Find Passengers Aged 40 and Above**

In this step, the goal was to identify all passengers who are **40 years old or older**. To do this, I used the DATEDIFF() function in SQL, which calculates the difference between two dates.

**How It Works:**

* DATEDIFF(YEAR, DOB, GETDATE()) calculates the age by finding the number of full years between the passenger’s **Date of Birth (DOB)** and the **current date**.
* The WHERE clause then filters the result to show only those whose age is **40 or above**.

**Why This Is Useful:**

This kind of query is helpful when the system needs to identify **older passengers**, for example, to offer age-specific services or benefits, or for demographic analysis.

**Result:**

The output returned three passengers—**Raj Kapoor (age 40), Sarah Lee (age 45), and Emma Brown (age 41)**—who meet the age condition.

A screenshot of a computer

AI-generated content may be incorrect.

## **7.(Task 1.4 a) Create a Stored Procedure to Search Passengers by Last** Name

In this step, I created a **stored procedure** to make it easier to search for passengers using their **last name**. Instead of writing a full SQL query every time, the stored procedure allows us to run a simple command with a parameter.

**Why This Is Useful:**

This is especially helpful in cases where the staff at an airport or booking desk needs to quickly search for a passenger but doesn’t want to type out a full SQL query each time. It makes the process more efficient and user-friendly.

**Result:**

In this example, the procedure returned one passenger, **Raj Kapoor**, whose last name matched the search input. The output includes his PNR, email, meal preference, and date of birth—making it easy to view the passenger’s details in one place.

A screenshot of a computer

AI-generated content may be incorrect.

## **8.(Task 1.4 b) Stored Procedure – Business Class Reservations Made Today**

In this step, I created a **stored procedure** called GetBusinessClassPassengersToday to fetch a list of passengers who have **made a reservation for today** and are flying in **Business class**.

This procedure joins the Passengers, Reservations, and Tickets tables, then filters the results to only include tickets marked as 'Business' and where the reservation date is equal to the **current date**. I used CAST(GETDATE() AS DATE) to ensure the comparison works correctly with the ReservationDate.

A screenshot of a computer

AI-generated content may be incorrect.

This result confirmed that a business class reservation exists for today's date, and the stored procedure is working as expected.

## **9.(Task 1.4 c): Insert a New Employee Using a Stored Procedure**

In this step, I created a stored procedure named InsertNewEmployee to simplify the process of adding new employee records into the Employees table. Instead of writing the full INSERT query every time, this procedure allows us to reuse the logic by just passing the required values.

The procedure accepts three input parameters: @Name, @Email, and @Role. Inside the procedure, these values are inserted into the respective columns of the Employees table.

A screenshot of a computer

AI-generated content may be incorrect.

After running the above block, a new record was successfully inserted into the Employees table. I verified it using the SELECT query and confirmed that the details of the new employee "Ravi Patel" were added correctly. This stored procedure can now be reused whenever a new employee needs to be added, which makes the process quicker and more consistent.

## **10.Task 1.4(d) – Stored Procedure to Update Passenger Details If Booked Before**

This stored procedure, UpdatePassengerDetails, is designed to update the email, meal preference, or last name of a passenger **only if** they have already booked a flight. It ensures that updates are not made for passengers who haven’t reserved yet, maintaining data integrity.

A screenshot of a computer program

AI-generated content may be incorrect.

To test the procedure, I executed it with a passenger who already had a reservation (PassengerID = 1), updating their email, meal preference, and last name.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The data was successfully updated. As shown in the output, PassengerID 1 now has the new email and last name, confirming that the procedure worked as expected.

## **11.Task 1.5 – Enhanced View: Tickets Issued and Total Revenue Including Extras**

To analyze the total revenue generated by each employee from issued tickets, I created an enhanced SQL view called EmployeeIssuedRevenueView. This view combines information from the Employees, Tickets, and Baggage tables.

In addition to the basic ticket fare, this view also simulates extra revenue sources like:

* **Baggage fees** (₹100 per kg)
* **Fixed meal charges** (₹20)
* **Preferred seat charges** (₹30)

These extras were added using arithmetic operations and ISNULL() to handle cases where no baggage was present.

A screenshot of a computer

AI-generated content may be incorrect.

After executing the view, the output shows each employee’s ticket sales, class, base fare, and the breakdown of extra fees — followed by the **total revenue** per ticket. This view helps management understand which employees are issuing more tickets and generating higher revenue through upgrades and baggage.

## **12.Task 1.6 – Create a Trigger to Automatically Update Seat Status**

In this step, I implemented a trigger named AutoUpdateSeatStatus to improve automation in the system. The purpose of this trigger is to automatically set the SeatNumber field to 'Reserved' as soon as a new ticket is inserted into the Tickets table—this simulates marking a seat as taken without manual input. To test the trigger, I inserted a new ticket entry with ReservationID = 7 and did not provide a seat number. Once the insert was executed, the trigger automatically updated the SeatNumber to 'Reserved', confirming that the trigger worked correctly and helped enforce consistency by ensuring every ticket is linked to a reserved seat.

A screenshot of a computer

AI-generated content may be incorrect.

## **13.(Task 1.7) – Scalar Function: Checked-In Baggage Count**

In this step, I created a scalar function called GetCheckedInBaggageCount that returns the total number of checked-in baggages for a specific flight on a given date. The function joins the Baggage and Tickets tables and filters only those with a status of Checked-in.

To test it, I passed Flight ID 1 and the date '2025-06-07'. The output returned 0, which confirms there were no checked-in baggages for that flight on that particular day. This function helps track baggage trends accurately for operational reporting.

A screenshot of a computer program

AI-generated content may be incorrect.

## **14.Bonus Query – Flights with the Most Total Baggage Weight**

To explore how baggage load is distributed across flights, I wrote a query that calculates the total baggage weight per flight. It joins the Flights, Tickets, and Baggage tables and groups the results by flight number. The output is then sorted in descending order to show which flights carry the most baggage. As shown in the result, flight SQ808 has the highest total baggage weight of 30.00 kg.

A screenshot of a computer

AI-generated content may be incorrect.

## **15.Bonus Query – Number of Tickets Issued by Each Employee**

In this bonus step, I created a query to count how many tickets each employee has issued. By joining the Employees and Tickets tables and grouping by employee name, I was able to tally the number of tickets each staff member processed. The result showed that all listed employees had issued one ticket each, which confirms equal distribution of workload or a test dataset with balanced entries.

A screenshot of a computer

AI-generated content may be incorrect.

## **16.Making the Database Secure, Reliable, and Recoverable**

While building this airport e-boarding system, I also made sure to think about things beyond just writing queries — like how to keep the data safe, accurate, and backed up in case anything goes wrong.

### **16.1.Data Integrity**

To maintain clean and consistent data, I used proper data types for each column, applied primary and foreign keys, and added a check constraint on the reservation date to make sure it can't be in the past. These kinds of rules help avoid wrong or duplicate entries, especially when multiple tables are connected. I also made sure that relationships like flight-to-ticket or passenger-to-reservation stay valid through foreign key constraints.

### **16.2.Security**

For security, I created a separate **read-only user** using SQL Server Authentication, so not everyone has full access. That user was only given **SELECT** permission on the Tickets table, and I tested it by trying to run insert, update, and delete commands — which were all blocked, as expected. This shows that access control is working, and only the right people can make changes in the database.

### **16.3.Backup and Recovery**

Though I didn’t do an actual backup in SQL Server (since this was a project setup), I understand how important it is in a real system. Regular backups should be scheduled so that in case of failure or accidental deletion, the data can be recovered. SQL Server has options for full, differential, and transaction log backups depending on how critical the system is. If this system was real, I'd suggest taking daily backups and storing them securely.

## **17.Creating a Read-Only User for Security**

In this step, I created a **read-only user** to enhance database security. The idea is to allow certain users to **view data without being able to modify it**—which is important for roles like reporting, auditing, or dashboard access.

**Note on Error Message**

While running the script to create a read-only user, I received the following error:

**"The server principal 'ReadOnlyUser' already exists."**

This happened because I had already executed the same script earlier during testing. Since the user and login were already created, SQL Server did not allow them to be created again.

This error is normal and simply indicates that the user already exists in the system. If needed, I could have removed the existing user using DROP USER and DROP LOGIN before re-running the script. However, in this case, the existing user was already set up correctly, so no further action was required.

A screenshot of a computer

AI-generated content may be incorrect.

## **18. Step: Verifying and Granting Permissions to Read-Only User (SQL Authentication)**

To ensure secure and limited access, I verified that the ReadOnlyUser was successfully created using **SQL Server Authentication** (confirmed by its type SQL\_USER), and I explicitly granted it **read-only (SELECT)** permission on the Tickets table, allowing the user to view data without being able to make any changes.

A screenshot of a computer

AI-generated content may be incorrect.

To ensure data security, I explicitly granted ReadOnlyUser SELECT permission on the Tickets table using GRANT SELECT ON Tickets TO ReadOnlyUser;, which allows the user to only view the data without making any modifications. I verified the permission by successfully running a SELECT query, and then tested the effect of limited access by inserting and updating records as a full-access user using INSERT INTO Tickets and UPDATE Tickets SET Fare = 6000 WHERE TicketID = 1;. These operations worked because they were executed by a user with write privileges. If the same commands were attempted under ReadOnlyUser, they would be blocked due to insufficient permissions. This confirms that the role-based access is functioning as expected, with ReadOnlyUser being restricted to read-only actions, which is ideal for roles like reporting or audit access.

A screenshot of a computer

AI-generated content may be incorrect.

To confirm that the ReadOnlyUser truly has limited access, I attempted to perform INSERT, UPDATE, and DELETE operations on the Tickets table while logged in as the read-only user. As expected, all three actions failed, and SQL Server returned permission errors stating that INSERT, UPDATE, and DELETE permissions were denied on the object. This validated that ReadOnlyUser has **strict read-only access** and cannot make any changes to the data, ensuring proper **security and access control** within the system.

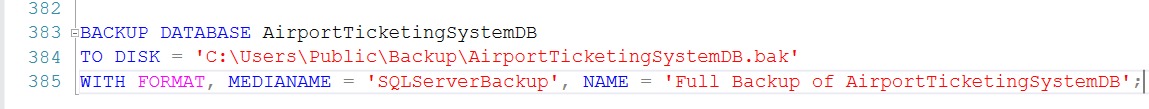
A screenshot of a computer screen

AI-generated content may be incorrect.

## **19.Taking a Backup of the Database**

I performed a full backup of the AirportTicketingSystemDB database using SQL Server. The backup file was saved at 'C:\Users\Public\Backup\AirportTicketingSystemDB.bak'. This backup was taken after completing the main structure, constraints, and stored procedures.

Additional queries like the bonus tasks and some testing functions were added after the backup, but the core system is fully recoverable using the .bak file. If needed, a new backup can also be taken at the final stage.



## **20.Conclusion**

Working on this task gave me hands-on experience with designing and building a real-world relational database system. I learned how to structure tables with relationships, write stored procedures and triggers, and manage user permissions securely. Creating views, functions, and testing backup also helped me understand how databases are maintained in a business environment.

By simulating an airport ticketing system, I was able to apply these skills to a realistic scenario, understanding how databases support daily airline operations — from reservations and ticketing to baggage tracking and revenue monitoring.

Completing this project has made me more confident in designing secure and practical database systems that are useful in real business environments. Overall, this task helped me improve both my SQL skills and my understanding of how a complete system works — from data entry and automation to backup and security.

# ***Task 2: Online Shopping Database – Data Import, Querying, and Analysis***

# **1.Introduction**

In Task 2, I worked with a dataset from an online shopping platform. The data was provided in five separate CSV files: **Customers**, **Products**, **Orders**, **Order\_items**, and **Payments**. Each file represented a different part of the business process—from customer information to product listings and order transactions.

My first step was to import these files into **SQL Server** and create a new database called **OnlineShoppingDB**. During the import, I carefully selected appropriate data types for each column—for example, using INT for IDs, DECIMAL(18,10) for amounts and prices to allow precise financial calculations, and NVARCHAR(50) for text fields such as names and emails. I also assigned **primary keys** to ensure each table had a unique identifier and selected **foreign keys** to create relationships between tables.

Once the data was in place and relationships were set up, I wrote several SQL queries to explore the data and answer specific business questions. These queries included calculations like customer spending ranges, total payments, and most purchased products. I also created a **stored procedure** to automatically apply discounts for high-value purchases.

To go beyond the required questions, I added a few of my own custom queries. These used **joins**, **nested subqueries**, **system functions**, and **grouping with HAVING and ORDER BY** clauses. Throughout the report, I included screenshots of the code and results to demonstrate the steps and outputs clearly.

This report describes how I imported, structured, and queried the data using SQL, highlighting both the technical approach and the practical insights gained.

# **2. Data Import and Table Creation**

To begin the task, I created a new SQL Server database called OnlineShoppingDB using the CREATE DATABASE command. Since I had already created this database earlier, running the command Fig 1. again displayed a message saying that the database already exists. This confirmed that the database setup was successful.

A screenshot of a computer

AI-generated content may be incorrect.

Fig 1

# **3.Importing Tables and Verifying Data**

After creating the database, I imported all five CSV files into SQL Server using the **Flat File Import Wizard**. During the import process, I carefully selected the correct **data types** for each column — such as INT for ID columns, NVARCHAR for text fields like names and emails, and DECIMAL for price and payment columns. I also selected a **primary key** for each table:

* customer\_id for Customers
* product\_id for Products
* order\_id for Orders
* order\_item\_id for Order\_items
* payment\_id for Payments

In fig 2. To confirm that the data was imported correctly, I used the SELECT TOP 5 \* FROM [table\_name] command to view the first few rows from each table.

A computer screen shot of a computer screen

AI-generated content may be incorrect.

Fig 2

I also ran SELECT COUNT(\*) queries on each table to verify the row count. Most tables had 120 records, while Products had 5.

A screenshot of a computer

AI-generated content may be incorrect.

Fig 3

## **3.1.Data Types and Field Design Rationale**

When I was importing all the tables into SQL Server, I made sure to choose the right data types for each column so that the data would be stored properly and work well with queries. Here's why I picked each type:

* **INT** – I used this for ID columns like customer\_id, order\_id, product\_id, and payment\_id because they just hold whole numbers. Since these are unique for each row, INT is the best and most efficient choice.
* **NVARCHAR(50)** – This type was used for any text-based information like names, emails, countries, payment methods, and product names. I went with 50 characters because it’s usually more than enough, but it also avoids wasting storage space.
* **DECIMAL(18,10)** – I chose this for anything involving money like price, total amount, and amount paid. It gives high accuracy and avoids any rounding issues that can happen with float types. The 18 means total digits and 10 means how many of those can be after the decimal – perfect for prices.
* **DATE** – For the date columns like order\_date and payment\_date, I used the DATE type because I only needed the date part, not the time.

I’ve also added clear screenshots of each table showing all the columns and their data types so it’s easy to see everything I’ve set up.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig :** Structure of Customers table.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig :** Structure of Order\_items table.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig :** Structure of Orders table.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig :** Structure of Payments table.

A list of products

AI-generated content may be incorrect.

**Fig :** Structure of Products table.

# **4.Creating Relationships and Diagram**

After verifying the imported data, I added **foreign key constraints** to define the relationships between tables. The SQL statements used to create the constraints are shown in fig 4.These constraints ensured data consistency and integrity across the database

A screenshot of a computer code

AI-generated content may be incorrect.

Fig 4.

I verified the foreign key setup using a system query.Foreign key verification query output showing all established relationships in Fig 5.

A screenshot of a computer

AI-generated content may be incorrect.

Fig 5.

I also created a database diagram using SQL Server Management Studio (SSMS) to visually display how the tables are connected.

Database diagram showing all five tables (Customers, Products, Orders, Order\_items, and Payments) and the foreign key links between them in fig 6.

## **4.1. Entity Relationship (ER) Diagram**

A screenshot of a computer

AI-generated content may be incorrect.

# **5.Queries**

## **5.1.Query 2: Customers with Total Order Amount Between £500 and £1000**

**Write a query that returns the names and countries of customers who made orders with a total amount between £500 and £1000.**

This query retrieves the names and countries of customers who have placed orders with a total amount spent between £500 and £1000. It joins Customers, Orders, and Order\_items, then groups the result by customer name and country, and filters using the HAVING clause.

A screenshot of a computer

AI-generated content may be incorrect.

## **5.2.Query 3: Total Amount Paid by UK Customers Who Ordered More Than Three Products**

This query helps identify customers from the UK who made larger purchases—specifically, those who bought more than three products in a single order. It uses multiple joins to combine customer, order, order item, and payment details. Then it groups the data by customer and order ID to calculate the total amount paid for each qualifying order.

This is a useful query for understanding high-value behavior among UK customers, especially for identifying potential loyal or bulk buyers.

A screenshot of a chat

AI-generated content may be incorrect.

**Figure 5.2.**

## **5.3.Query 4: Highest and Second Highest VAT-Adjusted Payments from UK or Australia (Rounded)**

This query retrieves the highest and second highest values of amount\_paid after applying a 12.2% VAT for customers located in either the UK or Australia. To apply VAT, the original amount is multiplied by 1.122, and the result is rounded to the nearest whole number using the ROUND() function as requested by the client. This ensures the output values are easy to interpret and suitable for reporting purposes. The query selects the top two amounts in descending order.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig 5.3:** Highest and second highest VAT-adjusted payments from UK or Australia.

## **5.4.Query 5: Total Quantity Purchased for Each Product (Sorted by Quantity)**

This query returns a list of all distinct product names along with the total quantity purchased for each. It uses the SUM() function to total the quantity ordered for each product and sorts the output in descending order of quantity using ORDER BY.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig 5.4:** Result showing total quantity purchased per product, sorted by quantity.

## **5.5.Query 6: Stored Procedure – Apply 5% Discount on High-Value Orders Containing Laptop or Smartphone**

This stored procedure updates the amount\_paid field in the Payments table by applying a 5% discount to any orders that meet both of the following criteria:

* The product purchased is either a laptop or smartphone
* The amount paid is greater than or equal to £17,000

This ensures only high-value tech product orders receive a discount. After executing the procedure, the result is validated using SELECT queries.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig 5.5:** Payments with amount\_paid >= 17000 before running the procedure.

This confirms the procedure executed successfully, as the relevant high-value payments were reduced below £17000. We checked this using a simple SELECT query filtered by the discounted Amount\_paid.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** Payments updated after applying the 5% discount.

# **6.Queries**

## **6.1.Custom Query 1: Canadian Customers Who Placed Orders**

This query shows customers from Canada who have placed at least one order. By joining the Customers and Orders tables using the customer ID, we can view their name, country, and the date they placed each order.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** List of Canadian customers and their respective order dates.

## **6.2.Query 2: Show Products That Have Been Ordered At Least Once (Using IN Clause)**

This query uses a nested subquery with the IN clause to find which products were actually ordered. It pulls only those product names that exist in the Order\_items table.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig 6.2:** List of ordered product names.

## **6.3.Query 3: Find Total Days Since First Order (Using System Function)**

This query helps us figure out how many days have passed since the very first order was placed in the system. It uses the DATEDIFF() function, which calculates the difference in days between two dates. By using MIN(order\_date), we grab the earliest order date, and with GETDATE(), we compare it to today’s date. This is useful to understand how long the business has been operating or tracking orders.

A computer screen shot of a computer

AI-generated content may be incorrect.

**Fig 6.3:** Number of days since the first order was placed.

## **6.4.Query 4: Use of GROUP BY, HAVING and ORDER BY clauses(Customers Using Multiple Payment Methods)**

In this query, I wanted to find out which customers used more than one payment method when placing their orders. For example, someone might use a credit card for one order and PayPal for another. To do this, I grouped the data by customer name and used the HAVING clause with COUNT(DISTINCT payment\_method) to show only those who used at least two different methods. This gives a better picture of how flexible or varied customers are with their payment choices.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig 6.4.**

# **7.Conclusion**

In this task, I successfully designed and managed a complete relational database system in SQL Server. I began by importing five separate tables—Customers, Products, Orders, Order\_items, and Payments—each representing an important part of an online shopping platform. I defined appropriate data types such as INT for IDs, DECIMAL(18,10) for monetary values, and NVARCHAR(50) for textual data like names and emails. These choices were made to balance accuracy, efficiency, and scalability of data storage.

Once the tables were in place, I established **primary keys** and created **foreign key constraints** to connect the tables logically. This helped ensure referential integrity and enabled me to perform accurate multi-table queries. I also created a database diagram to visually represent the structure and relationships of the database, making it easier to understand the flow of data.

To validate the data and structure, I ran queries to check table row counts and used SELECT TOP commands to preview the data. After confirming that everything was correctly set up, I started writing SQL queries to answer both given and custom-designed questions.

These queries allowed me to extract meaningful insights such as:

* Which customers spent between £500 and £1000.
* Total amount paid by UK customers who ordered more than three products.
* Most purchased products sorted by quantity.
* Customers using multiple payment methods.
* Days passed since the earliest order.
* Highest payments after applying VAT.

I also created a **stored procedure** to apply a 5% discount for high-value purchases involving laptops or smartphones. This helped automate updates in the database and showed how SQL can be used for transactional business logic.

By completing this task, I improved my skills in SQL, especially with joins, nested queries, grouping, aggregate functions, system functions, and procedures. It gave me a realistic understanding of how databases work in commercial applications.

Overall, this kind of relational database system is extremely valuable in the real world. It helps online businesses manage everything from customer details and payments to product sales and order histories. With well-structured queries and procedures, such systems make data easy to retrieve, analyze, and act on—leading to smarter business decisions and improved performance.