

National Economics University
Faculty of Data Science and Artificial Intelligence



DATABASE COURSE PROJECT ASSIGNMENT

TOPIC: LIBRARY INFORMATION MANAGER

Group members:

Duong Dinh Anh - 11247256

Nguyen Hoang Tuan - 11247363

Tran Khai Van - 11247368

Nguyen Thi Nien - 11247337

Supervisor: Tran Duc Minh

December 5, 2025

Contents

1	Introduction & Business Problem	4
1.1	Organizational Overview	4
1.2	Operational Challenges	4
1.3	System Objectives	5
2	Library Management System: Data Structure & Normalization Analysis	6
2.1	ERD Overview	6
2.2	Table Descriptions	7
2.2.1	Category	7
2.2.2	Staff	7
2.2.3	Member	8
2.2.4	Book	8
2.2.5	Loans	9
2.2.6	Payment	9
2.3	Data Normalization Process	10
2.3.1	First Normal Form (1NF)	10
2.3.2	Second Normal Form (2NF)	10
2.3.3	Third Normal Form (3NF)	10
3	Database Script Analysis and Implementation Strategy	12
3.1	The Structural Framework (<code>schema.sql</code>)	12
3.1.1	Environment Initialization and Idempotency	12
3.1.2	Core Entity Definition (Normalization & Classification)	12
3.1.3	Inventory Architecture (Book Table)	13
3.1.4	Transactional Architecture (Loan & Payment Tables)	13
3.2	Data Simulation and Testing Scenarios (<code>seed.sql</code>)	13
3.2.1	Foundation Data Population	13
3.2.2	Scenario-Based Transaction Testing	13
3.2.3	Financial Aggregation Testing	14
3.3	Conclusion	14
4	Database Implementation and Business Logic Architecture	15
4.1	Data Abstraction Layer (Database Views)	15
4.1.1	Inventory Status View (<code>v_books_status</code>)	15
4.1.2	Circulation Tracking View (<code>v_active_loans</code>)	16
4.1.3	Financial Liability View (<code>v_overdue_fines</code>)	16
4.2	Transactional Business Logic (Stored Procedures)	16

4.2.1	Borrowing Transaction (<code>sp_borrow_book</code>)	17
4.2.2	Statistical Aggregation (<code>sp_monthly_activity_report</code>)	17
4.3	Automated Integrity Enforcement (Database Triggers)	18
4.3.1	Return Synchronization Trigger (<code>trg_after_loan_update</code>)	18
5	Implementation Results & Visualization Analysis	20
5.1	MySQL Implementation Verification	20
5.1.1	User Privileges & System Configuration	20
5.1.2	View Execution: Inventory Status	20
5.1.3	Complex Query: Most Borrowed Books	21
5.1.4	Trigger Validation: Automated Fine Calculation	22
5.2	Python Integration & Reporting	23
5.2.1	Python Console Output	23
5.2.2	Monthly Activity Report	24
5.3	Data Visualization & Analytics	25
5.3.1	Inventory Distribution Analysis	25
5.3.2	Active Loans Status	26
5.3.3	Trigger Logic Linear Regression	27
	Conclusion	29

Chapter 1

Introduction & Business Problem

1.1 Organizational Overview

The project focuses on a university library system responsible for managing a vast and diverse collection of academic resources, including books, journals, and digital media. The library serves a distinct community of members comprising students, faculty, and staff, each requiring access to these resources for educational and research purposes.

1.2 Operational Challenges

The library currently faces challenges in efficiently tracking the circulation of its inventory. As the number of books and members grows, manual or decentralized methods of tracking are becoming prone to errors and data inconsistency. The core operational challenges include:

- **Inventory Management:** The library needs to classify books into specific categories (e.g., Science, Literature, IT) and maintain accurate counts of total versus available copies. Tracking duplicate copies and unique ISBNs is essential for inventory control.
- **Circulation Tracking:** A primary requirement is to track the lifecycle of a loan—from the moment a book is issued to when it is returned. The system must distinguish between books that are currently borrowed, returned, or overdue.
- **Member & Staff Administration:** The system must manage personal details for all library members and internal staff. It is crucial to categorize members (Student, Faculty, Staff) to potentially apply different borrowing privileges or policies in the future.
- **Financial Management (Fines & Payments):** A significant pain point is the manual calculation of late fees. The library requires an automated solution to calculate fines based on the number of days a book is overdue and to record subsequent payments (via cash, card, or transfer) accurately.

1.3 System Objectives

To resolve these issues, the library requires a centralized relational database system named the **Library Information Manager**. This system aims to achieve the following objectives:

- **Data Consistency:** Ensure that data regarding books, members, and transactions remains accurate and synchronized across the organization.
- **Process Automation:**
 - Automatically decrease stock availability when a loan is issued.
 - Automatically increase stock availability when a book is returned.
 - Automatically calculate overdue days and generate fine records if a book is returned after its due date.
- **Reporting and Analysis:** Provide management with the ability to generate reports on borrowing frequency, overdue trends, and financial revenue. The system should also support data analysis (via Python) to visualize key metrics, such as the most popular categories or the ratio of active vs. overdue loans.

Chapter 2

Library Management System: Data Structure & Normalization Analysis

2.1 ERD Overview

The Library Management System is designed around six core entities: **Category**, **Book**, **Member**, **Loan**, **Payment**, and **Staff**.

The relationships between these entities are established to ensure data integrity and efficient retrieval. Specifically, the system operates on a **One-to-Many (1:N)** relationship model:

- A single **Category** classifies multiple **Books**.
- A **Member** can initiate multiple **Loan** transactions.
- A specific **Book** title can be involved in numerous **Loan** records over time.
- A single **Loan** transaction may generate multiple **Payment** records (such as fines or fees).

This structure allows for comprehensive tracking of library operations.

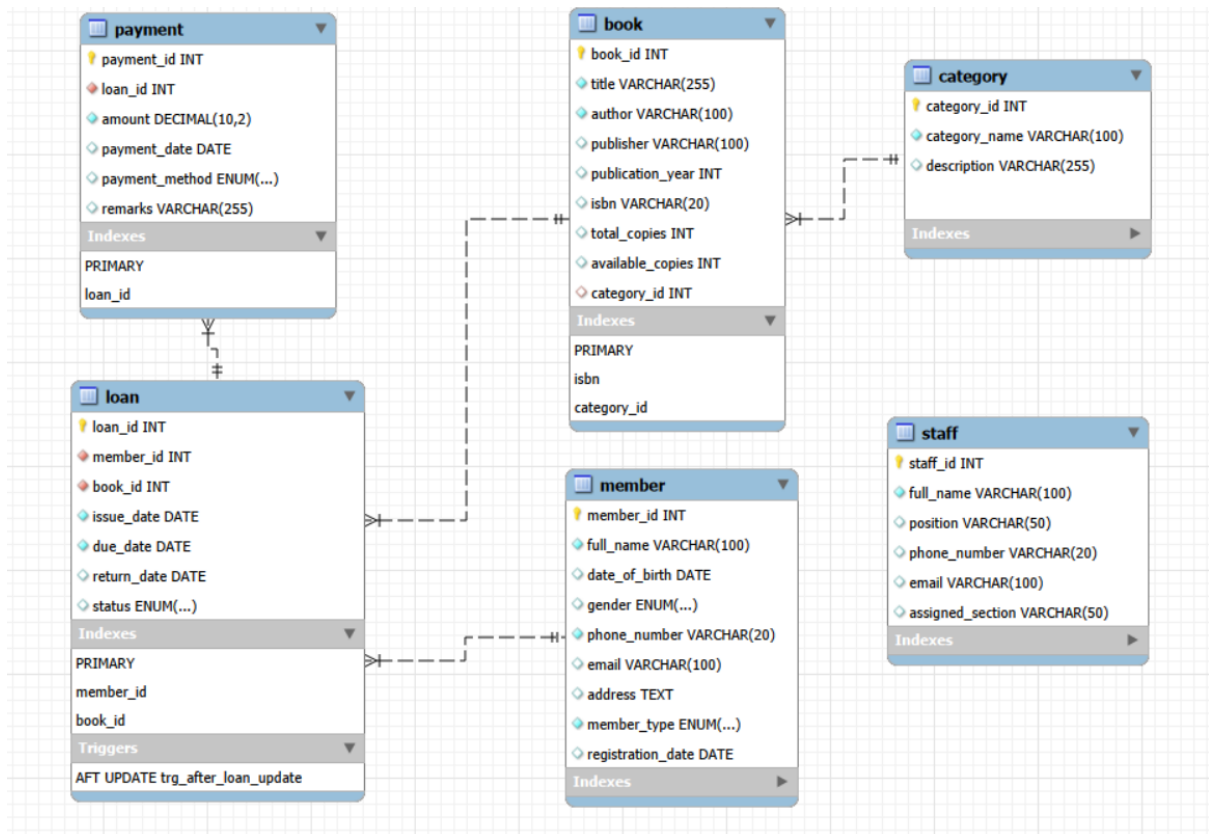


Figure 2.1: Entity Relationship Diagram (ERD) of the Library Management System.

2.2 Table Descriptions

2.2.1 Category

Stores information about book genres/categories.

Field	Type	Key	Description
category_id	INT	PK	Unique identifier for the category (Auto-increment).
category_name	VARCHAR(100)		Name of the book category (Unique).
description	VARCHAR(255)		Brief description or details about the category.

Table 2.1: Category Table Structure

2.2.2 Staff

Stores details of library employees.

Field	Type	Key	Description
staff_id	INT	PK	Unique identifier for the staff member.
full_name	VARCHAR(100)		Full legal name of the staff member.
position	VARCHAR(50)		Job title or role (e.g., Librarian, Manager).
phone_number	VARCHAR(20)		Contact phone number.
email	VARCHAR(100)		Work email address (Unique).
assigned_section	VARCHAR(50)		The specific library section managed by the staff.

Table 2.2: Staff Table Structure

2.2.3 Member

Stores information about library users (borrowers).

Field	Type	Key	Description
member_id	INT	PK	Unique identifier for the library member.
full_name	VARCHAR(100)		Full name of the member.
date_of_birth	DATE		Member's date of birth.
gender	ENUM		Gender of the member ('Male', 'Female', 'Other').
phone_number	VARCHAR(20)		Contact phone number.
email	VARCHAR(100)		Contact email address (Unique).
address	TEXT		Residential address of the member.
member_type	ENUM		Classification ('Student', 'Faculty', 'Staff').
registration_date	DATE		The date when the member registered.

Table 2.3: Member Table Structure

2.2.4 Book

Stores details of the books available in the library.

Field	Type	Key	Description
book_id	INT	PK	Unique identifier for the book.
title	VARCHAR(255)		Title of the book.
author	VARCHAR(100)		Name of the book's author.
publisher	VARCHAR(100)		Name of the publishing house.
publication_year	INT		Year the book was published.
isbn	VARCHAR(20)		International Standard Book Number (Unique).
total_copies	INT		Total number of physical copies owned by the library.
available_copies	INT		Current number of copies available for borrowing.
category_id	INT	FK	Foreign key referencing <code>Category.category_id</code> .

Table 2.4: Book Table Structure

2.2.5 Loans

Records the history of borrowing transactions.

Field	Type	Key	Description
loan_id	INT	PK	Unique identifier for the loan transaction.
member_id	INT	FK	Foreign key referencing <code>Member.member_id</code> .
book_id	INT	FK	Foreign key referencing <code>Book.book_id</code> .
issue_date	DATE		The date the book was borrowed.
due_date	DATE		The expected return date.
return_date	DATE		The actual date the book was returned (NULL if active).
status	ENUM		Status ('Borrowed', 'Returned', 'Overdue').

Table 2.5: Loans Table Structure

2.2.6 Payment

Records financial transactions related to fines or fees.

Field	Type	Key	Description
payment_id	INT	PK	Unique identifier for the payment transaction.
loan_id	INT	FK	Foreign key referencing <code>Loan.loan_id</code> .
amount	DECIMAL		The monetary value of the payment.
payment_date	DATE		The date the payment was made.
payment_method	ENUM		Method ('Cash', 'Card', 'Transfer').
remarks	VARCHAR(255)		Additional notes (e.g., reason for the fine).

Table 2.6: Payment Table Structure

2.3 Data Normalization Process

The database schema has undergone a rigorous normalization process to transform it from an Unnormalized Form (UNF) to the Third Normal Form (3NF).

2.3.1 First Normal Form (1NF)

To achieve 1NF, the initial dataset was analyzed to ensure atomicity and eliminate repeating groups. In a raw format, a member's record might have listed multiple borrowed books in a single cell. We resolved this by separating borrowing activities into a distinct **Loans** table. Consequently, every field in the database now contains only atomic (single) values, and every table is defined by a unique Primary Key.

2.3.2 Second Normal Form (2NF)

The requirement for 2NF is the elimination of partial dependencies, which typically occur in tables with composite primary keys. In our design, every table utilizes a **Surrogate Key** (a single-column, auto-incrementing Integer like `loan_id` or `book_id`) as its Primary Key. Since the primary key consists of only one attribute, it is impossible for non-key attributes to depend on only a "part" of the key. Therefore, the schema automatically satisfies 2NF.

2.3.3 Third Normal Form (3NF)

To reach 3NF, we focused on removing transitive dependencies - where a non-key attribute depends on another non-key attribute. We implemented two significant optimizations:

- **Category Separation:** Originally, category details (Name, Description) might have been stored in the **Book** table. This would mean *Description* depended on *Name*, which depended on *Book_ID*. By moving these details to a separate **Category** table and linking via `category_id`, we eliminated this dependency.
- **Payment Table Optimization:** Initially, the **Payment** table contained both `loan_id` and `member_id`. However, since a **Loan** is already uniquely assigned to a

Member, storing `member_id` in the **Payment** table was redundant and created a transitive dependency ($\text{Payment} \rightarrow \text{Loan} \rightarrow \text{Member}$). We removed `member_id` from the Payment table. Now, to identify who made a payment, the system joins the Payment table with the Loan table. This ensures absolute data consistency and eliminates redundancy.

Conclusion: The current Entity Relationship Diagram (ERD) is fully normalized to **3NF**, ensuring a robust, scalable, and anomaly-free database structure.

Chapter 3

Database Script Analysis and Implementation Strategy

The implementation of the Library Management System is effectuated through two distinct SQL scripts. The `schema.sql` script serves as the structural foundation, utilizing Data Definition Language (DDL) to enforce architectural constraints and normalization standards. Complementing this, the `seed.sql` script employs Data Manipulation Language (DML) to populate the system with scenario-based datasets designed for rigorous testing of business logic and reporting mechanisms.

3.1 The Structural Framework (`schema.sql`)

This script defines the physical schema of the database, establishing the entities, attributes, and relationships necessary to support the application's domain logic. The design strictly adheres to the Third Normal Form (3NF) to ensure data integrity.

3.1.1 Environment Initialization and Idempotency

The script begins with `DROP DATABASE IF EXISTS` and `CREATE DATABASE` commands. This ensures an idempotent deployment process, guaranteeing that the database environment is clean and free of legacy conflicts prior to table creation.

3.1.2 Core Entity Definition (Normalization & Classification)

- **Category Table:** Acts as a lookup entity to prevent data redundancy. By externalizing genre data here, the system avoids repetitive string storage in the `Book` table, facilitating efficient updates and consistent reporting.
- **Member Table:** Utilizes `ENUM` constraints for the `member_type` attribute (Student, Faculty, Staff). This schema enforcement ensures that only valid user roles are entered, which is critical for applying role-based borrowing policies later in the application layer.
- **Staff Table:** Segregates administrative personnel from library patrons, ensuring a clear distinction between system operators and service users.

3.1.3 Inventory Architecture (Book Table)

The **Book** entity implements referential integrity via the `category_id` Foreign Key. Crucially, it tracks inventory through two distinct metrics: `total_copies` (asset ownership) and `available_copies` (current circulation potential). This separation is vital for the logic used in the `sp_borrow_book` stored procedure.

3.1.4 Transactional Architecture (Loan & Payment Tables)

- **The Loan Entity:** Functioning as an associative entity, this table records the temporal relationship between a **Member** and a **Book**. It captures the full lifecycle of a transaction through `issue_date`, `due_date`, and `return_date`, alongside a status flag for rapid state filtering.
- **The Payment Entity (3NF Compliance):** A critical design feature is the exclusion of the `member_id` from this table. The **Payment** table references only the **Loan** entity (`loan_id`). This eliminates transitive dependency ($\text{Payment} \rightarrow \text{Loan} \rightarrow \text{Member}$), thereby satisfying Third Normal Form requirements and ensuring that financial liability is strictly coupled to a specific transaction context.

3.2 Data Simulation and Testing Scenarios (`seed.sql`)

The `seed.sql` script is not merely a data population tool; it is a structured simulation suite designed to validate specific system behaviors and edge cases. The data is stratified into logical groups to test distinct business rules.

3.2.1 Foundation Data Population

Initial inserts generate a diverse set of Categories, Staff, and Members to validate data types and constraints. The **Book** insertions include edge cases, such as popular titles where `available_copies` = 0, to verify that the borrowing logic correctly rejects loans when stock is depleted.

3.2.2 Scenario-Based Transaction Testing

The **Loan** records are categorized into four distinct testing groups:

- **Group 1: Nominal Returns (Control Group):** Records where books were returned within the allowable timeframe. These records validate standard reporting and ensure that no false-positive fines are generated.
- **Group 2: Late Returns (Exception Handling):** Records where `return_date` > `due_date`. These entries serve as the test bed for the system's punitive logic (Triggers and Views), ensuring that overdue days are calculated correctly.
- **Group 3: Active Circulation (State Monitoring):** Records with NULL return dates and current timestamps. These are used to validate the `v_active_loans` view, ensuring the system can correctly identify items currently off the shelf.

- **Group 4: Overdue Active (Alerting):** Records with `NULL` return dates where the `due_date` has passed. These entries simulate non-compliant borrowers, allowing for the testing of alert mechanisms and overdue status visualization.

3.2.3 Financial Aggregation Testing

The `Payment` inserts are mapped 1-to-1 with the “Group 2” late loans. This dataset allows for the verification of financial aggregation queries (e.g., `sp_monthly_activity_report`), ensuring that the system accurately sums collected fines and links them to the correct historical loan records.

3.3 Conclusion

In synthesis, `schema.sql` provides the rigorous constraints necessary for data consistency, while `seed.sql` provides the comprehensive datasets required to verify that those constraints and the associated business logic function correctly under various operational conditions.

Chapter 4

Database Implementation and Business Logic Architecture

The implementation of the Library Management System prioritizes data integrity, operational efficiency, and strict adherence to normalization principles. The database architecture utilizes three core components: **Views** for data abstraction, **Stored Procedures** for transactional encapsulation, and **Triggers** for automated consistency enforcement.

4.1 Data Abstraction Layer (Database Views)

To facilitate efficient reporting and shield the application layer from the complexity of the underlying Third Normal Form (3NF) schema, a series of virtual tables (Views) have been established. These views provide a denormalized presentation of data suitable for end-user consumption.

4.1.1 Inventory Status View (v_books_status)

This view serves as the primary dashboard for inventory management. By executing an `INNER JOIN` between the `Book` and `Category` entities, it resolves numerical foreign keys into human-readable descriptors. Furthermore, it introduces a derived attribute, `availability_pct`, which calculates the real-time stock percentage on the database server. This approach reduces the computational load on the client-side application and ensures consistent statistical reporting across all interfaces.

Listing 4.1: SQL for View `v_books_status`

```
1 CREATE OR REPLACE VIEW v_books_status AS
2 SELECT
3     b.book_id, b.title, c.category_name,
4     b.total_copies, b.available_copies,
5     ROUND((b.available_copies / b.total_copies) * 100, 1) AS
6         availability_pct
7 FROM BOOK b
8 JOIN CATEGORY c ON b.category_id = c.category_id;
```

4.1.2 Circulation Tracking View (v_active_loans)

Designed to monitor ongoing library transactions, this view implements a temporal filtering mechanism (`WHERE return_date IS NULL`) to isolate active loans. A critical feature of this view is the dynamic status evaluation logic. Utilizing a conditional `CASE` expression, the system evaluates the `due_date` against the current system timestamp (`CURDATE()`) to automatically classify a transaction as 'Active' or 'Overdue' without requiring physical updates to the database records.

Listing 4.2: SQL for View v_active_loans

```
1 CREATE OR REPLACE VIEW v_active_loans AS
2 SELECT
3     l.loan_id, m.full_name AS member_name, b.title AS book_title,
4     l.issue_date, l.due_date,
5     CASE
6         WHEN CURDATE() > l.due_date THEN 'Overdue'
7         ELSE 'Active'
8     END AS status
9 FROM Loan l
10 JOIN Member m ON l.member_id = m.member_id
11 JOIN Book b ON l.book_id = b.book_id
12 WHERE l.return_date IS NULL;
```

4.1.3 Financial Liability View (v_overdue_fines)

This view validates the system's adherence to 3NF constraints. Since the `Payment` table was normalized to exclude the redundant `member_id`, this view reconstructs the necessary relationship chain (`Payment → Loan → Member`) to associate financial liabilities with specific users. This demonstrates the system's ability to maintain a normalized structure while ensuring full data traceability for financial audits.

Listing 4.3: SQL for View v_overdue_fines

```
1 CREATE OR REPLACE VIEW v_overdue_fines AS
2 SELECT
3     m.member_id, m.full_name, b.title,
4     p.amount, p.remarks,
5     DATEDIFF(l.return_date, l.due_date) AS days_overdue
6 FROM Payment p
7 JOIN Loan l ON p.loan_id = l.loan_id
8 JOIN Member m ON l.member_id = m.member_id
9 JOIN Book b ON l.book_id = b.book_id
10 WHERE p.remarks LIKE '%Late%';
```

4.2 Transactional Business Logic (Stored Procedures)

To ensure Atomicity, Consistency, Isolation, and Durability (ACID) properties, critical business operations are encapsulated within Stored Procedures. This server-side logic centralization prevents data anomalies arising from concurrent access or improper handling by client applications.

4.2.1 Borrowing Transaction (sp_borrow_book)

This procedure manages the lending lifecycle through a strictly validation-first approach. Before initiating a transaction, the system performs a `SELECT ... INTO` operation to verify inventory availability. If the condition `available_copies > 0` is met, the procedure executes an atomic sequence: inserting a new record into the `Loan` entity with a calculated `due_date` and simultaneously decrementing the physical inventory in the `Book` entity. This mechanism effectively prevents "overselling" scenarios where a book might be lent out despite zero availability.

Listing 4.4: Stored Procedure `sp_borrow_book`

```
1 CREATE PROCEDURE sp_borrow_book(IN p_member_id INT, IN p_book_id
  INT)
2 BEGIN
3     DECLARE v_available INT;
4     SELECT available_copies INTO v_available FROM Book WHERE
       book_id = p_book_id;
5     IF v_available > 0 THEN
6         INSERT INTO Loan (member_id, book_id, issue_date,
           due_date, status)
7         VALUES (p_member_id, p_book_id, CURDATE(), DATE_ADD(
           CURDATE(), INTERVAL 14 DAY), 'Borrowed');
8         UPDATE Book SET available_copies = available_copies - 1
           WHERE book_id = p_book_id;
9     END IF;
10 END //
```

4.2.2 Statistical Aggregation (sp_monthly_activity_report)

To support management decision-making, this procedure leverages the database engine's native aggregation capabilities. Instead of retrieving raw data for client-side processing, the procedure utilizes conditional aggregation (`SUM(CASE WHEN...)`) to synthesize complex metrics—such as total borrowings, late returns, and financial collections—within a single query execution plan. This design significantly minimizes network latency and data transfer overhead.

Listing 4.5: Stored Procedure `sp_monthly_activity_report`

```
1 CREATE PROCEDURE sp_monthly_activity_report(IN p_year INT, IN
  p_month INT)
2 BEGIN
3     SELECT
4         COUNT(loan_id) AS total_borrowed,
5         SUM(CASE WHEN return_date IS NOT NULL THEN 1 ELSE 0 END)
           AS total_returned,
6         SUM(CASE WHEN return_date > due_date THEN 1 ELSE 0 END)
           AS total_overdue_returns,
7         COALESCE(SUM(p.amount), 0) AS total_fines_collected
8     FROM Loan l
9     LEFT JOIN Payment p ON l.loan_id = p.loan_id
10    WHERE YEAR(l.issue_date) = p_year AND MONTH(l.issue_date) =
       p_month;
```

4.3 Automated Integrity Enforcement (Database Triggers)

An Event-Driven Architecture is implemented via Database Triggers to automate maintenance tasks and enforce business rules that react to state changes within the system.

4.3.1 Return Synchronization Trigger (`trg_after_loan_update`)

This trigger functions as the system's automated consistency guardian. It is activated exclusively upon the transition of a loan status from 'Active' (NULL return date) to 'Returned'.

- **Inventory Synchronization:** Upon activation, the trigger automatically executes an UPDATE statement to increment the `available_copies` in the `Book` table, ensuring the physical inventory always reflects the circulation history.
- **Automated Financial Logic:** Simultaneously, the trigger evaluates the temporal difference between the `return_date` and `due_date`. If a violation is detected (`DATEDIFF > 0`), the system automatically calculates the punitive fee and inserts a record into the `Payment` table. Notably, this insertion respects the 3NF schema by referencing only the `loan_id`, allowing the relational model to implicitly identify the liable member.

Listing 4.6: Trigger `trg_after_loan_update`

```

1 CREATE TRIGGER trg_after_loan_update AFTER UPDATE ON Loan
2 FOR EACH ROW
3 BEGIN
4     DECLARE v_days_overdue INT;
5     DECLARE v_fine_per_day DECIMAL(10,2) DEFAULT 2.00;
6
7     IF OLD.return_date IS NULL AND NEW.return_date IS NOT NULL
8     THEN
9         UPDATE Book SET available_copies = available_copies + 1
10        WHERE book_id = NEW.book_id;
11
12        IF NEW.return_date > NEW.due_date THEN
13            SET v_days_overdue = DATEDIFF(NEW.return_date, NEW.
14            due_date);
15            INSERT INTO Payment (loan_id, amount, remarks)
16            VALUES (NEW.loan_id, (v_days_overdue * v_fine_per_day
17            ), CONCAT('Auto-fine: Late', v_days_overdue, '
18            days'));
19        END IF;
20    END IF;
21 END //
```

The resulting database system is robust and self-sustaining. By delegating data integrity checks, calculations, and relational enforcement to the database layer (via Procedures and Triggers), the architecture minimizes human error and ensures that business rules are applied consistently, regardless of the external application accessing the data.

Chapter 5

Implementation Results & Visualization Analysis

This chapter details the validation of the Library Management System through the execution of complex queries, integration with Python, and data visualization. Each section analyzes the outputs to confirm that the system meets the functional requirements defined in Chapter 2.

5.1 MySQL Implementation Verification

5.1.1 User Privileges & System Configuration

The first step in deployment involved configuring the database environment. As shown in Figure 5.1, the `ALTER USER` command was successfully executed to set the authentication plugin to `mysql_native_password`. This configuration is critical for ensuring compatibility with the Python connector driver, preventing authentication errors during the integration phase.

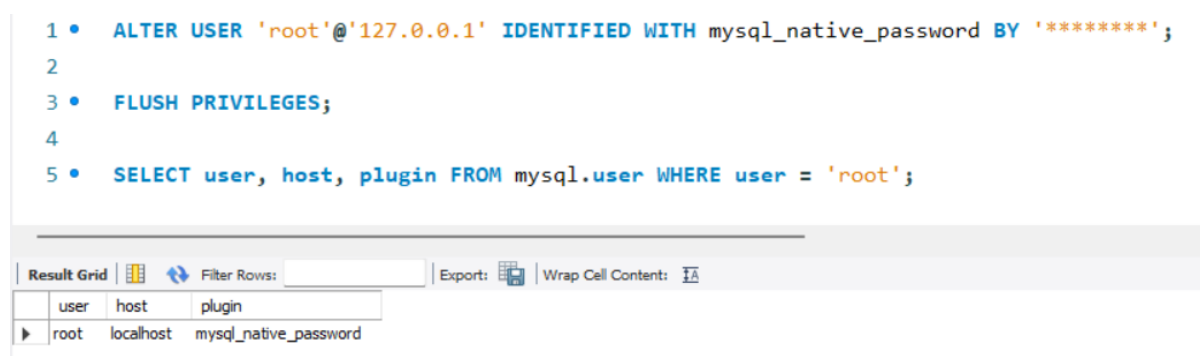


Figure 5.1: Configuration of MySQL User Authentication Plugin.

5.1.2 View Execution: Inventory Status

The execution of the `v_books_status` view (Figure 5.2) provides a consolidated look at the library's inventory. By joining the `Book` and `Category` tables, the system displays human-readable category names (e.g., Science, IT) instead of numeric IDs.

Key Insight: The calculated column `availability_pct` offers immediate insight into stock levels. For instance, *The Origin of Species* shows **100% availability** (5/5 copies), indicating low current demand, while *Sapiens* shows **0.0%**, signaling that all copies are currently loaned out. This validates the view's logic in supporting inventory management decisions.

```

1  -- check view_book_status
2  • SELECT * FROM v_books_status
3  ORDER BY book_id ASC;
4

```

	book_id	title	category_name	total_copies	available_copies	availability_pct
1	1	A Brief History of Time	Science	5	5	100.0
2	2	Cosmos	Science	3	2	66.7
3	3	The Selfish Gene	Science	4	4	100.0
4	4	Silent Spring	Science	2	1	50.0
5	5	The Origin of Species	Science	5	5	100.0
6	6	The Great Gatsby	Literature	5	3	60.0
7	7	Pride and Prejudice	Literature	4	4	100.0
8	8	To Kill a Mockingbird	Literature	6	2	33.3
9	9	1984	Literature	8	5	62.5
10	10	Moby Dick	Literature	3	3	100.0
11	11	Clean Code	IT	10	8	80.0
12	12	The Pragmatic Progra...	IT	7	5	71.4
13	13	Introduction to Algori...	IT	5	2	40.0
14	14	Design Patterns	IT	6	6	100.0
15	15	Head First Java	IT	8	7	87.5
16	16	Rich Dad Poor Dad	Business	10	2	20.0
17	17	Think and Grow Rich	Business	5	5	100.0
18	18	Zero to One	Business	4	3	75.0
19	19	The Lean Startup	Business	6	4	66.7
20	20	Principles	Business	3	3	100.0
21	21	Sapiens	History	12	0	0.0

Figure 5.2: Query results from `v_books_status` showing availability percentages.

5.1.3 Complex Query: Most Borrowed Books

The query identifying the most borrowed books successfully aggregates data from the `Loan` table (Figure 5.3).

Analysis: The results show that books like *A Brief History of Time*, *Cosmos*, and *The Selfish Gene* each have **2 recorded loans**. This query proves that the system can accurately track borrowing frequency by joining `Book`, `Loan`, and `Category` tables and performing a `COUNT()` aggregation grouped by book title. This metric is essential for deciding which books to restock.

5	-- check most borrowed books:
6	• SELECT
7	b.book_id,
8	b.title,
9	c.category_name,
10	COUNT(l.loan_id) AS times_borrowed
11	FROM Book b
12	JOIN Loan l ON b.book_id = l.book_id
13	JOIN Category c ON b.category_id = c.category_id
14	GROUP BY b.book_id, b.title, c.category_name
15	ORDER BY times_borrowed DESC;
16	

Result Grid		Filter Rows:	<input type="text"/>	Export:	Wrap Cell Content:
	book_id	title	category_name	times_borrowed	
▶	1	A Brief History of Time	Science	2	
	2	Cosmos	Science	2	
	3	The Selfish Gene	Science	2	
	4	Silent Spring	Science	2	
	5	The Origin of Species	Science	2	
	6	The Great Gatsby	Literature	2	
	7	Pride and Prejudice	Literature	2	
	8	To Kill a Mockingbird	Literature	1	
	9	1984	Literature	1	
	10	Moby Dick	Literature	1	
	11	Clean Code	IT	1	
	12	The Pragmatic Progra...	IT	1	

Figure 5.3: SQL Query results identifying the most borrowed books.

5.1.4 Trigger Validation: Automated Fine Calculation

The impact of the `trg_after_loan_update` trigger is evidenced in the query results from the `Payment` table (Figure 5.4).

Mechanism Verification: The table displays records with `Days_Late` ranging from 3 to 8 days. Crucially, the `Fine_Amount` corresponds perfectly to the formula:

$$Fine = Days_Late \times \$2.00$$

- Example: A delay of **6 days** resulted in a **\$12.00** fine.
- Example: A delay of **3 days** resulted in a **\$6.00** fine.

Conclusion: This confirms that the trigger logic is functioning correctly, automatically calculating fines and inserting records without manual intervention.

28	-- check trigger impact:
29	• SELECT
30	p.amount as Fine_Amount,
31	p.remarks,
32	m.member_id,
33	m.full_name
34	FROM Payment p
35	JOIN Loan l ON p.loan_id = l.loan_id
36	JOIN Member m ON l.member_id = m.member_id
37	WHERE p.remarks LIKE '%Late%' OR p.remarks LIKE 'Auto-fine%';
38	
39	

Result Grid	Filter Rows:	Export:	Wrap Cell Content:

	Fine_Amount	remarks	member_id	full_name
▶	10.00	Late fee (5 days)	1	John Smith
	12.00	Late fee (6 days)	2	Emily Davis
	10.00	Late fee (5 days)	3	Michael Brown
	16.00	Late fee (8 days)	4	Sarah Wilson
	12.00	Late fee (6 days)	5	Kevin White
	14.00	Late fee (7 days)	6	Laura Martin
	6.00	Late fee (3 days)	7	Chris Lee
	14.00	Late fee (7 days)	8	Anna Scott

Figure 5.4: Payment table records generated automatically by the Trigger.

5.2 Python Integration & Reporting

5.2.1 Python Console Output

The successful connection between the Python application and the MySQL database is demonstrated in Figure 5.5. The script executes without errors, confirming that the modular connection logic (`create_connection`) and configuration parameters are correct. This establishes a reliable pipeline for fetching data for further analysis.

```

1  import pymysql
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import seaborn as sns
5
6  sns.set_theme(style="whitegrid")
7  plt.rcParams['figure.figsize'] = (10, 6)
8
9  # =====
10 # DATABASE CONNECTION
11 # =====
12 def get_connection():
13     return pymysql.connect(
14         host="127.0.0.1",
15         user="root",
16         password="*****",
17         database="LibraryManagementDB"
18     )

```

Figure 5.5: Python script execution ensuring database connectivity.

5.2.2 Monthly Activity Report

The output from the `sp_monthly_activity_report` stored procedure visualizes the library's performance for **June 2025** (Figure 5.6).

Data Interpretation:

- **Total Borrowed (3):** Indicates moderate borrowing activity for the month.
- **Total Returned (3):** Matches the borrowing count, suggesting a balanced circulation flow.
- **Total Overdue Returns (2):** A significant portion (2 out of 3 returns) were late, highlighting a potential issue with borrower compliance.
- **Total Fines Collected (\$22.00):** This financial metric aggregates the penalties from the overdue returns, proving the procedure's ability to sum data from the `Payment` table.

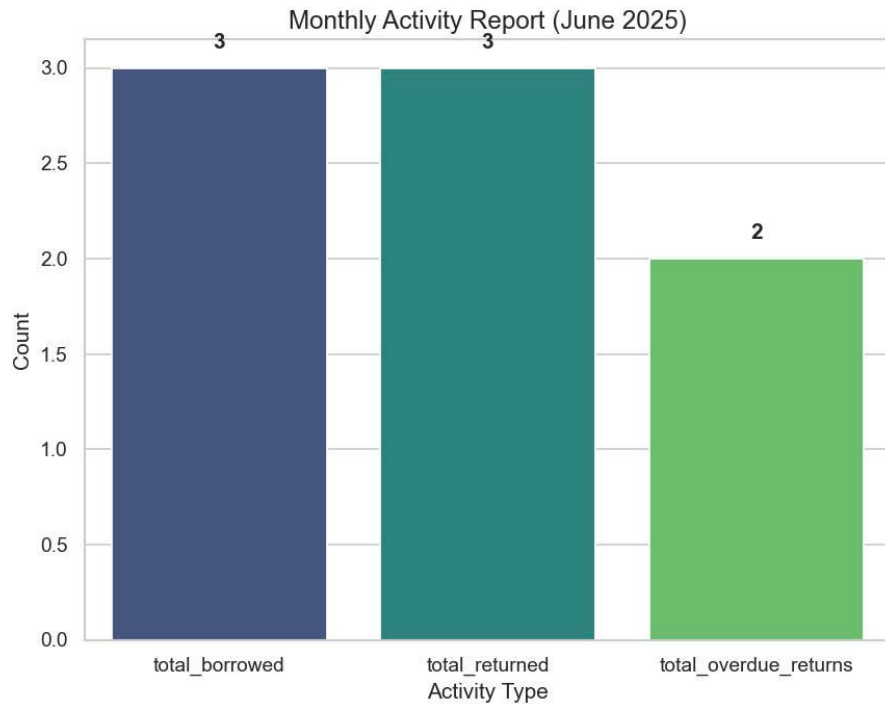


Figure 5.6: Visualization of the Monthly Activity Report (June 2025).

5.3 Data Visualization & Analytics

5.3.1 Inventory Distribution Analysis

The Grouped Bar Chart (Figure 5.7) offers a comparative analysis of inventory across categories. The Grey bars represent total stock, while Green bars represent available stock.

Insight: The **IT** category has the highest total volume (36 books) and high availability (28 books). In contrast, the **History** category shows a tighter margin (28 total vs. 13 available), suggesting that History books are currently more popular or slower to return than IT books.

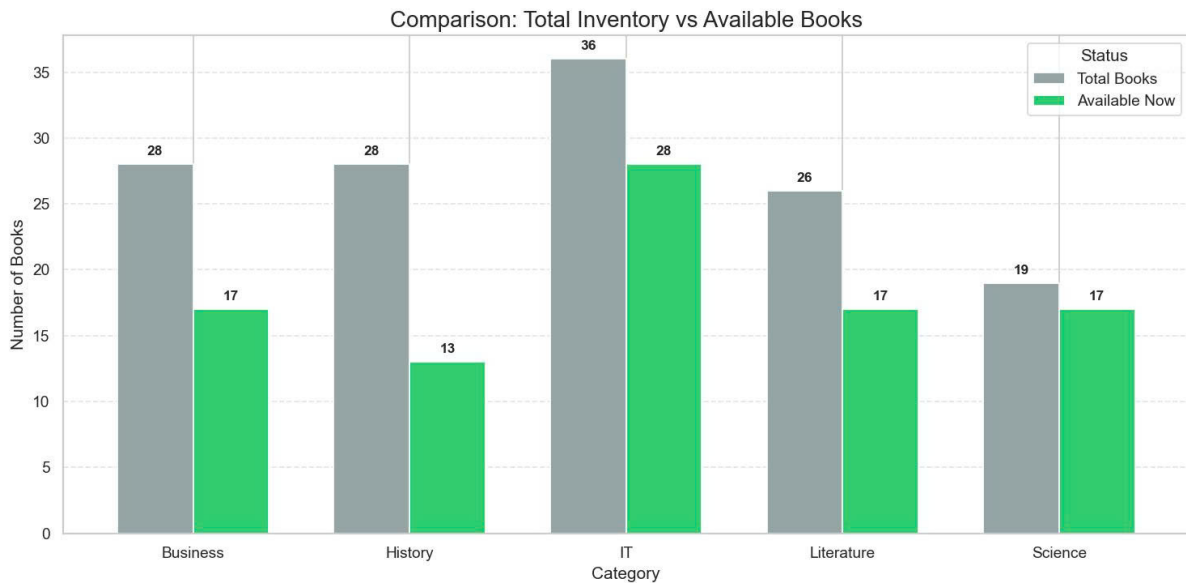


Figure 5.7: Inventory Status by Category: Total vs. Available Books.

5.3.2 Active Loans Status

The Pie Chart (Figure 5.8) illustrates the ratio of **Active** loans to **Overdue** loans for the current period.

Analysis: The chart displays a distribution of approximately **63.6%** **Active** loans versus **36.4%** **Overdue** loans.

- **Compliance Indicator:** The majority of current loans (nearly 64%) are within their allowable borrowing period, indicating a generally healthy circulation flow.
- **Risk Assessment:** However, the overdue segment (36.4%) remains significant. This metric serves as a key performance indicator (KPI) for library staff, suggesting that while most users comply, a substantial number of items are not returned on time, validating the necessity of the automated fine calculation triggers implemented in the system.

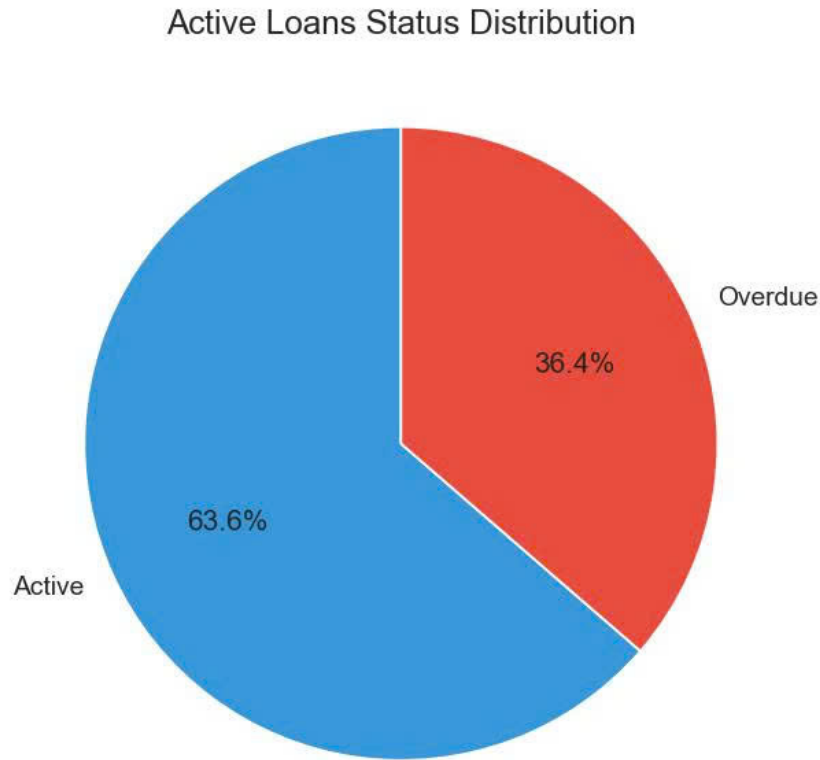


Figure 5.8: Active Loans Status Distribution.

5.3.3 Trigger Logic Linear Regression

The Scatter Plot (Figure 5.9) provides a mathematical validation of the fine calculation logic.

- **Linearity:** The plot shows a perfect positive linear relationship between “Days Overdue” (X-axis) and “Fine Amount” (Y-axis).
- **Slope Validation:** The slope of the line corresponds exactly to the fine rate (\$2.00/day). For example, at $X = 5$, $Y = 10$; at $X = 8$, $Y = 16$.
- **Significance:** This visualization proves that the system creates data that is **consistent, predictable, and fair**. There are no outliers or calculation errors in the database trigger logic.

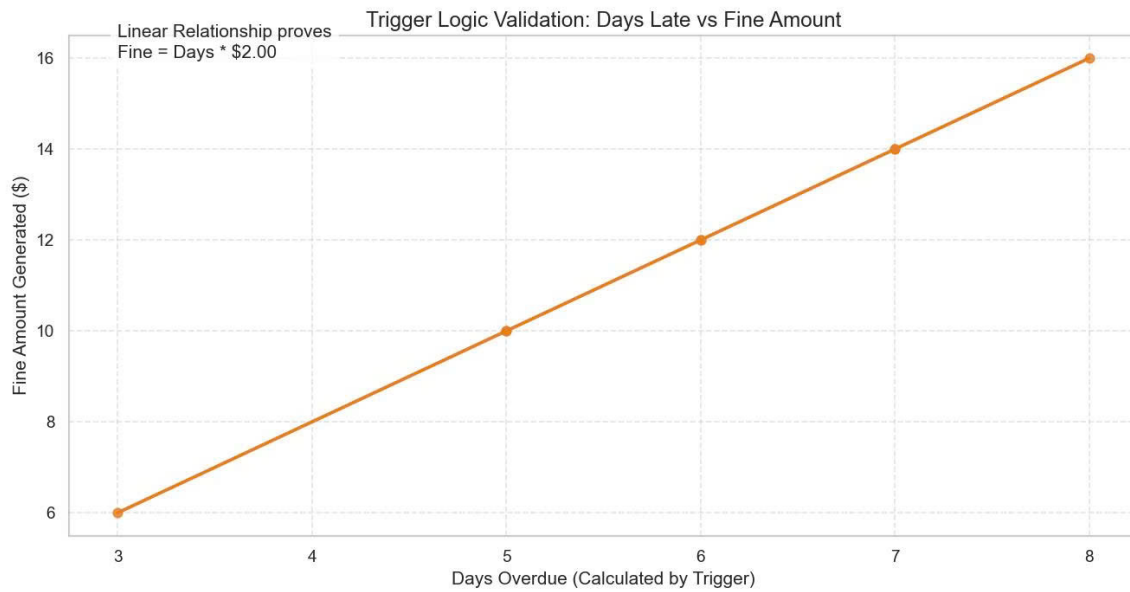


Figure 5.9: Trigger Logic Validation: Linear relationship between Days Late and Fine Amount.

Summary of Results

The combination of SQL query results and Python visualizations confirms that the **Library Information Manager** has been implemented successfully. The system correctly enforces business rules (via Triggers), supports complex data retrieval (via Views and Joins), and enables insightful decision-making through automated reporting (Stored Procedures and Visualizations).

Conclusion

The development of the **Library Information Manager** project has successfully demonstrated the end-to-end process of designing, implementing, and validating a relational database system. By strictly adhering to database design principles and integrating them with modern programming tools, the system achieves a high level of reliability and automation.

Project Summary

The primary objective was to resolve the data inconsistencies and manual processing inefficiencies inherent in the library's previous operations. Through the deployment of a centralized MySQL database, the project has established a "Single Source of Truth" for all library assets, members, and transactions. The system effectively manages the complex lifecycle of book loans, from issuance to return and fine calculation.

Key Technical Achievements

The success of this project is defined by several critical technical milestones:

- **Robust Architecture (3NF Compliance):** The database schema was rigorously normalized to the Third Normal Form (3NF). Notably, the separation of **Category** and the optimization of the **Payment** table eliminated data redundancy and transitive dependencies, ensuring long-term data integrity.
- **Automated Business Logic:** By implementing Event-Driven Architecture via **Database Triggers**, the system automates critical tasks such as inventory synchronization and overdue fine calculations ($Fine = Days \times \$2.00$). This minimizes human error and ensures that business rules are enforced consistently at the database level.
- **Data Abstraction & Security:** The use of **Stored Procedures** and **Views** provides a secure and simplified interface for application access. Complex joins and aggregations are handled on the server side, optimizing performance and shielding the raw table structures from direct manipulation.
- **Application Integration & Analytics:** The seamless integration with Python (via `pymysql`) demonstrates the system's interoperability. The generated visualizations—specifically the linear regression of trigger logic and the inventory distribution charts—provide concrete evidence that the underlying data logic is mathematically accurate and operationally sound.

Final Remarks

In conclusion, the **Library Information Manager** is not merely a data storage solution but a comprehensive operational engine. It successfully transforms raw data into actionable insights through automated reporting and visualization. The system is robust, scalable, and ready to support the library's day-to-day operations with efficiency and precision.