



VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF MECHANICS

COURSE: DATABASE

**Book Store Management Database System:
Design, Implementation, and SQL Programming**

Author: Badshah Forhad, TDIU-23

Study Programme: BSc in Applied Artificial Intelligence

VILNIUS 2025

Table of Contents:

1. Introduction:.....	3
2. Database Design.....	4
2.1 Design Principles.....	4
2.2 List of Tables Created.....	5
2.3 Entity–Relationship Diagram.....	5
3. Implementation Details.....	6
3.1 Table Structure Example.....	7
3.2 Sample Data.....	7
4. Data Retrieval and Queries.....	8
4.1 Aggregate Queries.....	8
4.2 Pagination.....	9
4.3 Join Queries.....	9
5. SQL Programming.....	10
5.1 Stored Procedure.....	10
5.2 User-Defined Function.....	10
5.3 Trigger A trigger ReduceStock.....	10
5.4 Manual Transaction.....	11
6. Conclusion.....	11
7. Github Link.....	11

1. Introduction:

Database systems are essential tools for managing information efficiently, especially in environments where accuracy, consistency, and traceability are required. A bookstore requires properly organised data to manage authors, catalogue books, track inventory, record customer information, process orders, and log order items. The goal of this project was to develop a functional **Bookstore Database** using Microsoft SQL Server as both the database engine and query environment.

Throughout the project, I implemented the core concepts taught in the Databases course. These include designing multiple related tables, establishing primary and foreign key constraints, implementing a many-to-many relationship via a junction table (**OrderDetails**), and applying appropriate unique and non-unique indexes to optimise queries. I created tables for **Authors**, **Books**, **Customers**, **Orders**, and **OrderDetails**, selected suitable data types (e.g., **INT**, **VARCHAR**, **DECIMAL**, **DATETIME**), and used **IDENTITY** columns for auto-incremented primary keys.

I also populated the database with multi-row inserts and demonstrated data manipulation through **UPDATE**, **DELETE**, and **TRUNCATE** operations. For programmatic logic and automation, I implemented a stored procedure to retrieve customer orders, a scalar function to compute line totals, a trigger to adjust stock after new order lines are inserted, and manual transactions (**BEGIN/COMMIT/ROLLBACK**) to ensure atomic order creation. To support reporting and analysis, I wrote multiple **SELECT** queries that use aggregate functions (**SUM**, **COUNT**, **AVG**), **GROUP BY**, **ORDER BY**, and pagination (**OFFSET / FETCH**), and I created a view to present combined order summaries.

The final result is a well-organised Bookstore database with a clear schema, enforced referential integrity, practical automation, and a comprehensive set of SQL queries demonstrating the techniques learned in the course. This system is suitable for extension (e.g., payments, promotions, reviews) and provides a solid foundation for presenting how relational design and SQL programming support real-world retail workflows.

2. Database Design

2.1 Design Principles

The bookstore database was designed using standard relational modelling principles to ensure accuracy, scalability, and consistency across all stored information.

Key design principles applied include:

- Each table uses a primary key generated automatically with `IDENTITY(1, 1)` for unique row identification.
- Foreign key constraints maintain referential integrity between related tables such as *Books–Authors*, *Orders–Customers*, and *OrderDetails–Orders/Books*.
- A many-to-many relationship between *Orders* and *Books* is implemented using a junction table (`OrderDetails`), which stores quantities for each book in an order.
- Appropriate data types such as `INT`, `VARCHAR`, `DECIMAL`, and `DATETIME` were selected based on the nature of the data.
- Unique constraints were used on fields like `Customers.Email` to prevent duplicate records.
- Indexes were created, such as an index on `Orders.CustomerID`, to improve search performance and speed up join operations.

2.2 List of Tables Created

The bookstore system consists of the following tables:

- **Authors** – Stores author profiles including name and optional biography.
- **Books** – Contains information about each book such as title, genre, price, stock level, and its related author.
- **Customers** – Holds customer details including name, email, and address.
- **Orders** – Records customer orders along with order date and reference to the customer who placed the order.
- **OrderDetails** – A junction table linking *Orders* and *Books*, storing quantities for each book included in an order.

2.3 Entity–Relationship Diagram

The Entity–Relationship structure illustrates how all tables in the bookstore system are connected.

It shows the primary keys for each table and foreign key relationships such as:

- Authors → Books (one-to-many)
- Customers → Orders (one-to-many)
- Orders → OrderDetails (one-to-many)
- Books → OrderDetails (one-to-many)

This diagram demonstrates how orders, books, and customers interact within the database to support core operations like sales, inventory tracking, and reporting.

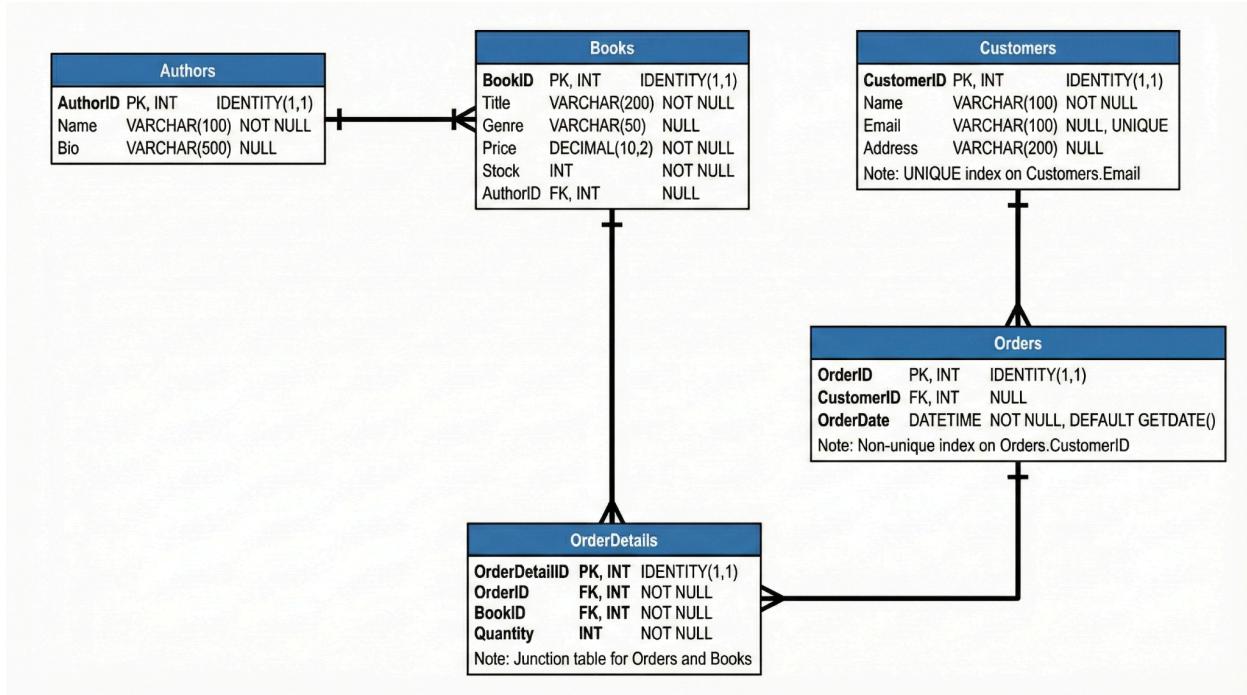


Figure 1: Entity–Relationship Diagram

3. Implementation Details

This section explains how the database was implemented, including the structure of the tables and the choices made during development.

3.1 Table Structure Example

The following table example illustrates the typical structure used in the database. All tables were created using appropriate data types, primary keys, foreign keys, and constraints to maintain data integrity.

Example Table: Books

The **Books** table stores all book-related information available in the bookstore. It includes details such as title, price, publication year, genre, and the author associated with the book.

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	BookID	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	Title	varchar	no	200			no	no	no	SQL_Latin1_General_CI_AS
3	Genre	varchar	no	50			yes	no	yes	SQL_Latin1_General_CI_AS
4	Price	decimal	no	9	10	2	no	(n/a)	(n/a)	NULL
5	Stock	int	no	4	10	0	no	(n/a)	(n/a)	NULL
6	AuthorID	int	no	4	10	0	yes	(n/a)	(n/a)	NULL

Figure 2: Structure of Books Table

3.2 Sample Data

All tables were populated with sample data using multi-row INSERT statements. The data was designed to resemble a realistic bookstore environment, with multiple books, authors, customers, and orders. Figure 3 shows part of the data inserted into the Books table, demonstrating different genres, prices, stock levels, and links to authors.

	BookID	Title	Genre	Price	Stock	AuthorID
1	1	Harry Potter and the Philosopher's Stone	Fantasy	20.99	44	1
2	2	A Game of Thrones	Fantasy	25.50	38	2
3	3	Murder on the Orient Express	Mystery	15.00	40	3
4	33	Murder on the Orient Express	Mystery	15.00	40	3
5	55	Harry Potter and the Philosopher's Stone	Fantasy	20.99	55	1
6	70	A Game of Thrones	Fantasy	25.50	30	2

Figure 3: Sample Inserted Data for Books

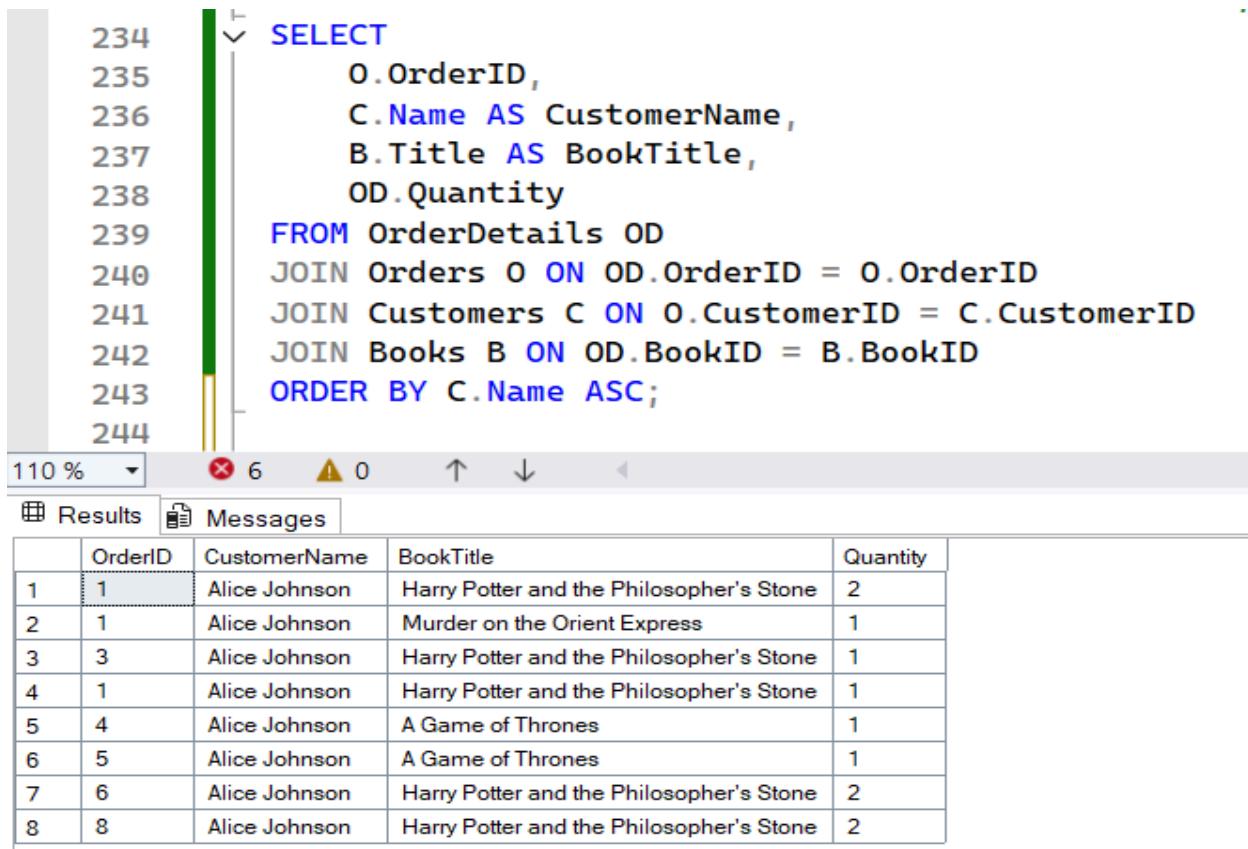
4. Data Retrieval and Queries

4.1 Aggregate Queries

To analyse bookstore activity, several aggregate queries were implemented:

- The total number of copies sold for each book using **SUM** on the Quantity field.
- The total number of orders for each book using **COUNT**.
- The average quantity of books per order using **AVG**.

These queries help demonstrate how aggregated data can provide useful insights into book sales and customer buying patterns.



The screenshot shows a SQL query being run in SSMS. The query joins four tables: OrderDetails, Orders, Customers, and Books, to retrieve the OrderID, CustomerName, BookTitle, and Quantity for each order. The results are ordered by CustomerName. The output table has columns: OrderID, CustomerName, BookTitle, and Quantity. The data shows multiple purchases of 'Harry Potter and the Philosopher's Stone' and 'A Game of Thrones' by 'Alice Johnson'.

```

234    SELECT
235        O.OrderID,
236        C.Name AS CustomerName,
237        B.Title AS BookTitle,
238        OD.Quantity
239    FROM OrderDetails OD
240    JOIN Orders O ON OD.OrderID = O.OrderID
241    JOIN Customers C ON O.CustomerID = C.CustomerID
242    JOIN Books B ON OD.BookID = B.BookID
243    ORDER BY C.Name ASC;
244

```

	OrderID	CustomerName	BookTitle	Quantity
1	1	Alice Johnson	Harry Potter and the Philosopher's Stone	2
2	1	Alice Johnson	Murder on the Orient Express	1
3	3	Alice Johnson	Harry Potter and the Philosopher's Stone	1
4	1	Alice Johnson	Harry Potter and the Philosopher's Stone	1
5	4	Alice Johnson	A Game of Thrones	1
6	5	Alice Johnson	A Game of Thrones	1
7	6	Alice Johnson	Harry Potter and the Philosopher's Stone	2
8	8	Alice Johnson	Harry Potter and the Philosopher's Stone	2

Figure 4: Output of Join Query

4.2 Pagination

Pagination was implemented using:

OFFSET 0 ROWS

```
FETCH NEXT 6 ROWS ONLY;
```

This allows selecting a limited number of rows at a time—useful for listing books or orders in pages when dealing with large datasets.

4.3 Join Queries

Join queries were used to retrieve related information across multiple tables:

- A two-table INNER JOIN connects Books with Authors to show each book and its author.
- A two-table LEFT JOIN connects Customers with Orders to list all customers along with their orders.
- A three-table JOIN connects Orders, Customers, and OrderDetails to show detailed order information including customer name, book title, and quantity ordered.

These queries demonstrate how relational databases efficiently combine information from multiple tables to generate meaningful insights.

5. SQL Programming

5.1 Stored Procedure

A stored procedure named GetCustomerOrders was created. It takes a CustomerID as input and retrieves all orders placed by that customer, including order dates, book titles, quantities, and line totals. This procedure simplifies fetching detailed order information programmatically.

5.2 User-Defined Function

A scalar function, CalculateLineTotal, was designed to compute the total price for a given order line, multiplying the book's price by the quantity ordered. This is useful for quickly calculating costs for multiple orders.

5.3 Trigger A trigger ReduceStock

After the Order was implemented on the OrderDetails table. It automatically reduces the stock of books whenever a new order detail is inserted, ensuring inventory remains accurate without manual updates.

5.4 Manual Transaction

A manual transaction block was used to demonstrate atomic operations when inserting a new order and its corresponding order details. If both inserts succeed, the transaction is committed; if an error occurs (e.g., invalid BookID), the transaction is rolled back, ensuring data consistency and avoiding partial updates.

6. Conclusion

The Bookstore Database Project applied SQL concepts in a practical environment. It manages books, authors, customers, orders, and order details with proper relationships and constraints.

The project included table creation, many-to-many relationships, data manipulation (INSERT, UPDATE, DELETE), aggregate queries, pagination, and JOIN operations. Stored procedures, functions, triggers, and manual transactions ensured automated processes and data integrity.

Overall, this project reinforced my understanding of SQL Server and relational databases in a real-world business context.

Github Link: <https://github.com/badshahforhad/bookstore.git>