# Library Management System – Database & Program Architecture Report

The library management system I designed is built around a relational SQLite database dynamically controlled through a structured Python program. The program acts as both a data-entry controller and a validation layer, while the database serves as the central storage system. Together, these components create a complete information architecture where multiple entities—authors, books, borrowers, loans, and book copies—are interconnected through foreign keys and supported by program-level logic.

## 1. Database Structure: Entities and Relationships

My library_database.db contains five functional tables: Authors, Library_database (books), Borrowers, Loans, and Book_Locations. These tables work together to represent the core parts of a real library system. Each table stores a specific type of information, and their relationships allow the system to track how all items interact. **Authors Table:**
This table stores all authors with unique Author_IDs, forming a one-to-many relationship since one author may write many books. **Library_database Table (Books):**
This table stores each book's title, genre, page count, and publication date. Each book includes an Author_ID foreign key linking it to the Authors table. My .db stores publication dates as REAL due to an earlier version of the system, but SQLite's flexible typing ensures full functionality. **Borrowers Table:**
This table stores unique Borrower_IDs representing all people who may borrow books. A single borrower may appear in many loan records. **Loans Table:**
This table acts as the transactional layer linking books to borrowers. It contains Book_ID, Borrower_ID, the loan date, and the return date. This implements a many-to-many relationship through a bridging structure that avoids duplicated data. **Book_Locations Table:**
This table tracks where book copies are stored and how many exist. One book may appear in multiple physical locations.

## 2. How the Python Program Controls and Integrates the Database

The Python program I created functions as both the front-end interface and the logic controller. It connects to the database, creates missing tables when necessary, and uses EasyGUI to ensure clean data entry. It validates: – Date formats (YYYY-MM-DD)
– Positive integers
– Required fields
This prevents invalid or inconsistent data from entering the system.

## 3. Data Flow & Interconnections During Operations

**Adding a Book:**
The program retrieves all authors from the database, ensures that I select a valid author, and then inserts the book with the correct Author_ID. **Adding a Loan:**
The program retrieves all books and borrowers, ensuring that my selections are always valid. The loan entry contains Book_ID, Borrower_ID, and relevant dates. **Viewing Data:**
The program performs SQL JOIN operations to show merged information across tables.
Example:
*SELECT L.Book_ID, L.Title, A.Author_Name FROM Library_database L LEFT JOIN Authors A ON L.Author_ID = A.Author_ID*
Loans also display borrower names and book titles through multi-table joins.

## 4. Integrity, Structure, and System Design

Even though SQLite's PRAGMA foreign_keys = ON is not activated, my program maintains validity by requiring users to select existing authors, books, borrowers, and locations. This ensures that all relationships remain consistent. The combination of relational structure, enforced selection, validation, SQL joins, and autoincrement keys creates a reliable and logically connected database architecture.

## Conclusion

The SQL library database I designed is a structured relational system where each entity supports the others. The Python program serves as the intelligent controller, ensuring that relationships remain valid and meaningful. This design mirrors real-world library management systems, with clear separation of responsibilities and smooth, consistent data flow across components.