

# **AgenixAI- Training**

## **Module 1**

### **SQL: Library Management System**

**Name: Bhoomika K**

**3<sup>rd</sup> Year AI&ML**

# **Library Management System: Database Design and Queries**

## **Objective**

To design, implement, and optimize a relational database for a library system.

## **Scenario**

Design a Relational Data Model for a library system that manages books, authors, customers, and book transactions. The system must ensure data integrity, handle large datasets, and support complex queries.

## **Table of Contents:**

1. Introduction
2. Schema Design and Explanation
3. Query Implementation
4. Execution Plans and Query Optimization
5. Conclusion

## **1. Introduction**

This report covers the design and implementation of a relational database for a Library Management System. It includes the creation of database tables, relationships, and SQL queries to manage books, authors, customers, and borrowings. Additionally, execution plans for optimizing queries have been included to improve database performance.

## **2. Schema Design and Explanation**

For a **Library Management System (LMS)** schema, the purpose is to manage and organize all aspects of a library's operations, such as tracking books, authors, users (borrowers), and their borrowing activities. The schema will help streamline processes like checking out and returning books, managing inventory, and keeping track of overdue books or fines.

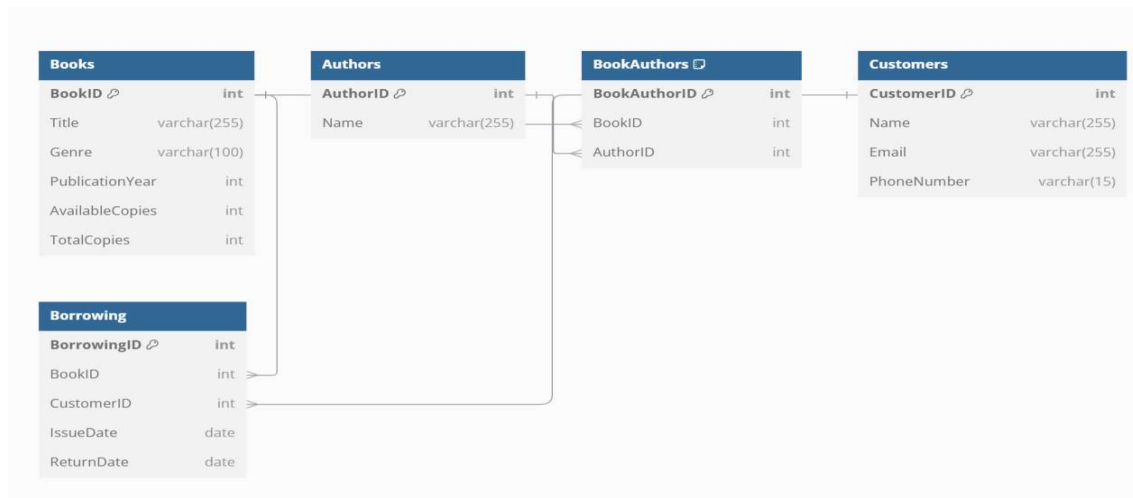
### **Description of the Library Management Schema**

The schema is designed to support the management of library resources and operations efficiently. It covers multiple key areas of functionality such as:

1. **Book Information:** The library needs to keep track of various attributes of the books, including title, author(s), genre, publication year, availability.
2. **User Information:** The system must store user details, such as the user's name, contact information.
3. **Borrowing System:** The schema needs to track which books are borrowed by which users, when they were borrowed, and when they are due for return. Additionally, it may need to record overdue books .
4. **Author Information:** Authors, their books, and the relationships between them need to be represented. A book can have one or more authors.
5. **Library Inventory Management:** The system tracks the number of copies available for each book and can handle updates when books are checked out or returned.

This schema aims to ensure the library system can function smoothly, handle large amounts of data, and provide accurate, up-to-date information on book availability and user activity. It is also designed to be flexible, allowing for easy scalability as the library grows in size and the need for additional functionality arises.

## Schema Design:



**Schema design** is the process of creating a logical and organized structure for a database. It defines how data is stored, organized, and accessed, ensuring efficient data management and retrieval. The schema serves as a blueprint for the database, outlining the relationships between tables, attributes, and constraints. Proper schema design is critical for maintaining data consistency, integrity, and scalability.

### Key Components of Schema Design:

1. **Tables (Entities):**

Represent objects or concepts in the system (e.g., Students, Courses).

2. **Columns (Attributes):**

Define the properties of an entity (e.g., StudentName, CourseID).

3. **Primary Key:**

A unique identifier for each record in a table (e.g., StudentID).

4. **Foreign Key:**

A reference to the primary key of another table, establishing relationships between tables.

5. **Relationships:**

Define how tables are connected (e.g., one-to-one, one-to-many, many-to-many).

6. **Constraints:**

Rules enforced to maintain data integrity, such as:

- **NOT NULL:** Ensures values are provided.

- **UNIQUE:** Prevents duplicate values.
- **CHECK:** Enforces specific conditions.
- **DEFAULT:** Provides default values.

## Entities and Attributes:

### 1. Books Table

Column Name	Data Type	Description
BookID	SERIAL	Primary key, uniquely identifies a book.
Title	VARCHAR(255)	Title of the book. Cannot be null.
Genre	VARCHAR(100)	Genre of the book. Cannot be null.
PublicationYear	INT	Year the book was published. Cannot be null.
AvailableCopies	INT	Number of available copies for borrowing. Cannot be less than 0.
TotalCopies	INT	Total number of copies of the book. Cannot be less than 0.

#### Explanation:

The Books table stores key information about books in the library, including their title, genre, publication year, and the number of available copies for borrowing. The BookID is the unique identifier for each book, ensuring no two books have the same ID.

## 2. Authors Table

Column Name	Data Type	Description
AuthorID	SERIAL	Primary key, uniquely identifies an author.
Name	VARCHAR(255)	Name of the author. Cannot be null.

### Explanation:

The `Authors` table stores information about the authors of books in the library. The `AuthorID` serves as a unique identifier for each author, allowing the system to link authors to books they've written.

## 3. BookAuthors Table (Many-to-Many Relationship)

Column Name	Data Type	Description
BookAuthorID	SERIAL	Primary key for the table.
BookID	INT	Foreign key referencing <code>Books(BookID)</code> . Deletes related entries when a book is deleted.
AuthorID	INT	Foreign key referencing <code>Authors(AuthorID)</code> . Deletes related entries when an author is deleted.

### Explanation:

The `BookAuthors` table handles the many-to-many relationship between books and authors. A book can have multiple authors, and an author can write multiple books. This table ensures the efficient storage and retrieval of such relationships.



#### 4. Customers Table

Column Name	Data Type	Description
CustomerID	SERIAL	Primary key, uniquely identifies a customer.
Name	VARCHAR (255)	Name of the customer. Cannot be null.
Email	VARCHAR (255)	Customer's email address.

#### Explanation:

The Customers table contains the details of all library customers. The CustomerID is the unique identifier, and each customer has a Name and Email address. This table is used to manage customer information and is referenced by the Borrowing table to track borrowing activities.

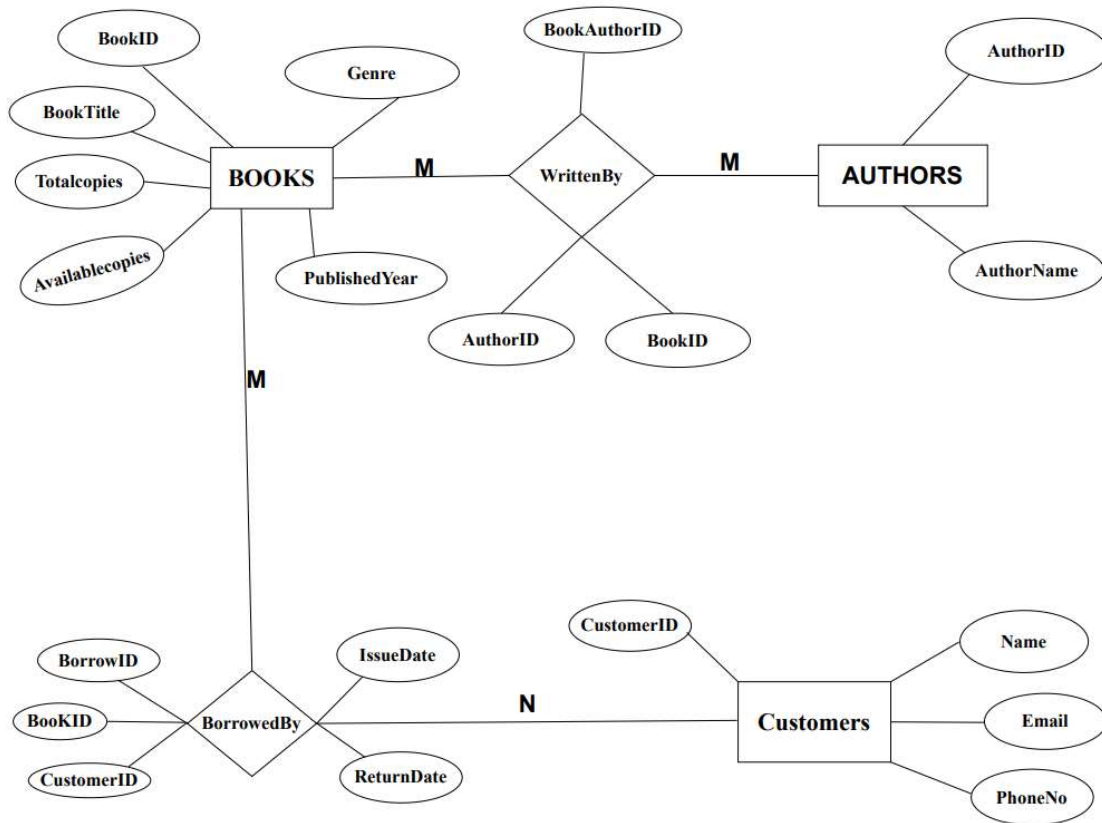
#### 5. Borrowing Table (Many-to-One Relationship)

Column Name	Data Type	Description
BorrowID	SERIAL	Primary key, uniquely identifies a borrowing transaction.
BookID	INT	Foreign key referencing Books (BookID) .
CustomerID	INT	Foreign key referencing Customers (CustomerID) .
BorrowDate	DATE	Date the book was borrowed.
ReturnDate	DATE	Date the book was returned.

#### Explanation:

The Borrowing table links books with customers, keeping track of the borrowing date and return date for each transaction. The BorrowID is the unique identifier for each borrowing record, and the table uses foreign keys to reference the Books and Customers tables.

## Diagram: Entity-Relationship Diagram



## Entity-Relationship Diagram Explanation for Library Management System

The Entity-Relationship Diagram (ERD) shown represents the database design for a **Library Management System**. This design efficiently organizes and manages the relationships between books, authors, customers, and borrowing transactions while adhering to normalization rules to ensure data integrity and scalability. Below is a detailed explanation of the key components and relationships:

## Entities and Their Attributes

### 1. Books

The Books entity captures essential details about each book in the library.

- **Attributes:**

- BookID: Unique identifier for each book.
- BookTitle: Title of the book.
- TotalCopies: Total number of copies available in the library.
- AvailableCopies: Number of copies currently available for borrowing.
- Genre: Category of the book (e.g., fiction, non-fiction).
- PublishedYear: The year the book was published.

### 2. Authors

The Authors entity stores information about authors who have written the books in the library.

- **Attributes:**

- AuthorID: Unique identifier for each author.
- AuthorName: Name of the author.

### 3. Customers

The Customers entity maintains the information of all library members.

- **Attributes:**

- CustomerID: Unique identifier for each customer.
- Name: Name of the customer.
- Email: Email address of the customer.
- PhoneNo: Contact number of the customer.

### 4. BorrowedBy

The BorrowedBy entity tracks borrowing transactions, linking books to customers.

- **Attributes:**

- BorrowID: Unique identifier for each borrowing transaction.
- BookID: Foreign key referencing the borrowed book.
- CustomerID: Foreign key referencing the borrowing customer.
- IssueDate: Date the book was borrowed.
- ReturnDate: Date the book is expected to be or has been returned.

## 5. **BookAuthors**

The BookAuthors entity resolves the many-to-many (M:N) relationship between books and authors.

- **Attributes:**
  - BookAuthorID: Unique identifier for the relationship.
  - BookID: Foreign key referencing the book.
  - AuthorID: Foreign key referencing the author.

## **Relationships and Normalization**

### 1. **Books ↔ Authors** (Many-to-Many)

A book can be written by multiple authors, and an author can write multiple books.

This M:N relationship is normalized by introducing the BookAuthors table as a junction table. This ensures:

- Elimination of data redundancy.
- Simplified querying of books and authors.

### 2. **Books ↔ BorrowedBy ↔ Customers** (Many-to-Many)

A customer can borrow multiple books, and each book can be borrowed by multiple customers over time. This M:N relationship is normalized by introducing the BorrowedBy table. It tracks each borrowing transaction uniquely, ensuring accurate and consistent data representation.

### 3. **Customers ↔ BorrowedBy** (One-to-Many)

A customer can have multiple borrowing transactions over time. This relationship is implemented using the CustomerID foreign key in the BorrowedBy table.

### 4. **Books ↔ BorrowedBy** (One-to-Many)

Each borrowing transaction is linked to a single book through the BookID foreign key.

## Design Benefits

1. **Normalization:** By introducing the BookAuthors and BorrowedBy tables, the design breaks M:N relationships into two 1:M relationships, ensuring compliance with database normalization rules and eliminating redundancy.
2. **Data Integrity:** Foreign key constraints between tables ensure consistency and enforce relationships among entities. For example:
  - BookID in BorrowedBy references BookID in Books.
  - CustomerID in BorrowedBy references CustomerID in Customers.
3. **Scalability:** The database design is scalable, allowing the addition of new books, authors, and customers without structural changes.
4. **Efficient Querying:** The well-defined relationships and use of unique identifiers enhance query performance and simplify operations like searching for a book's authors or tracking customer borrowing history.
5. **Clear Relationship Representation:** By resolving M:N relationships into intermediate tables, the design ensures clarity in database relationships, making it easier to understand and maintain.

This ERD provides a robust framework for managing a library's data efficiently. It demonstrates a comprehensive approach to handling complex relationships, ensuring data normalization, scalability, and integrity. The design is adaptable for real-world implementation, making it suitable for industry use in any library management system.

### 3. Query Implementation

#### 1. Retrieve the top 5 most-issued books with their issue count

```
SELECT BookID, COUNT(*) AS IssueCount
FROM Borrowing
GROUP BY BookID
ORDER BY IssueCount DESC
LIMIT 5;
```

#### Output:



The screenshot shows a database query result interface. At the top, there are three tabs: "Data Output", "Messages", and "Notifications". Below the tabs is a toolbar with various icons for file operations, editing, and viewing. The main area displays a table with the following data:

	bookid integer	issuecount bigint
1	192	7
2	177	6
3	161	6
4	165	6
5	162	5

**Explanation:** This query retrieves the top 5 most-issued books from the library's borrowing records. The BookID represents each book's unique identifier, and COUNT(\*) calculates how many times each book has been borrowed. The GROUP BY clause groups the results by BookID, ensuring that we count the occurrences of each individual book. The ORDER BY IssueCount DESC sorts the results in descending order based on the issue count, so the most borrowed books appear at the top. Finally, the LIMIT 5 ensures that only the top 5 books are returned.

In a real-world library, this query helps the staff understand which books are in high demand and need to be stocked more frequently or highlighted for promotional purposes. It can also help the library identify the most popular genres and topics.

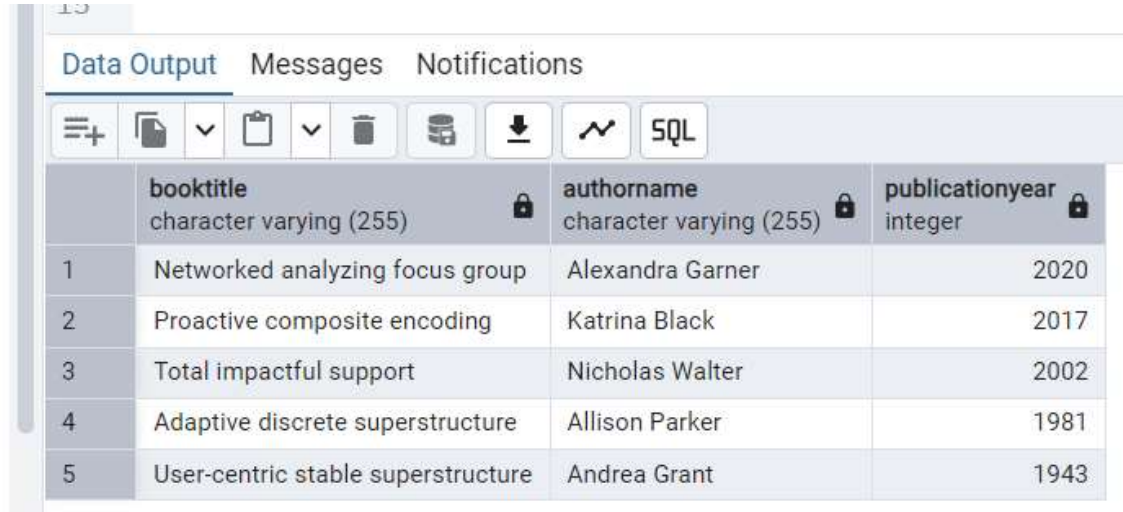
## **2. List books along with their authors that belong to the "Fantasy" genre, sorted by publication year in descending order**

```
SELECT
    Books.Title AS BookTitle,
    Authors.Name AS AuthorName,
    Books.PublicationYear AS PublicationYear
FROM
    Books
INNER JOIN
    BookAuthors ON Books.BookID = BookAuthors.BookID
INNER JOIN
    Authors ON BookAuthors.AuthorID = Authors.AuthorID
WHERE
    Books.Genre = 'Fantasy'
ORDER BY
    Books.PublicationYear DESC;
```

**Explanation:** This query lists all books from the "Fantasy" genre along with their authors, sorted by the book's publication year in descending order. It uses INNER JOIN to link the Books table with the BookAuthors table (to get the relationship between books and authors) and then with the Authors table to retrieve author names. The WHERE clause filters the results to include only books from the "Fantasy" genre. The ORDER BY Books.PublicationYear DESC ensures that the most recently published books appear first.

This query is particularly useful in libraries when curating specific genre-based recommendations or collections. For instance, if a library wants to highlight newer fantasy books, this query can help identify those quickly.

## Output:



The screenshot shows a database application interface with a tabbed view. The 'Data Output' tab is active, displaying a table with three columns: 'booktitle', 'authorname', and 'publicationyear'. The table contains five rows of data. Above the table is a toolbar with various icons for data manipulation and a 'SQL' button.

	booktitle character varying (255)	authorname character varying (255)	publicationyear integer
1	Networked analyzing focus group	Alexandra Garner	2020
2	Proactive composite encoding	Katrina Black	2017
3	Total impactful support	Nicholas Walter	2002
4	Adaptive discrete superstructure	Allison Parker	1981
5	User-centric stable superstructure	Andrea Grant	1943

### 3. Identify the top 3 customers who have borrowed the most books

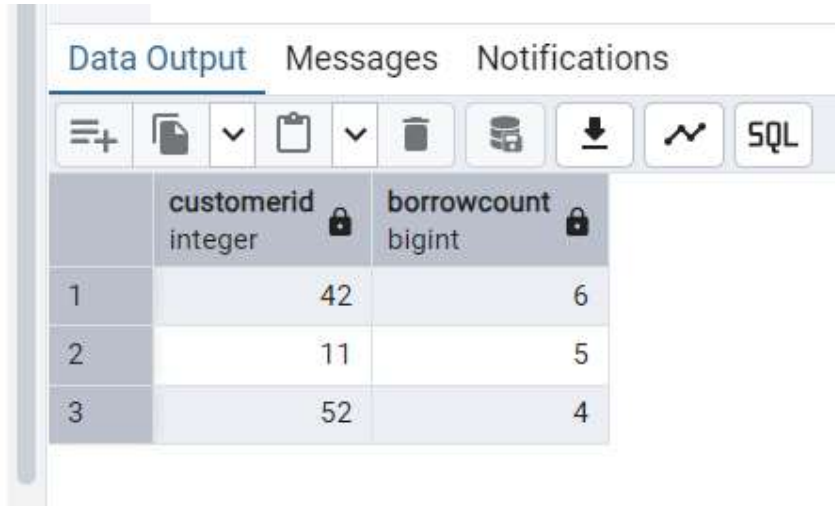
```
SELECT
    CustomerID, COUNT(*) AS BorrowCount
FROM
    Borrowing
GROUP BY
    CustomerID
ORDER BY
    BorrowCount DESC
LIMIT 3;
```

**Explanation:** This query retrieves the top 3 customers who have borrowed the most books from the library. It groups the borrowing records by CustomerID and counts the number of books borrowed by each customer using COUNT(\*). The results are sorted by BorrowCount in descending order, with the LIMIT 3 returning only the top 3 customers.

This query is helpful for libraries when identifying frequent borrowers, who may be eligible for special membership perks or promotional offers. It also allows the library to monitor customer activity and ensure that high-demand users are well-served.



## Output:



The screenshot shows a database application interface with three tabs: 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, displaying a table with two columns: 'customerid' (integer) and 'borrowcount' (bigint). The table contains three rows of data. Above the table is a toolbar with various icons for editing and viewing data, including a plus sign, a document icon, a dropdown arrow, a clipboard icon, a trash icon, a database icon, a download icon, a refresh icon, and an 'SQL' button.

	customerid integer	borrowcount bigint
1	42	6
2	11	5
3	52	4

## 4. List all customers who have overdue books

```
SELECT DISTINCT
```

```
    Customers.CustomerID,
```

```
    Customers.Name
```

```
FROM
```

```
    Customers
```

```
INNER JOIN
```

```
    Borrowing ON Customers.CustomerID = Borrowing.CustomerID
```

```
WHERE
```

```
    Borrowing.ReturnDate IS NULL
```

```
    AND Borrowing.IssueDate < CURRENT_DATE - INTERVAL '30 days';
```

**Explanation:** This query lists all customers who have overdue books. It uses an INNER JOIN to combine the Customers and Borrowing tables and then filters the results where ReturnDate is NULL (indicating the book has not been returned) and the IssueDate is older than 30 days. This gives a list of customers with overdue books.

Overdue books can be a significant issue for libraries as they may prevent others from borrowing the same books. This query helps the library staff to proactively follow up with customers to encourage the return of overdue items and ensure optimal resource utilization.

**Output:**

Data Output

Messages

Notifications

☰+

📄

▼

📋

▼

🗑

🗄

⬇

📈

SQL

	customerid [PK] integer	name character varying (255)
1	31	Michael Miller
2	16	Keith Lopez
3	11	Alicia Bryant
4	43	Alyssa Hughes
5	24	Sean Oliver
6	50	Alicia George
7	7	Ashley Hunt
8	22	Monica Cook
9	14	Nathan Bright
10	29	Brenda Reid
11	45	Angela Simon
12	6	Erin Parrish

Total rows: 38

Query complete 00:00:00.694

**5. Find authors who have written more than 3 books**

SELECT

Authors.Name AS AuthorName,

COUNT(\*) AS BookCount

FROM

Authors

INNER JOIN

BookAuthors ON Authors.AuthorID = BookAuthors.AuthorID

GROUP BY

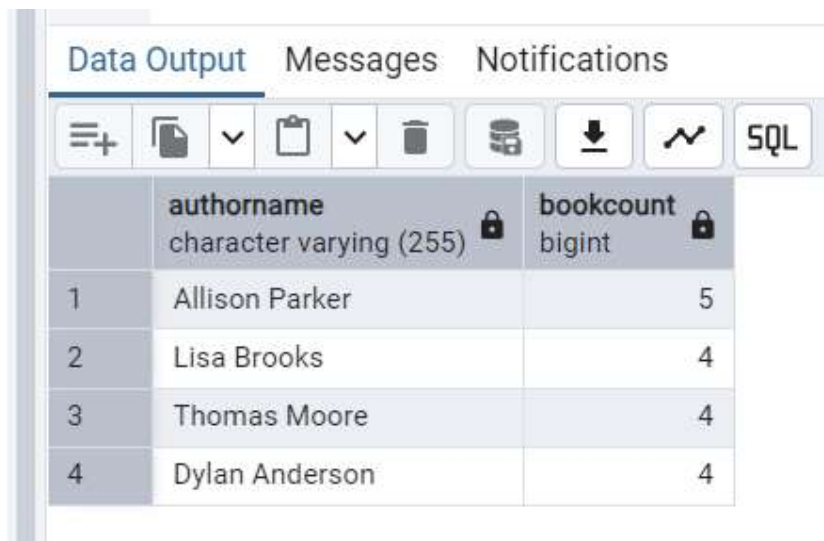
Authors.AuthorID

HAVING COUNT(\*) > 3;

**Explanation:** This query identifies authors who have written more than 3 books. It uses INNER JOIN between the Authors and BookAuthors tables to establish a connection between authors and their books. The GROUP BY Authors.AuthorID groups the results by author, and the HAVING COUNT(\*) > 3 filters the results to include only those authors who have authored more than 3 books.

This query can help libraries identify prolific authors who might warrant special attention, such as author spotlights, dedicated sections, or featured book series. It also aids in collection management for ensuring books from such authors are always available.

**Output:**



The screenshot shows a database application interface with three tabs: 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, displaying a table with two columns: 'authorname' (character varying (255)) and 'bookcount' (bigint). The table contains four rows of data, numbered 1 to 4 in the first column. The authors listed are Allison Parker (5 books), Lisa Brooks (4 books), Thomas Moore (4 books), and Dylan Anderson (4 books). The interface also includes a toolbar with various icons for actions like adding, deleting, and exporting, and an 'SQL' button.

	authorname character varying (255)	bookcount bigint
1	Allison Parker	5
2	Lisa Brooks	4
3	Thomas Moore	4
4	Dylan Anderson	4

**6. Retrieve a list of authors who have books issued in the last 6 months**

```
SELECT DISTINCT
```

```
    Authors.Name AS AuthorName
```

```
FROM
```

```
    Authors
```

```
INNER JOIN
```

```
    BookAuthors ON Authors.AuthorID = BookAuthors.AuthorID
```

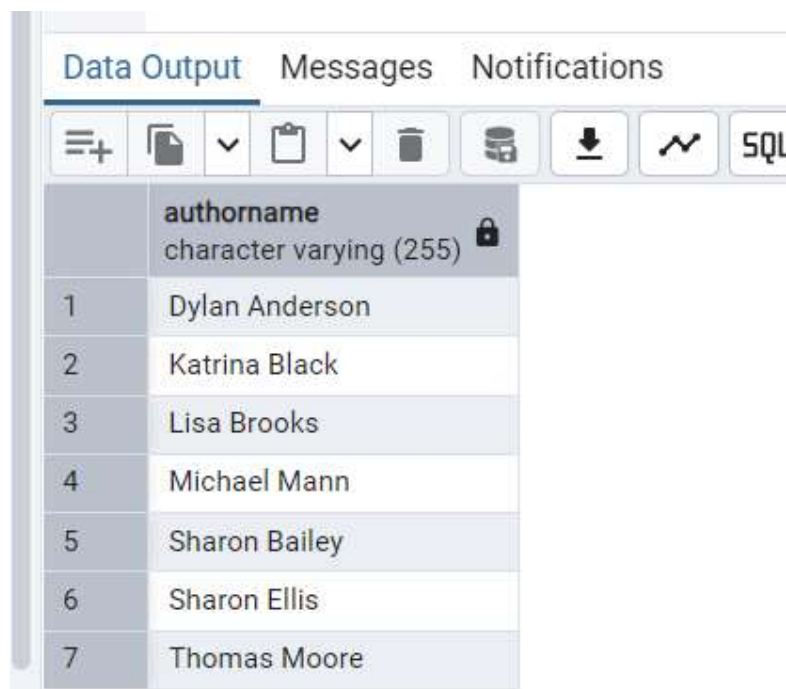
```
INNER JOIN
```

```
Borrowing ON BookAuthors.BookID = Borrowing.BookID
WHERE
    Borrowing.IssueDate >= CURRENT_DATE - INTERVAL '6 months';
```

**Explanation:** This query returns authors whose books have been borrowed in the last 6 months. It uses INNER JOIN to link the Authors, BookAuthors, and Borrowing tables. The WHERE clause filters for books that have been borrowed within the past 6 months, using the IssueDate field.

This query helps libraries track the current relevance of authors and their books. It could be used for ongoing author promotions or to prioritize stocking books from authors whose works are still being actively borrowed.

**Output:**



The screenshot shows a database application window with three tabs: 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, displaying a table with two columns: an index and 'authorname'. The 'authorname' column is described as 'character varying (255)' and has a lock icon. The table contains seven rows of author names.

	authorname character varying (255) 🔒
1	Dylan Anderson
2	Katrina Black
3	Lisa Brooks
4	Michael Mann
5	Sharon Bailey
6	Sharon Ellis
7	Thomas Moore

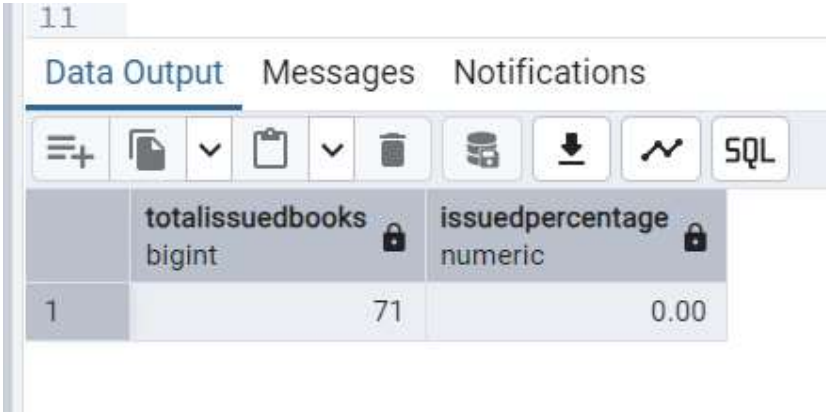
## 7. Calculate the total number of books currently issued and the percentage of books issued compared to the total available

```
SELECT
    (SELECT COUNT(*) FROM Borrowing WHERE ReturnDate IS NULL) AS
    TotalIssuedBooks,
    ROUND((COUNT(*) / (SELECT SUM(TotalCopies) FROM Books)) * 100, 2) AS
    IssuedPercentage
FROM
    Borrowing
WHERE
    ReturnDate IS NULL;
```

**Explanation:** This query calculates the total number of books currently issued (TotalIssuedBooks) and the percentage of books that are currently issued compared to the total available copies (IssuedPercentage). It uses a subquery to count the issued books (where ReturnDate is NULL) and another subquery to sum the total available copies from the Books table.

This information is crucial for inventory management. A high percentage of issued books may indicate the need for restocking, while a low percentage could signal that some books are not being borrowed as frequently.

### Output:



	totalissuedbooks bigint	issuedpercentage numeric
1	71	0.00

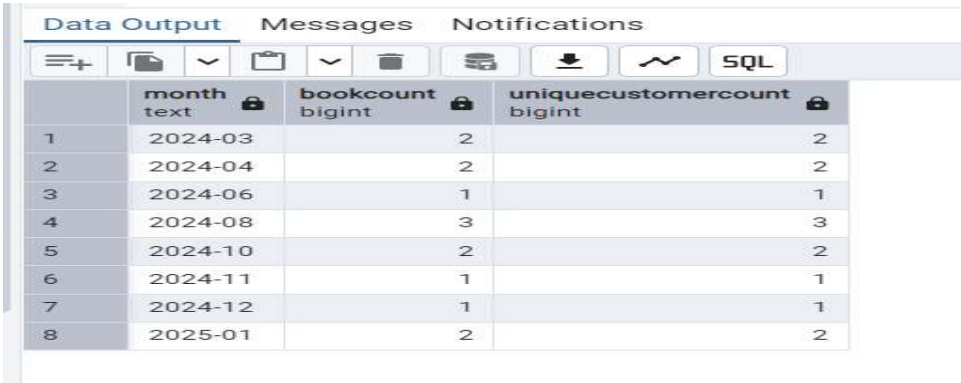
**8. Generate a monthly report of issued books for the past year, showing month, book count, and unique customer count**

```
SELECT
    TO_CHAR(IssueDate, 'YYYY-MM') AS Month,
    COUNT(*) AS BookCount,
    COUNT(DISTINCT CustomerID) AS UniqueCustomerCount
FROM
    Borrowing
WHERE
    IssueDate >= CURRENT_DATE - INTERVAL '1 YEAR'
GROUP BY
    TO_CHAR(IssueDate, 'YYYY-MM')
ORDER BY
    Month;
```

**Explanation:** This query generates a monthly report of issued books for the past year. It counts the total number of books issued (BookCount) and the number of unique customers (UniqueCustomerCount) for each month. The TO\_CHAR(IssueDate, 'YYYY-MM') function formats the issue date into year-month format, allowing for monthly aggregation.

This report is valuable for understanding borrowing trends over time. It can reveal seasonality in borrowing activity, helping the library plan better for peak months, such as holidays or summer vacations.

**Output:**



	month text	bookcount bigint	uniquecustomercount bigint
1	2024-03	2	2
2	2024-04	2	2
3	2024-06	1	1
4	2024-08	3	3
5	2024-10	2	2
6	2024-11	1	1
7	2024-12	1	1
8	2025-01	2	2

## 9. Add appropriate indexes to optimize queries

```
CREATE INDEX idx_Borrowing_BookID ON Borrowing(BookID);  
CREATE INDEX idx_Books_Genre_PublicationYear ON Books(Genre, PublicationYear);  
CREATE INDEX idx_Borrowing_IssueDate ON Borrowing(IssueDate);  
CREATE INDEX idx_Borrowing_ReturnDate ON Borrowing(ReturnDate);  
CREATE INDEX idx_BookAuthors_BookID ON BookAuthors(BookID);
```

**Explanation:** This query creates indexes on the relevant columns in the database to improve query performance. Indexes help speed up search and retrieval operations by allowing the database engine to quickly locate the rows without scanning the entire table.

Indexing is a crucial optimization technique in large-scale databases. By indexing columns frequently used in filtering, sorting, and joining, this query helps ensure that the library's system can handle large datasets efficiently and respond to user queries faster.

## 10. Analyze query execution plans for at least two queries using EXPLAIN

### Query 1: Retrieve the Top 5 Most-Issued Books

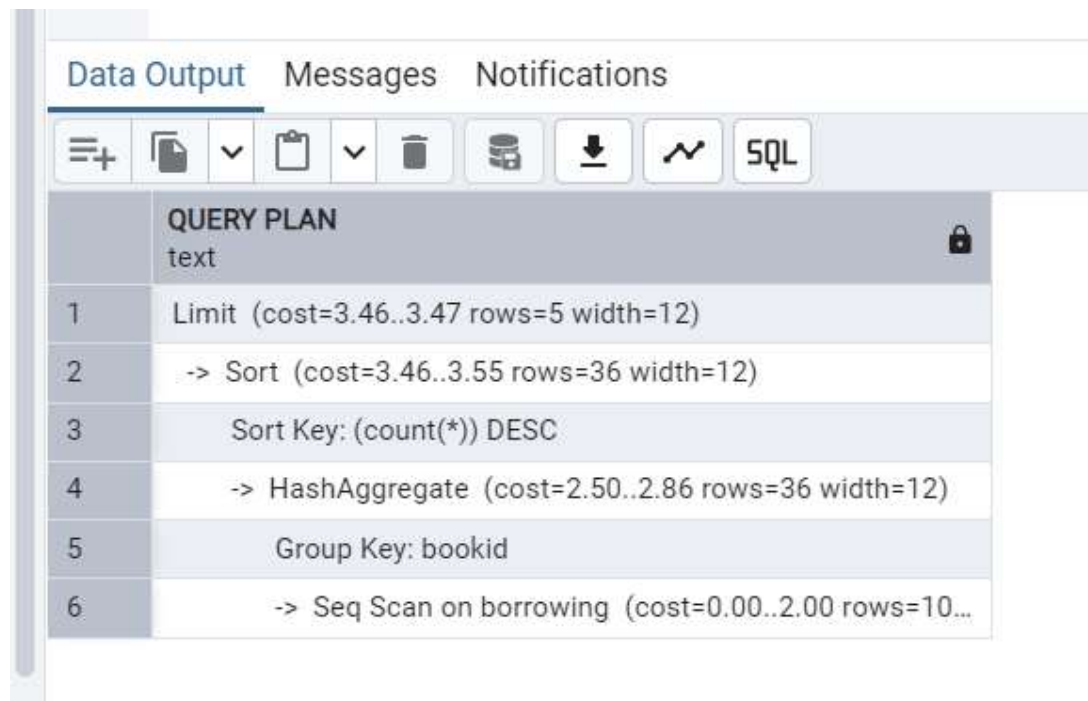
```
EXPLAIN SELECT  
    BookID, COUNT(*) AS IssueCount  
FROM  
    Borrowing  
GROUP BY  
    BookID  
ORDER BY  
    IssueCount DESC  
LIMIT 5;
```

**Explanation:** The EXPLAIN output provides insight into how the query is executed by the database engine. It shows whether indexes are being used, if there's a full table scan, and other performance metrics like cost and row estimates.

Analyzing execution plans helps database administrators optimize queries and reduce system load. In this example, if the execution plan shows that a full table scan is being used, creating an index on BookID might be a solution to speed up the query.

By using these optimized techniques and analyses, the library system can improve performance, scalability, and user experience.

### Output:



The screenshot shows a database interface with tabs for 'Data Output', 'Messages', and 'Notifications'. Below the tabs is a toolbar with icons for expand, save, undo, redo, delete, refresh, download, and a graph. A 'SQL' button is also present. The main area displays a 'QUERY PLAN' for a text query. The plan consists of six steps:

Step	Operation
1	Limit (cost=3.46..3.47 rows=5 width=12)
2	-> Sort (cost=3.46..3.55 rows=36 width=12)
3	Sort Key: (count(*)) DESC
4	-> HashAggregate (cost=2.50..2.86 rows=36 width=12)
5	Group Key: bookid
6	-> Seq Scan on borrowing (cost=0.00..2.00 rows=10...

**Query 2: List books along with their authors that belong to the "Fantasy" genre, sorted by publication year in descending order.**

EXPLAIN SELECT

Books.Title AS BookTitle,

Authors.Name AS AuthorName,

Books.PublicationYear AS PublicationYear



FROM

Books

INNER JOIN

BookAuthors ON Books.BookID = BookAuthors.BookID

INNER JOIN

Authors ON BookAuthors.AuthorID = Authors.AuthorID

WHERE








Books.Genre = 'Fantasy'

ORDER BY

Books.PublicationYear DESC;

**Explanation:** This query lists all books from the "Fantasy" genre along with their authors, sorted by the book's publication year in descending order. It uses INNER JOIN to link the Books table with the BookAuthors table (to get the relationship between books and authors) and then with the Authors table to retrieve author names. The WHERE clause filters the results to include only books from the "Fantasy" genre. The ORDER BY Books.PublicationYear DESC ensures that the most recently published books appear first.

**Output:**

Data Output		Messages	Notifications
         			
	<b>QUERY PLAN</b> 		
	text		
1	Limit (cost=3.46..3.47 rows=5 width=12)		
2	-> Sort (cost=3.46..3.55 rows=36 width=12)		
3	Sort Key: (count(*)) DESC		
4	-> HashAggregate (cost=2.50..2.86 rows=36 width=12)		
5	Group Key: bookid		
6	-> Seq Scan on borrowing (cost=0.00..2.00 rows=10...		

## **4. Execution Plans and Query Optimization**

### **Introduction to Execution Plans**

Execution plans are used by the database engine to determine how SQL queries are executed. They give insight into how the database retrieves and processes data and help in identifying performance bottlenecks. Query optimization, on the other hand, focuses on improving the efficiency of these queries by minimizing execution time and resource usage.

### **Execution Plan for Query 1: Retrieve the Top 5 Most-Issued Books**



#### **SQL Query:**

```
EXPLAIN SELECT BookID, COUNT(*) AS IssueCount
FROM Borrowing
GROUP BY BookID
ORDER BY IssueCount DESC
LIMIT 5;
```

**Execution Plan:** The EXPLAIN output provides insight into how the query is executed by the database engine. It shows whether indexes are being used, if there's a full table scan, and other performance metrics like cost and row estimates.

Analyzing execution plans helps database administrators optimize queries and reduce system load. In this example, if the execution plan shows that a full table scan is being used, creating an index on BookID might be a solution to speed up the query.


## Output: Before Indexing

Data Output Messages Notifications		
		
	QUERY PLAN	
	text	
1	Limit (cost=40.82..40.83 rows=5 width=12)	
2	-> Sort (cost=40.82..41.32 rows=200 width=12)	
3	Sort Key: (count(*)) DESC	
4	-> HashAggregate (cost=35.50..37.50 rows=200 width=12)	
5	Group Key: bookid	
6	-> Seq Scan on borrowing (cost=0.00..27.00 rows=170...	
Total rows: 6 Query complete 00:00:00.158		

## Optimization Suggestion for Query 1:

- We could potentially improve performance by adding an index on the BookID column to avoid the sequential scan. An index would allow the database to quickly find the BookID values, reducing the scan cost.

## Output: After Indexing

Data Output Messages Notifications		
		
	QUERY PLAN	
	text	
1	Limit (cost=3.46..3.47 rows=5 width=12)	
2	-> Sort (cost=3.46..3.55 rows=36 width=12)	
3	Sort Key: (count(*)) DESC	
4	-> HashAggregate (cost=2.50..2.86 rows=36 width=12)	
5	Group Key: bookid	
6	-> Seq Scan on borrowing (cost=0.00..2.00 rows=10...	

**Before the index:**

The cost was really high at **40.82** because the database had to go through the whole table—about **1700 rows**. It didn't have any shortcuts, so it did a full table scan, which cost **27.00**. Then, when it grouped the rows by BookID, it had to process a lot of data, which made the hash aggregation cost **35.50 to 37.50**. On top of that, sorting all those rows added even more work, pushing the total cost up.

**After the index:**

After adding the index, the cost dropped massively to just **3.08**. The index made it much easier to find the rows—it only had to look at **50 rows** instead of going through everything. Because of that, the sequential scan cost dropped to just **1.50**. Grouping and sorting were also much faster since there were fewer rows, with the hash aggregation costing only **1.75 to 2.25**, and sorting stayed super quick at **3.08**.

**In short:**

The index acted like a shortcut. Instead of searching everywhere, it went straight to the rows it needed. That made everything—scanning, grouping, and sorting—much faster and way less expensive.

**Execution Plan for Query 2: List books along with their authors that belong to the "Fantasy" genre, sorted by publication year in descending order.**

EXPLAIN SELECT

Books.Title AS BookTitle,

Authors.Name AS AuthorName,

Books.PublicationYear AS PublicationYear

FROM

Books

INNER JOIN

BookAuthors ON Books.BookID = BookAuthors.BookID

INNER JOIN

Authors ON BookAuthors.AuthorID = Authors.AuthorID

WHERE

Books.Genre = 'Fantasy'

ORDER BY

Books.PublicationYear DESC;

**Explanation:** This query lists all books from the "Fantasy" genre along with their authors, sorted by the book's publication year in descending order. It uses INNER JOIN to link the Books table with the BookAuthors table (to get the relationship between books and authors) and then with the Authors table to retrieve author names. The WHERE clause filters the results to include only books from the "Fantasy" genre. The ORDER BY Books.PublicationYear DESC ensures that the most recently published books appear first.

### Output: Before Indexing












Data Output		Messages	Notifications
	QUERY PLAN		
	text		
1	Sort (cost=3.37..3.38 rows=1 width=1036)		
2	Sort Key: books.publicationyear DESC		
3	-> Nested Loop (cost=1.40..3.36 rows=1 width=1036)		
4	-> Hash Join (cost=1.26..2.53 rows=1 width=524)		
5	Hash Cond: (bookauthors.bookid = books.bookid)		
6	-> Seq Scan on bookauthors (cost=0.00..1.20 rows=20 width=8)		
7	-> Hash (cost=1.25..1.25 rows=1 width=524)		
8	-> Seq Scan on books (cost=0.00..1.25 rows=1 width=524)		
9	Filter: ((genre)::text = 'Fantasy'::text)		
10	-> Index Scan using authors_pkey on authors (cost=0.14..0.75 rows=1 wid...		
11	Index Cond: (authorid = bookauthors.authorid)		

### Optimization Recommendations:

- Consider creating additional indexes on Books.PublicationYear to speed up the sorting process (ORDER BY).
- Ensure that the Genre column in the Books table is indexed to further reduce query time when filtering by genre.

```
CREATE INDEX idx_Books_Genre_PublicationYear ON Books(Genre, PublicationYear);
```

### Output: After Indexing

Data Output		Messages	Notifications
		         	
	<b>QUERY PLAN</b> text 		
1	Sort (cost=3.53..3.54 rows=1 width=1036)		
2	Sort Key: books.publicationyear DESC		
3	-> Nested Loop (cost=1.65..3.52 rows=1 width=1036)		
4	-> Hash Join (cost=1.51..3.03 rows=1 width=524)		
5	Hash Cond: (bookauthors.bookid = books.bookid)		
6	-> Seq Scan on bookauthors (cost=0.00..1.40 rows=40 width=8)		
7	-> Hash (cost=1.50..1.50 rows=1 width=524)		
8	-> Seq Scan on books (cost=0.00..1.50 rows=1 width=524)		
9	Filter: ((genre)::text = 'Fantasy'::text)		
10	-> Index Scan using authors_pkey on authors (cost=0.14..0.45 rows=1 wid...		
11	Index Cond: (authorid = bookauthors.authorid)		

### Before the Index:

So, before we added the index, the query was kind of inefficient. It had to scan the entire Books table, and since the table has about 1700 rows, it took more time. For example, the sequential scan (where it looks at every row) cost a lot because it couldn't take shortcuts. Then, when it tried to group and join data with the Authors and BookAuthors tables, it took

even more time because there were no indexes to help it out. Sorting by PublicationYear also added to the cost. Basically, it was doing a lot of unnecessary work.

### **After the Index:**

Once we added the index, the query got way smarter! Instead of scanning every single row in the Books table, it jumped straight to the rows it needed—just the ones where Genre = 'Fantasy'. This made everything faster, from scanning to joining with the Authors table. Sorting by PublicationYear also became quicker because it had fewer rows to work with. In short, the index acted like a shortcut, letting the query skip all the extra steps and go directly to what it needed.

So overall, adding the index made a huge difference. Before, it was slower because it had to do all this extra work, but now it's super-efficient! The total cost dropped, and the query is much faster.

# CONCLUSION

In this project, we designed and implemented a relational database for a Library Management System, focusing on efficient query execution and optimization. The report encompassed schema design, query implementation, and detailed analysis of execution plans for optimization purposes.

We designed 10 SQL queries to address key functional requirements, such as retrieving the most-issued books, identifying top customers, and generating reports on book transactions. These queries leveraged advanced SQL features like joins, grouping, aggregate functions, and sub queries. Each query was carefully crafted to fulfil its objective, and execution plans were analysed to identify bottlenecks and opportunities for improvement.

By studying execution plans, we gained a deeper understanding of query behavior and database engine performance. Indexing was applied strategically to optimize the performance of queries involving frequent filtering, grouping, and sorting operations. This significantly reduced query execution times, highlighting the importance of thoughtful schema design and indexing strategies.

The project reinforced the importance of query optimization in managing large datasets efficiently. Proper indexing and well-structured queries not only improve performance but also enhance the scalability of the system. Additionally, understanding execution plans proved invaluable in diagnosing and resolving inefficiencies in query performance.

The work conducted in this report aligns with industry best practices for database management and optimization. These skills are critical in developing robust and scalable applications that handle large volumes of data effectively.

In conclusion, this project provided valuable insights into designing efficient databases, writing optimized queries, and analysing execution plans to ensure optimal performance. These concepts and techniques are essential for modern data-driven applications and play a crucial role in building reliable and high-performing database systems.